EPFL

Thèse n° 10 544

On the Theory and Practice of Modern Secure Messaging

Présentée le 25 avril 2024

Faculté informatique et communications
Laboratoire de sécurité et de cryptographie
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Daniel Patrick COLLINS

Acceptée sur proposition du jury

Prof. R. Guerraoui, président du jury
Prof. S. Vaudenay, directeur de thèse
Prof. P. Rösler, rapporteur
Dr B. Poettering, rapporteur
Prof. M. Kapralov, rapporteur

École polytechnique fédérale de Lausanne

2024

To mum and dad.

Abstract

Billions of people now have conversations daily over the Internet. A large portion of this communication takes place via secure messaging protocols that offer "end-to-end encryption" guarantees and resilience to compromise like the widely-used Double Ratchet protocol of Perrin and Marlinspike. This thesis explores secure messaging from a cryptographic perspective in both the analysis and improvement of existing messaging solutions as well as the design of protocols with new security and efficiency characteristics.

The first half of the thesis considers communication between *two* parties. We first draw our attention to the impending threat of quantum computers on Diffie-Hellman-based key exchange protocols, and in particular on the widely used Extended Triple Diffie-Hellman (X3DH) key exchange protocol of Perrin and Marlinspike. We propose a new deniable authenticated key exchange protocol, K-Waay, that is based on the relatively conservative plain learning-with-errors (LWE) assumption and is faster than previous proposals. We then consider *active attack detection* to ensure parties can detect if and when they have been compromised and impersonated by an adversary, even if messages can be delivered out-of-order like in the Double Ratchet protocol. We consider both in-band and out-of-band detection (the latter offering better security but being less convenient for users) and prove formally that immediate active attack detection is sometimes inherently expensive but, despite this, a relaxed yet meaningful notion of active attack detection can be achieved practically.

The second half of this thesis then deals with communication between a dynamic *group* of parties. Firstly, we formalise the *group administration* problem where a (dynamic) portion of a given group is entrusted with additional privileges: we identify and formalise their core role of enforcing *access control*. We propose two protocols extending the *continuous group key agreement* methodology underpinning the recent IETF Messaging Layer Security (MLS) standard, and demonstrate experimentally that administration can be achieved with very little overhead for MLS. Finally, we formalise the practical Sender Keys group messaging protocol used by WhatsApp and Signal (which in fact relies on two-party communication at its core) and prove in a new security model that the core protocol structure is sound. Through our formalisation, we report some drawbacks of Sender Keys, especially in terms of its resilience to state compromise, and propose some tweaks to overcome them using standard cryptographic primitives, each of which either incurs little overhead or in fact improves practical efficiency.

Résumé

De nos jours, des milliards de personnes utilisent régulièrement internet pour communiquer. Une grande partie de ces communications utilisent des protocoles de messagerie sécurisée qui offrent des garanties de “chiffrement de bout en bout” comme le très répandu protocole Double Ratchet de Perrin et Marlinspike. Dans cette thèse, nous explorons ainsi la notion de messagerie sécurisée. Nous étudions à la fois des solutions existantes mais nous proposons aussi de nouveaux protocoles supportant de nouvelles contraintes de sécurité et d'efficacité.

La première moitié de cette thèse porte sur la communication entre deux parties. Nous portons d'abord notre attention sur la menace imminente que les ordinateurs quantiques représentent pour les protocoles d'échange de clés basés sur le protocole Diffie-Hellman. Le protocole d'échange de clés Extended Triple Diffie-Hellman (X3DH) de Perrin et Marlinspike, qui est largement utilisé, y est particulièrement vulnérable. Nous proposons ainsi un nouveau protocole, K-Waay, qui permet l'échange de clés authentifiées tout en offrant la possibilité de déni plausible. Notre protocole est plus rapide que les propositions précédentes basées, et repose sur l'hypothèse que le problème d'apprentissage avec erreurs (LWE) est difficile. Nous examinons ensuite *la détection d'attaques* afin de garantir que les deux parties puissent détecter si un adversaire a usurpé une identité ou s'il a réussi à les compromettre, et cela même si les messages sont reçus dans le désordre, comme dans le protocole Double Ratchet. Nous considérons à la fois la détection en bande et hors bande (cette dernière offrant une meilleure sécurité mais étant moins pratique). Nous concluons que la détection immédiate d'une attaque active est parfois intrinsèquement coûteuse mais que, malgré cela, il est possible de la rendre pratique et efficace en relaxant légèrement le modèle de sécurité.

La seconde partie de cette thèse traite des communications entre un groupe dynamique de parties. Nous commençons par formaliser le problème *d'administration de groupe* où un sous-ensemble (dynamique) d'un groupe donné se voit confier des privilèges supplémentaires : nous identifions et formalisons leur rôle principal qui est de gérer le contrôle d'accès. Nous proposons deux protocoles étendant la méthodologie *d'accord de clé de groupe continu* qui étaye la nouvelle norme Messaging Layer Security (MLS) de l'IETF, et nous démontrons expérimentalement que l'administration peut être réalisée avec un surcoût faible pour MLS. Enfin, nous formalisons le protocole de messagerie de groupe Sender Keys utilisé par WhatsApp et Signal (qui repose sur une communication bipartite). Nous prouvons dans un nouveau modèle de sécurité que le protocole est robuste. Grâce à notre formalisation, nous signalons

certains inconvénients de Sender Keys, en particulier en matière de résilience face à un adversaire étatique, et nous proposons des ajustements basés sur des primitives cryptographiques standardisées afin de les surmonter.

Acknowledgements

I would first like to thank my advisor, Serge Vaudenay, for his continued guidance over the last four years, for taking a chance on me, and for allowing me to flourish in his lab LASEC. Thank you also to my private defence jury – Serge, Rachid Guerraoui, Michael Kapralov, Bertram Poettering and Paul Rösler – for their valuable time and feedback.

Thank you to the current members of LASEC. Loïs, my officemate of four years, thank you for our endless conversations, hilarious in-jokes and fun collaborations. Thank you Boris for your advice and company, Bénédikt for our conversations and your insights, Abdullah for your kindness and for bonding with me over our shared interests, and Laurane for our various activities and many enjoyable chats. Thank you Sylvie for your support of both me and the lab.

Thank you to the former and spiritual members of LASEC. Andi, for bonding over pop culture, projects and everything in between; Khashayar, for our variety of conversations and the memorable lifts home; Ritam, for always being up for cake and to chat; Simone, for all the memories – the travel, collaborations and friendship. David and Phillip, I'm glad our collaborations and relationship did not end when you left LASEC. Thank you Subhadeep for your guidance and companionship when it was just us two in the evenings. Thank you Khanh for your friendship. Thank you Aymeric, Novak, Helen, Fatih and Gwangbae – your time at LASEC was valuable to me. Martine, thank you for your help and service over the years. Thank you to my project students for the opportunity to advise you and to the lab interns and visitors over the years.

Thank you to Rachid and Thanasis for your semester project supervision – distributed computing is still in my heart – and Jovan for your friendship.

A special thank you to Dario, Giulio, Julian and Paul for each hosting me in their labs for research visits – all of these trips were special and productive, and it was great to spend time with you and everyone else who I met during these visits. Thank you to Julian for taking me under your wing and Paul for mentoring me.

This thesis, and my research, would have been impossible without the effort and input of my collaborators, both past and present, and all those who supported me in the cryptographic community – a particular shoutout to those from Bochum, CISPA and IMDEA.

Acknowledgements

Thank you to all of the friends I met in Lausanne that helped met through my PhD since I arrived in September 2019. Guohao, for the endless conversations (we have had several since I started writing these acknowledgements) and irreplaceable friendship; Plouton, for always outdoing yourself in cooking and your warm companionship; Rodrigo, for your integrity, honesty and strength. Bakul, for teaching me how to be a roommate. ML4ED, for your companionship on the second floor of INF – thank you Jade for visiting Loïs and I! Christian and Sylvain, thanks for the memories. Nicolas, coucou! Greg, thanks for your friendship. Aditya, Dina, Lars, Michal, Valentin and all those I've forgotten, merci!

Thank you to my Honours thesis advisor Vincent Gramoli for introducing me to and encouraging me in the world of research. Thank you to Tyler Crain for your formative guidance and for encouraging me to travel for my PhD, without whom this would not have happened. Thank you to all my other colleagues, peers and friends from the University of Sydney. Thank you to my teachers in school and university.

I would be remiss not to mention my friends, family and mentors who I met and grew up with in Sydney and beyond. Thank you to my mum, dad and brother for your endless love and support, to Tash and Joey, to Ash, Chris, Eric, Frank, Ian, and everyone else – I have missed you all dearly.

Finally, I would like to thank all those who I did not thank above, as there were many more who impacted my life and helped me.

Lausanne, April 9, 2024 D. C.

Contents

Abstract (English/Français) i

Acknowledgements v

1 Introduction 1
- 1.1 Context and Motivation . . . 1
- 1.2 Contribution . . . 4
  - 1.2.1 Part I: Two-Party Communication . . . 4
  - 1.2.2 Part II: Group Messaging . . . 7
- 1.3 List of Works . . . 11

2 Preliminaries 13
- 2.1 Notation and Conventions . . . 13
- 2.2 Cryptographic Primitives . . . 15
  - 2.2.1 Key-Encapsulation Mechanism (KEM) . . . 15
  - 2.2.2 Signatures . . . 16
  - 2.2.3 Pseudorandom Function (PRF) . . . 17
  - 2.2.4 Hash Function . . . 18
  - 2.2.5 Incremental Set Hash Function . . . 18
  - 2.2.6 Symmetric Encryption . . . 19

I Two-Party Communication 23

3 K-Waay: Fast and Deniable Post-Quantum X3DH Without Ring Signatures 25
- 3.1 Contribution . . . 25
  - 3.1.1 Summary . . . 27
  - 3.1.2 Technical Overview . . . 29
  - 3.1.3 Additional Related Work . . . 33
- 3.2 Split-KEM . . . 33
  - 3.2.1 Security . . . 34
  - 3.2.2 Deniability . . . 37
- 3.3 Deniable Authenticated Key Exchange . . . 38
  - 3.3.1 Syntax . . . 38
  - 3.3.2 Security Model . . . 39

3.3.3 Deniability . . . 44
3.4 K-Waay: Post-Quantum X3DH from Split-KEM . . . 45
3.4.1 Construction . . . 45
3.4.2 Security . . . 47
3.5 Deniable Split-KEM from Lattices . . . 56
3.5.1 Lattice Toolbox . . . 56
3.5.2 Extended-LWE . . . 57
3.5.3 Construction . . . 61
3.5.4 Security Analysis . . . 61
3.5.5 Building a UNF-1KCA and IND-1BatchCCA Split-KEM . . . 69
3.5.6 Concrete Instantiation . . . 70
3.6 Evaluation and Discussion . . . 72
3.6.1 Benchmarks . . . 73
3.6.2 Advantages, Limitations and Discussion . . . 76

4 On Active Attack Detection in Messaging with Immediate Decryption 79
4.1 Contribution . . . 79
4.1.1 Summary . . . 81
4.1.2 Technical Overview . . . 82
4.1.3 Additional Related Work . . . 85
4.2 (Authenticated) Ratcheted Communication . . . 85
4.3 In-Band Active Attack Detection: RID . . . 91
4.3.1 RID-Secure RC . . . 94
4.4 Out-Of-Band Active Attack Detection: UNF . . . 99
4.4.1 UNF-Secure ARC from a RID-Secure RC . . . 100
4.4.2 UNF-Secure ARC from Any RC . . . 102
4.5 Lower Bounds for Active Attack Detection . . . 105
4.5.1 Communication Cost for r-RID Security . . . 105
4.5.2 Communication Cost for r-UNF Security . . . 111
4.6 Optimisations and Performance/Security Trade-Offs . . . 111
4.6.1 On the Practicality of s-RID and s-UNF Security . . . 112
4.6.2 Epoch-Based Optimisation for s-RID Security . . . 113
4.6.3 Pruning for UNF Security . . . 115
4.6.4 Lightweight Bidirectional Authentication . . . 119

II Group Messaging 125

5 Cryptographic Administration for Secure Group Messaging 127
5.1 Contribution . . . 127
5.1.1 Summary . . . 129
5.1.2 Technical Overview . . . 130
5.1.3 Additional Related Work . . . 132

5.2 (Administrated) Continuous Group Key Agreement . . . 132
5.2.1 Continuous Group Key Agreement . . . 132
5.2.2 Administrated CGKA . . . 136
5.2.3 Correctness . . . 137
5.2.4 Security . . . 140
5.3 A-CGKA Constructions . . . 145
5.3.1 Individual Admin Signatures . . . 146
5.3.2 Dynamic Group Signature . . . 159
5.3.3 Description . . . 160
5.3.4 Integrating A-CGKA into MLS . . . 169
5.4 Evaluation and Discussion . . . 170
5.4.1 Benchmarks and Performance . . . 171
5.4.2 Modelling in Related Work . . . 175

6 WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs 177
6.1 Contribution . . . 177
6.1.1 Summary . . . 179
6.1.2 Technical Overview . . . 180
6.1.3 Additional Related Work . . . 183
6.2 Two-Party Channels . . . 184
6.2.1 Primitive Definition and Correctness . . . 184
6.3 Group Messenger . . . 191
6.3.1 Security Model . . . 194
6.3.2 Modelling Two-Party Channel Ciphertexts . . . 197
6.4 Sender Keys . . . 198
6.4.1 Protocol . . . 198
6.4.2 Security . . . 201
6.5 Analysis and Improvements . . . 206
6.5.1 Security Analysis and Limitations . . . 206
6.5.2 Proposed Improvements: Sender Keys+ . . . 209
6.5.3 Sender Keys/Sender Keys+ vs CGKA . . . 212
6.5.4 Sender Keys in Practice . . . 213
6.6 Sender Keys and Sender Keys+: Full Protocols and Security . . . 214
6.6.1 Protocol Descriptions . . . 214
6.6.2 Sender Keys Security . . . 219
6.6.3 Sender Keys+ Security . . . 226

7 Conclusion 231
7.1 Primitive Summary . . . 231
7.1.1 Two-Party Communication . . . 231
7.1.2 Group Communication . . . 232
7.2 Discussion and Future Work . . . 233

7.2.1 Composability . . . 233
7.2.2 Model Limitations . . . 233
7.2.3 Unification and Verification . . . 234
7.2.4 Deniable Post-Quantum X3DH (Chapter 3) . . . 234
7.2.5 Active Attack Detection in Messaging (Chapter 4) . . . 235
7.2.6 Group Administration (Chapter 5) . . . 236
7.2.7 Sender Keys (Chapter 6) . . . 238

A Appendices **241**
A.1 QROM Preliminaries . . . 241
A.2 Proof of Theorem 5 (Chapter 3) . . . 243
A.2.1 Proof in the QROM . . . 243
A.2.2 Proof in the ROM . . . 246
A.3 Proof of Theorem 6 (Chapter 3) . . . 247
A.3.1 Proof in the ROM . . . 247
A.3.2 Proof in the QROM . . . 248
A.4 Tables for Sender Keys (Chapter 6) . . . 252

Bibliography **277**

Curriculum Vitae **279**

1 Introduction

1.1 Context and Motivation

For thousands of years, the primary aim of cryptography has been to enable secret communication between physically separated parties. In 1976, Diffie and Hellman proposed an elegant and now ubiquitous method for enabling two parties to share a secret over an untrusted network [DH76]. As a result of the ensuing cryptographic revolution, cryptography is used today by billions of people and countless computers each day to protect communication via protocols like Transport Layer Security (TLS) [Res18] and, more recently, secure messaging applications.

Secure messaging, the focus of this thesis, has exploded in popularity and importance in the last decade. This has been enabled by two main factors. Firstly, billions of people now own and use smartphones, which enable highly convenient, fast and user-friendly digital conversations. Secondly, and in connection with this, there has been an increased awareness of mass surveillance, both in and especially outside of the security community, spearheaded most notably by Edward Snowden's whistleblowing in 2013 [Gua13]. The documents that Snowden leaked to the public revealed that the National Security Agency (NSA), among other state actors, ran (and undoubtedly continue to run) extensive national and international spying and surveillance programs. Looking ahead, this motivates our consideration of advanced attack vectors like secret state compromise in this thesis [BSJ+15].

Secure messaging shares many of the same security requirements as typical communication protocols like TLS. Firstly, communication must be *confidential*, preventing an adversary from deducing anything more from a communication transcript than possibly some metadata. *Authentication* ensures that two (or more) parties can be sure that they are communicating with each other and not some unintended party. Finally, *integrity* ensures that communication cannot be feasibly tampered with by a network adversary without detection.

Messaging nevertheless imposes more requirements than say TLS, motivating specialised protocol design. For one, messaging sessions between two parties are expected to last for

months or years, whereas TLS sessions are typically on the order of minutes. Due to this and the fact that smartphones, the primary medium for instant messaging, have a particularly large attack surface, state *exposure* (or *compromise* [CCG16]) is inevitable for many users. Thus, resilience to state exposure has become a central requirement both in the literature and in practice alike. Moreover, users are not expected to be online at all times, so users should be able to communicate asynchronously as smoothly as possible (thus precluding multi-round protocols for sending and receiving). In popular messaging systems like WhatsApp, an intermediate service (or *central server* for simplicity) manages communication between all parties that in particular forwards messages to users whenever they are online. Some level of plausible deniability, or simply *deniability*, is also sometimes desirable, allowing users to deny having either used a message system or simply having had a particular conversation.[1]

For dealing with state exposure in secure messaging, two complementary security notions have become standard in the aftermath of the Snowden revelations. Firstly, *forward security* (FS) [Gün90, BG21] ensures that, upon state exposure, communication *prior* to the exposure is still secure. To achieve FS, secret keying material must somehow be updated over time and new keys replaced with old ones. We will assume in this thesis that secure memory/state erasures are possible since otherwise old keys are always accessible to the adversary upon compromise, preventing FS. Secondly, *post-compromise security* (PCS) [CCG16] ensures that upon state exposure, while security in the present is lost, parties can eventually and automatically *restore* security such that the effect of state compromise is reduced or kept to a minimum. Indeed, some form of PCS has been achieved in practice many years before it was studied in the context of messaging through the common "good practice" of key rotation or re-establishing secure channels after some time or on-demand.[2] As computing power has advanced significantly over the last few decades, however, it is now possible to cheaply and *continuously* update keying material, enabling strong FS and PCS guarantees. An important aspect of research is thus to identify inherent limitations in achieving FS and PCS and to identify salient performance/security trade-offs for practice.

In the simpler case of *two-party* messaging, we assume both parties, which we will sometimes refer to as Alice and Bob, send and receive messages between each other. Modern messaging protocols are descendants of the seminal Off-The-Record (OTR) protocol [BGB04], which provides FS and PCS through the use of continuous Diffie-Hellman key exchanges. OTR was a significant step forward at the time since popular online messaging applications like MSN Messenger and AOL Instant Messenger provided no end-to-end security guarantees for users. Inspired by OTR, the Double Ratchet [PM16, ACD19] component of the Signal protocol (previously TextSecure [FMB+16]) was proposed by Perrin and Marlinspike. It has since become the de-facto standard for messaging [EM19], used by applications including

[1] In this thesis, we mainly consider deniability on the *protocol* level, e.g., constructing a deniable key exchange protocol; achieving deniability on the *system* level or in a fully *practical* sense is more complex and sensitive, and to some extent is more of a social or legal matter than a strictly technical one [CCHD23].

[2] Terms to describe post-compromise security in the literature include future secrecy, backward secrecy and self-healing [DV19].

WhatsApp, Signal, Facebook Secret Conversations and Wire. In a nutshell, the Double Ratchet extends OTR by employing a second ratchet, a forward-secure symmetric ratchet, in addition to the continuous Diffie-Hellman exchange (the *asymmetric* ratchet) present in both protocols.

In group messaging, the group is now dynamic, meaning parties can be added and removed from a given group and users must agree on the group state and messages over time, complicating protocol design and implementation. A naive but nonetheless functional approach here is for all parties to build group messaging on top of two-party channels, which necessarily incurs $O(n)$ sender communication overhead for every message sent in a group of n users. This was, and in some edge cases still is, the approach taken by the Signal application in practice. For increased efficiency, the Sender Keys protocol [Mar14] used by WhatsApp [Wha20] and Signal [M+16] enables senders to produce constant-sized ciphertexts that are then forwarded to all group members (via so-called *server fan-out*). In these protocols, refreshing all secrets for post-compromise security incurs $O(n^2)$ communication overhead for a group with n users. While Sender Keys was already deployed (as it still is), MLS (Message Layer Security) was being developed by the IETF in a tight collaboration between industry and academia (and was standardised in 2023 [BBR+23]). The MLS protocol, by contrast, achieves logarithmic-sized per-party network overhead in "good" or "fair-weather" cases [ACDT20] for group key updates.[3]

In either setting, parties must somehow authenticate each other before executing the messaging protocol. This is most commonly achieved by assuming that some public key infrastructure (PKI) is made available to all parties with which they can upload long and short-term keying material. In two-party communication, and sometimes group messaging, authentication is realised via authenticated key exchange (AKE) that leverages the PKI to securely establish initial keying material. Due to the combination of long-term and short-term keys used, key exchange protocols can achieve *forward secrecy*, including many based on the Diffie-Hellman key exchange mentioned above, which ensures that exposing long-term keys does not compromise previously established session keys. Nowadays, many messaging systems use the Extended Triple Diffie-Hellman (X3DH) protocol [MP16b], a descendent of the 3DH protocol [KP05], that provides features including asynchronicity, deniability and forward secrecy, but notably not post-quantum security, i.e., security in the presence of an adversary equipped with a quantum computer.[4] In group messaging protocols like MLS, rather than performing pairwise key exchange, parties instead upload keying material to a PKI that is directly used for authentication, e.g., when new members are added to a group. In addition, given the threat of state exposure and impersonation attacks that may go undetected, we will argue later on that it is vital to continuously authenticate the conversation transcript itself.

In this thesis, we view secure messaging in the lens of *provable security*. In provable security, a security definition for some primitive or protocol is proposed and its security is reduced to

[3] This nonetheless degrades to *linear* overhead in the worst case given a dynamic group in which members adaptively update their keying material and the common group key over time.

[4] In 2023, Signal developed and deployed the PQXDH protocol [KS23] that provides security assuming the adversary can perform active attacks using classical computation and passive attacks using quantum computation.

the (conjectured) hardness of mathematical problems like the hardness of factoring or, more generally, to that of abstract cryptographic building blocks. From initial works on provable security in the 1980s onwards, starting from the definition of semantic security [GM82], it has become standard, at least when public-key cryptography is involved, that new constructions are analysed in this framework. Moreover, as the techniques have been more widely understood and developed, increasingly complex protocols and systems are now analysed in terms of provable security. In the context of secure messaging, security definitions here are experiments played between a challenger and an adversary driving protocol execution that can, among other attacks, expose secret keying material during execution. We therefore adopt this framework in order to precisely state and prove security properties about the protocols that we consider in this thesis.

Finally, we observe that works that study secure messaging after authentication often consider either ratcheted (or continuous) key exchange, or secure messaging proper. Throughout this thesis, we focus on either secure messaging or continuous key exchange, depending on context. To build ratcheted key exchange from messaging is straightforward in general as parties can simply sample fresh keys and send them as (encrypted) application messages. One has more freedom designing a protocol in the converse direction; for example, MLS employs a complex key schedule on top of TreeKEM to build group messaging [ACDT21a]. Thus, the two settings are closely related.

1.2 Contribution

In this section, we outline the main contributions of this thesis contained in Chapters 3 to 6. Part I of this thesis, entitled Two-Party Communication, comprises of Chapters 3 and 4, and Part II, entitled Group Messaging, comprises of Chapters 5 and 6. Before these two parts, we introduce in Chapter 2 some notation and notions we use throughout the thesis. After these two parts, we summarise and discuss the findings of this thesis in Chapter 7 before concluding.

1.2.1 Part I: Two-Party Communication

The first part of this thesis consists of Chapter 3, which is concerned with two-party authenticated key exchange suitable for messaging, and Chapter 4, which considers the detection of active attacks in two-party messaging.

K-Waay: Fast and Deniable Post-Quantum X3DH Without Ring Signatures

In Chapter 3, we first consider the problem of authenticated key exchange for bootstrapping two-party channels which is useful for both two-party and group messaging (e.g., for protocols that use two-party channels like Sender Keys). Indeed, Signal's X3DH protocol is widely used in practice and satisfies several unique requirements that make it suitable for messaging.

Apart from *asynchronicity*, it provides a form of *deniability* [VGIK20], which in the case of X3DH allows a party, who is being framed, to deny having participated in a conversation with their counterpart who is framing them. However, since X3DH essentially consists of several parallel Diffie-Hellman key exchanges, it does not provide protection against a passive polynomial-time quantum adversary, and Signal's recent PQXDH protocol does not protect against an *active* quantum adversary.

To rectify this, Brendel et al. [BFG+22b] and Hashimoto et al. [HKKP21, HKKP22] concurrently proposed alternate X3DH protocols that protect against even active quantum attackers. These protocols rely on two-user ring signatures [RST01], or equivalently [HKKP22], a form of designated-verifier signatures, to achieve authentication while ensuring deniability. At present, post-quantum instantiations of these primitives are more complex and their security is less well-understood than primitives like the CRYSTALS-Kyber key encapsulation mechanism (KEM) [BDK+18] recently standardised by the United States National Institute of Standards and Technology (NIST) government agency. For example, existing works that rely on random oracles [BR93] tend not to contain proofs in the *quantum* random oracle model [BDF+11], i.e., a random oracle that the adversary can query in superposition. In addition, proposed parameters for concrete instantiations tend not to be large enough since they tend not to take the security loss of the security proofs into account as we observe in the chapter.

We propose instead to use a slightly modified variant of the Frodo key exchange protocol [BCD+16] to achieve deniable, post-quantum authentication. As such, its security is based on the vanilla learning-with-errors (LWE) assumption, which is more conservative than using structured lattices [ANS23]; for this same reason, the German agency BSI chose to recommend FrodoKEM, a KEM derived from Frodo key exchange, over CRYSTALS-Kyber [BSI23]. In particular, structured lattices may very well have exploitable weaknesses due to their additional mathematical structure [Tea21]. To capture this primitive generically, we significantly extend the security requirements of the split-KEM primitive introduced by Brendel et al. [BFG+20] as their security notions, as we show, are insufficient for building (deniable) authenticated key exchange. In more detail, a split-KEM extends KEM in that encapsulation now requires both a secret key and a public key, and decapsulation uses the corresponding public and secret keys respectively. We introduce an unforgeability notion for split-KEM, which captures the difficulty of crafting a ciphertext without any secrets, as well as a notion for deniability, capturing that split-KEM encapsulation computationally looks the same using either the intended secret and public keys *or* the decapsulation keying material as input.

We formalise an X3DH-like AKE protocol that we call K-Waay that uses split-KEM, ephemeral and long-term KEMs, and signatures which parties use to attest to ephemeral KEM and split-KEM public keys uploaded to a server to be used asynchronously by a given party's counterpart. Our AKE security model, like that of Hashimoto et al. [HKKP21, HKKP22], supports ephemeral state exposure, although it is weaker than theirs since only the ephemeral-ephemeral portion of the key exchange (that overall uses both long-term and ephemeral keys) relies on split-KEM. As we do not model reusable semi-static keys directly, we propose a novel countermeasure in

the (quantum) random oracle model to allow for ephemeral key reuse that prevents ephemeral key exhaustion under adversarial or otherwise extreme conditions. Our deniability notion for AKE strengthens that of Brendel et al. [BFG+22b] by additionally giving the judge or adversary the ephemeral state of the receiver. We also benchmark our key exchange protocol and demonstrate even with our conservative choice of split-KEM instantiation and parameters which capture the security loss from the proofs, that K-Waay is between 4x and 6x faster than existing ring signature-based approaches.

On Active Attack Detection in Messaging with Immediate Decryption

In Chapter 4, we then consider two-party messaging proper where parties are assumed to have successfully completed key exchange (e.g., by using K-Waay). A notable property of Signal's Double Ratchet core is that it supports *immediate decryption* as formalised by Alwen et al. [ACD19] and achieved by a number of recent academic protocols [ACD19, PP22, BRT23, CZ24]. That is, protocol execution should natively support receiving messages in different orders to what they were sent in, as well as fully dropped ciphertexts. This is especially important in challenging network settings like mesh networking [BRT23] and can also be useful for efficiency, e.g., the messaging service provider can choose to buffer sent videos for users.

Given the likelihood of state exposure, active attacks, where the adversary uses the state of a compromised party and impersonates them, pose a significant threat. Messaging systems, including Signal via its safety numbers protocol [Mar17], allow parties to compare their *long-term* keys out-of-band via QR code or manual comparison. Hence, given the out-of-band channel provides authenticity, this allows parties to audit the PKI and detect active attacks affecting these keys. In the event of state compromise, however, the adversary can mount an active attack and inject application messages, in which case there is no cryptographic mechanism in place to detect it. Moreover, without an additional assumption or channel, it is not possible to detect active attacks, since the adversary can block all honest messages sent by a compromised party and replace them with its own. To alleviate this, some solutions based on out-of-band communication have been previously proposed, but either do not fully authenticate the conversation transcript [DH21] or have other drawbacks like requiring multiple rounds of communication and not formally capturing immediate decryption [DGP22].

In contrast, to detect attacks *in-band*, that is, over the same channel as is used for communication, *RECOVER security* was defined by Durak and Vaudenay [DV19] and later extended by Caforio et al. [CDV21]. RECOVER security allows parties to detect that, if a single honest message is able to get through after compromise, that active attacks *can* be detected, even if the adversary has full knowledge of secret states. More precisely, RECOVER security encompasses two complementary notions. First, r-RECOVER guarantees that if Bob receives a forgery, Bob will reject all subsequent honest messages. Second, s-RECOVER guarantees that if Bob receives a forgery, his communication partner Alice will reject all messages sent *by* Bob after he received the forgery. However, their notions and constructions are not compatible

with immediate decryption, and so cannot be readily adopted by systems like Signal.

To remedy these issues, we systematically explore active attack detection for messaging with immediate decryption in both the in-band and out-of-band settings. For RECOVER security, we generalise the two notions above to support immediate decryption, and propose a baseline construction that compiles a messaging scheme into one that is secure under both of our notions. We also propose two analogous security notions assuming parties have access to an authentic out-of-band channel, and construct secure schemes in two ways: first, from a RECOVER-secure messaging scheme, incurring no additional overhead, and secondly, from any messaging scheme. We then show that in order to satisfy our r-RECOVER notion or the analogous out-of-band notion, the ciphertext size must grow linearly in the number of messages sent (and security parameter) assuming unidirectional communication. Our information-theoretic argument implies that our baseline construction, which attaches all sent ciphertexts to each ciphertext to achieve r-RECOVER security, is asymptotically optimal. To bypass this negative result, we argue that s-RECOVER security is comparatively cheaper to achieve and can be sufficient for practice as it still provides r-RECOVER-like guarantees (albeit after an honest round-trip). We propose a series of optimisations, where ultimately the communication overhead of s-RECOVER security is a single hash digest and a few indices (in proportion to how synchronised parties are). Since s-RECOVER security effectively provides 'delayed' r-RECOVER guarantees, we believe that this solution is affordable enough to be adopted in practice by systems like Signal.

1.2.2 Part II: Group Messaging

The second part of this thesis moves to group messaging: Chapter 5 considers the effects of group administration on security, and Chapter 6 explores the Sender Keys protocol used by WhatsApp and Signal.

Cryptographic Administration for Secure Group Messaging

In Chapter 5, we consider a problem that only arises in group messaging, namely *group administration*. In practice in a group conversation, one or more group members are generally afforded more privileges than others. Systems like Telegram offer admins diverse, fine-grained control over the conversation flow, including disallowing certain types of messages from being sent. Fundamentally, though, the core of administration, which is what we target here, is to enforce *access control*. More precisely, we consider a privileged set of users, the group *administrators* (or admins), who solely have the ability to add and remove users from the conversation.

We observe that many practical protocols do not provide adequate protection for access control, and in particular the ability to enforce group administration. The *burgle into the group* attack [RMS18] allows an adversary to trivially forge messages that add and remove parties

from a group, thereby allowing them to completely bypass any cryptography used. This issue notably affected WhatsApp in particular without the need for any state compromise.[5] Similar attacks have been reported for the federated messaging system Matrix [ACDJ23, ADJ24].

In this chapter, we aim to formalise group administration, which provides a natural mechanism to prevent these types of attacks and enable cryptographic access control within a group. To this end, we start from the recent continuous group key agreement (CGKA) primitive [ACDT20]. In CGKA, a sequence of secrets that can be used for messaging are established over time by a dynamic group of users. CGKA captures the core of group messaging, and TreeKEM, which can be cast as a CGKA [ACDT20, ACDT21a], forms the basis of the recently standardised MLS protocol and largely determines its performance. In addition, MLS is expected to be increasingly adopted in the future [Hog23, Gie23], and MLS and CGKA have received much academic attention in recent years [ACDT20, ACJM20, KPPW$^+$21, ACDT21a, HKP$^+$21, AAN$^+$22b, AJM22, BDG$^+$22, AHKM22, AMT23]. We therefore consider group administration through the lens of CGKA. We observe that neither previous CGKA protocols, nor existing work on group messaging in general, captured group administration at the cryptographic level. The closest we are aware of here is the work of Rösler et al. [RMS18] which considers group administration as a security requirement but does not adopt the provable security methodology. Consequently, in CGKA protocols in the literature, this renders all group members formally as group administrators with the same rights.

In this chapter, we introduce *administrated CGKA*, or A-CGKA, which natively captures administration and supports the adding, removing and updating of administrators (the latter being for forward security and post-compromise security). Namely, we extend the (unauthenticated) CGKA security notion of Alwen et al. [ACDT20] to additionally ensure that only admins can change the group membership and update admin secrets, even if several non-admin users are compromised, with the exception that we allow non-admins to update their own keys and remove themselves. We provide two constructions with different efficiency and security characteristics that build on top of a CGKA. Our first construction, Individual Admin Signatures (IAS), associates each admin with an evolving signature key pair and requires messages for admin-exclusive operations to be accompanied by admin signatures. Our second construction, Dynamic Group Signature (DGS), associates the set of admins with a secondary CGKA, or *admin* CGKA, following the observation that administration need not be exercised by individually authenticated administrators. In particular, the admin CGKA is used to manage the set of admins, where admins use the dynamic CGKA secret to derive common signature key-pairs that attest to admin operations. The composition of CGKAs employed in DGS may be of independent interest more generally for managing hierarchies in large-scale messaging. We then formally prove our two protocols secure. Moving from CGKA to group messaging, we detail how to integrate IAS-like administration into MLS by leveraging the existing credential infrastructure in MLS. By extending an existing implementation of MLS, we benchmark the

[5] We are not aware of any information indicating that the attack has been fixed in WhatsApp [RMS18]. Signal, for which state exposure was required to mount the attack, has since migrated to a newer private group management system [CPZ20] that is immune to it.

resulting protocol and demonstrate that it (as expected) incurs minimal overhead over MLS without administration. We finally discuss several directions for future work in both theory and practice for group administration.

WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs

In Chapter 6, the final main chapter of this manuscript, we focus on the group messaging protocol *Sender Keys* [Mar14] used by applications like WhatsApp and Signal. The core of Sender Keys is conceptually simple and works as follows. Each group member is associated with a *sender key*, which consists of a symmetric chain key ck and a signature public/secret key pair $(\mathsf{spk}, \mathsf{ssk})$. All group members learn each ck and spk corresponding to each group member via two-party channels (maintained by each pair of group members). To send an application message, a user hashes ck to derive a pair $(\mathsf{ck}', \mathsf{mk})$, encrypts using mk, signs the ciphertext and sends it alongside the signature. For forward security, they also delete mk and replace ck with ck'. Fresh sender keys are shared over two-party channels, and in particular all sender keys are refreshed whenever a group member is removed from a group.

Despite its practical importance, the protocol had not previously been formally modelled or described in depth. Rösler et al. [RMS18] previously analysed WhatsApp's implementation of Sender Keys which revealed flaws like the aforementioned burgle into the group attack, but did not provide a full protocol description or general characterisation of its security. While being more complex, MLS and CGKA protocols have seen more academic attention despite not yet being widely deployed. Sender Keys nonetheless remains as a practical alternative to MLS: WhatsApp is able to support billions of daily users and supports up to 1024 users per group [Wha23]. In addition, while offering logarithmic-sized updates in "good" cases, sufficiently "bad" and concurrent executions can lead to $O(n)$-sized update messages, which is inherent to CGKA at least when using off-the-shelf primitives [BDG+22].

Towards formalising Sender Keys, we first propose a primitive syntax and security model suitable for the protocol. Our security model parametrises the security of the underlying two-party channels, which allows us to observe that Sender Keys offers weaker than expected post-compromise security guarantees. In particular, key refreshes after compromise may not be secure due to the channels not having fully recovered, which can limit security in practice if the channels remain stale. We provide a full protocol description based on our understanding of Sender Keys mainly from WhatsApp's white paper [Wha20] and Signal's source code [M+16]. Our protocol also captures key updates without group changes that are implemented by Signal. We then prove security in our model, in which we have to notably restrict the adversary due to inherent limitations in Sender Keys. In this, we identify its deficiencies, including insecure group membership (affecting WhatsApp and to a lesser extent Signal in practice) and a lack of forward-secure authentication.

To remedy these security issues, we then propose and prove secure a new protocol that we call *Sender Keys+* that improves security without significantly affecting performance, and in

particular improves both the efficiency and security of key updates. By comparison to our baseline Sender Keys protocol, Sender Keys+ now uses signatures to attest to group changes and adds a message authentication code (or replaces encryption with an AEAD [Rog02]) in sending and receiving. For updates, first observe that fully recovering from compromise requires all sender keys to be refreshed, which incurs $O(n^2)$ communication for a group with n members, since each user needs to send a (constant-sized) sender key to all $O(n)$ users over two-party channels. We observe that, given sufficient synchronisation of parties, that to update, a user can sample a secret that is then hashed into each chain key, i.e., can update *all* sender keys at once, which restores confidentiality (assuming restored two-party channels) in just $O(n)$ communication (which is optimal). We also provide a theoretical comparison between Sender Keys(+) and CGKA-based protocols, from which we can conclude that Sender Keys, and especially Sender Keys+, may be sufficiently secure and performant for many practical deployments.

1.3 List of Works

The following works were produced while the author of this manuscript was working at EPFL. Works in **bold** appear in the body of this thesis. Full versions or preprints are also cited where relevant.

1. **Daniel Collins, Loïs Huguenin-Dumittan, Ngoc Khanh Nguyen, Nicolas Rolin, and Serge Vaudenay. K-Waay: Fast and deniable post-quantum X3DH without ring signatures. *USENIX Security 2024.*** [CHDN+24a, CHDN+24b]
2. **David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with Sender Keys? Analysis, improvements and security proofs. *ASIACRYPT 2023.*** [BCG23b, BCG23a]
3. **Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. *CRYPTO 2023.*** [BCC+23a, BCC+23b]
4. Renas Bacho, Daniel Collins, Chen-Da Liu-Zhang, and Julian Loss. Network-agnostic security comes (almost) for free in DKG and MPC. *CRYPTO 2023.* [BCLZL23, BCLZL22]
5. **David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. *USENIX Security 2023.*** [BCV23, BCV22]
6. Subhadeep Banik, Daniel Collins, and Willi Meier. Near collision attack against Grain V1. *ACNS 2023.* [BCM23a, BCM23b]
7. Daniel Collins, Simone Colombo, and Loïs Huguenin-Dumittan. Real world deniability in messaging. *Cryptology ePrint Archive, Paper 2023/403.* [CCHD23][6]
8. Andrea Caforio, Daniel Collins, Subhadeep Banik, and Francesco Regazzoni. A small GIFT-COFB: Lightweight bit-serial architectures. *AFRICACRYPT 2022.* [CCBR22a, CCBR22b]
9. Hailun Yan, Serge Vaudenay, Daniel Collins, and Andrea Caforio. Optimal symmetric ratcheting for secure communication. *The Computer Journal, 2021.* [YVCC23]
10. Andrea Caforio, Daniel Collins, Ognjen Glamočanin, and Subhadeep Banik. Improving first-order threshold implementations of SKINNY. *INDOCRYPT 2021.* [CCGB21a, CCGB21b]
11. Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. *DSN 2020* [CGK+20a, CGK+20b]

[6] This extended abstract was accompanied by a presentation at the IACR conference RWC 2023.

2 Preliminaries

In this chapter, we present some preliminary material that is used in subsequent chapters in this thesis. We first introduce some choices we make for notation and conventions, particularly describing how we define security. We then introduce some cryptographic primitives that either appear in several chapters or are otherwise suited to be defined here.

2.1 Notation and Conventions

We denote the empty string by ε and a blank value by $\perp$, the latter which we return in case of algorithm failure. Given a set S, S^* (respectively S^n) is the set of all strings of arbitrary length (resp. of length n) whose elements are in S. For two sets S and T, let $S \xleftarrow{\cup} T$ denote the reassignment of S to the set $S \cup T$, and let $S \xleftarrow{-} T$ denote the reassignment of S to the set $S \setminus T$. Let $[n] = \{1, \ldots, n\}$, i.e., the set of integers between 1 and n. Let $[a, b] = \{a, \ldots, b\}$, i.e., the set of integers between a and b. We consider randomised algorithms throughout the thesis: we let $\mathscr{R}$ denote a randomness space. We write $s \xleftarrow{\$} S$ to denote uniformly sampling s from S, and $y \xleftarrow{\$} f(x)$ to denote running randomised algorithm f with input x and uniform randomness. Alternatively, we write $y \leftarrow f(x; r)$ to denote running f with fixed randomness r. Let 1^λ be the security parameter (here stated in unary), and $\mathsf{poly}(\lambda)$ denotes a polynomial in λ. We denote the indicator function with respect to boolean P by $\mathbb{1}[\![P]\!]$, which is equal 1 if P is true and 0 otherwise.

Alice and Bob. A *user*, *participant*, or *party* is an entity that takes part in a protocol. In Part I of this thesis (Chapters 3 and 4), we consider primitives that are executed between two parties. As is common, we refer these two parties as Alice and Bob, or A and B. Let P be one party (A resp. B) and $\overline{\mathsf{P}}$ be their partner (B resp. A).

In Part II of this thesis (Chapters 5 and 6), we will consider primitives that deal with more than two parties. Here, users are identified by a unique identity string *ID*, which is a public value. Groups are also uniquely identified by a public group identifier gid. We assume these are known by and agreed upon by all parties for simplicity. The core primitives of study in

these chapters will be stateful, thus users keep an internal *state* γ with all information used for protocol execution. This may include keys, message records, dictionaries and parameters, among other values.

Security. To capture the security of cryptographic primitives and protocols, we work within the framework of *provable security*. In provable security, a security notion is defined as an experiment played between an adversary, often $\mathscr{A}$, and a challenger, say C, in which $\mathscr{A}$ is tasked to perform some task, such as outputting a forged signature or more generally breaking some cryptographic primitive in some experiment-dependent sense. The *advantage* of $\mathscr{A}$ in the experiment will be stated as a probability that is taken over the randomness of the challenger and adversary. In general, we say a scheme is *secure* if the advantage of any *efficient* adversary is *small* (we define efficient and small below). Such a claim is proven by reduction to hardness assumptions (e.g., the hardness of factoring integers) and/or the security of some cryptographic building blocks (e.g., SUF-CMA security of a signature scheme).

We are interested in both classical and quantum security in this thesis, i.e., security against a classical or quantum adversary. We denote by an *efficient* adversary a probabilistic polynomial-time (PPT) algorithm or quantum polynomial-time (QPT) algorithm, unless explicitly stated otherwise, where we mean polynomial in the security parameter λ. Our theorem statements will therefore hold insofar as the underlying assumptions hold and/or building blocks are secure with respect to a classical or quantum adversary. We consider an advantage to be *small* if it is *negligible* in λ. A function $f : \mathbb{N} \to \mathbb{R}$ is *negligible* if for every positive c, there exists integer N_c such that for all $x > N_c$, $|f(x)| < 1/x^c$ (for our purposes f outputs a probability). We let negl be a negligible function (in λ, which we omit writing and is implicit). We nonetheless state our security bounds exactly when proving security, i.e., the advantage of the scheme being proven secure will be upper bounded by some explicitly stated function of advantages of other primitives/assumptions and other terms.

Maps. We use maps, or associative arrays, which associate keys with values: $m[\cdot] \leftarrow x$ defines a new map with values initially set to x and $m[k]$ returns the element indexed by key k. Keys are tuples of any length $n \geq 1$. We index maps with integers starting from 1; in this case, $m[a : b]$ returns the list of elements whose keys are between a and b. We access the element of a tuple using a dot notation, e.g., $\gamma.k$ denotes the value k in the tuple γ. The function $\mathsf{length}(m)$ returns the number of keys in map m. All dictionaries can optionally be indexed by an oracle query q to represent the state of a dictionary at the time q is made, e.g., $\mathsf{E}[ID; q]$ denotes the value of $\mathsf{E}[ID]$ at the beginning of query q.

Keywords. In oracles and algorithms, some special predicates are used. The predicate '**require** P' enforces that a logical condition P is satisfied; otherwise the oracle/algorithm finishes immediately and returns $\perp$. The predicate '**reward** P' is executed in games and is such that if P holds, the adversary satisfies a winning condition in a game. Namely, in games where the advantage is defined by the probability that the adversary outputs 1, a game variable win is set to 1, and in indistinguishability games the game reveals the bit $b \in \{0, 1\}$ to the adversary.

The keyword '**public** var' indicates that the adversary has read access to the variable var. The prefix operator $\mathsf{Alg}(++x)$, is equivalent to writing first $x \leftarrow x+1$ and then $\mathsf{Alg}(x)$. Algorithms, oracle names and cryptographic parameters are denoted in sans-serif font.

(Quantum) Random Oracle Model. Some of our results will hold in the *random oracle model* (ROM) [BR93] or the *quantum* random oracle model (QROM) [BDF+11]. A random oracle is a uniformly random function, say H, associated with some input and output space. In the ROM or QROM, the adversary's queries are visible to whichever entity (the challenger or some other adversary, for example) is controlling the random oracle. In the QROM, the adversary can query the random oracle in superposition. That is, the adversary provides a quantum state as input, say $|\phi\rangle = \sum \alpha_x |x\rangle$, and the quantum random oracle returns the evaluated state, in this case $\sum \alpha_x |H(x)\rangle$.

2.2 Cryptographic Primitives

We introduce some core cryptographic primitives, including their syntax as well as correctness and security notions where appropriate, which we invoke throughout this thesis as needed.

2.2.1 Key-Encapsulation Mechanism (KEM)

Definition 1 (KEM)**.** A KEM KEM is a tuple of three efficient algorithms $(\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps})$ defined as follows:

- $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$: The key generation function takes the security parameter λ as input, and outputs a pair of public/secret keys $(\mathsf{pk}, \mathsf{sk})$.
- $K, \mathsf{ct} \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk})$: The encapsulation function takes a public key pk as input, and outputs a ciphertext ct and a key K.
- $K/\perp \leftarrow \mathsf{Decaps}(\mathsf{sk}, \mathsf{ct})$: The decapsulation function takes a secret key sk and a ciphertext ct as inputs, and outputs a key K or the error symbol $\perp$.

Finally, we say a KEM is $(1-\delta)$-correct if

$$\Pr\left[K \neq K' : \begin{array}{c} (\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda); \\ K, \mathsf{ct} \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}); \\ K' \leftarrow \mathsf{Decaps}(\mathsf{sk}, \mathsf{ct}) \end{array}\right] \leq \delta\,.$$

Definition 2 (KEM Indistinguishability)**.** We consider the games defined in Figure 2.1. Let $\mathscr{K}$ be a finite key space. A KEM scheme over $\mathscr{K}$ $\mathsf{KEM} = (\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps})$ is *IND-CPA* (resp. *IND-CCA*) if for any efficient adversary $\mathscr{A}$ with no access to the decapsulation oracle

(respectively any efficient adversary $\mathscr{A}$) we have

$$\mathsf{Adv}^{\text{ind-cpa/cca}}_{\mathsf{KEM}}(\mathscr{A}) := \left| \Pr[\mathsf{IND\text{-}CPA/CCA}_{\mathsf{KEM}}(\mathscr{A}) \Rightarrow 1] - \frac{1}{2} \right| = \mathsf{negl}\,.$$

where $\Pr[\mathsf{IND\text{-}CPA/CCA}_{\mathsf{KEM}}(\mathscr{A}) \Rightarrow 1]$ is the probability that $\mathscr{A}$ wins the $\mathsf{IND\text{-}CPA/CCA}_{\mathsf{KEM}}(\mathscr{A})$ game defined in Figure 2.1.

Game $\mathsf{IND\text{-}CPA/CCA}_{\mathsf{KEM}}(\mathscr{A})$	Oracle $\mathsf{DEC}(\mathsf{ct})$
1 : $b \xleftarrow{\$} \{0,1\}$	1 : **if** $\mathsf{ct} = \mathsf{ct}^*$: **return** $\perp$
2 : $(\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$	2 : $K' \leftarrow \mathsf{Decaps}(\mathsf{sk},\mathsf{ct})$
3 : $(\mathsf{ct}^*, K_0) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk})$	3 : **return** K'
4 : $K_1 \xleftarrow{\$} \mathcal{K}$	
5 : $b' \xleftarrow{\$} \mathscr{A}^{\mathsf{DEC}}(\mathsf{pk}, \mathsf{ct}^*, K_b)$	
6 : **return** $\mathbb{1}[\![b' = b]\!]$	

Figure 2.1: Indistinguishability games for KEM. In the IND-CPA game, the adversary cannot query DEC.

2.2.2 Signatures

Definition 3. A signature scheme is a tuple of three efficient algorithms (KeyGen,Sign,Vrfy):

- $(\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$: The key generation function outputs a pair of keys.
- $\sigma \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}, m)$: The signing function takes as inputs a secret key sk and the message to sign m, and it outputs a signature σ.
- $0/1 \leftarrow \mathsf{Vrfy}(\mathsf{pk}, m, \sigma)$: The verification function takes as inputs a public key pk, the signed message m, and the signature σ, and it outputs either 0 or 1 (for failure and success, respectively).

Finally, we say a signature scheme is $(1-\delta)$-correct if for all messages m:

$$\Pr\left[\mathsf{Vrfy}(\mathsf{pk}, m, \sigma) = 0 : \begin{array}{c}(\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda);\\ \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)\end{array}\right] \leq \delta$$

Definition 4 (SUF-CMA Security)**.** We consider the game shown in Figure 2.2. We say a signature scheme Sig is *SUF-CMA* if for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\text{suf-cma}}_{\mathsf{Sig}}(\mathscr{A}) := \Pr[\mathsf{SUF\text{-}CMA}_{\mathsf{Sig}}(\mathscr{A}) \Rightarrow 1] = \mathsf{negl}\,.$$

Game $\mathsf{SUF\text{-}CMA}_{\mathsf{Sig}}(\mathscr{A})$	Oracle $\mathsf{SIGN}(m)$
1: $L \leftarrow \emptyset$	1: $\sigma \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}, m)$
2: $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$	2: $L \leftarrow L \cup \{(m, \sigma)\}$
3: $m^*, \sigma^* \xleftarrow{\$} \mathscr{A}^{\mathsf{SIGN}}(\mathsf{pk})$	3: **return** σ
4: **return** $\mathbb{1}[\![\mathsf{Vrfy}(\mathsf{pk}, m^*, \sigma^*)$ **and** $(m^*, \sigma^*) \notin L]\!]$	

Figure 2.2: SUF-CMA game.

2.2.3 Pseudorandom Function (PRF)

We introduce security notions for a pseudorandom function (PRF) and two generalisations, namely dual PRF and triple PRF. For these, we consider functions of the form $F : \mathscr{K}^\ell \times D \to R$, i.e., that map $\ell \geq 1$ keys in $\mathscr{K}$ and a value in a domain D to a value in a range R. For such a function, we are interested in the security of F as a PRF when the i-th argument acts as the PRF key for all $i \in [1, \ell]$. Thus, we will consider the ℓ inputs to the function from the perspective of the caller as the $\ell - 1$ other values in $\mathscr{K}$ and the D inputs, and use the notation F_i as shorthand to indicate this. In fact, our notions match Giacon et al.'s formulation of a *split-key PRF* [GHP18].

Definition 5 (PRF)**.** Let $F : \mathscr{K} \times D \to R$ be a function. We consider the game shown in Figure 2.3. We say that F is a secure *PRF* if, for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\mathrm{prf}}_F(\mathscr{A}) := \left| \Pr\left[\mathsf{PRF}_F(\mathscr{A}) \Rightarrow 1\right] - \frac{1}{2} \right| = \mathsf{negl}\,.$$

Definition 6 (Dual PRF)**.** Let $F : \mathscr{K} \times \mathscr{K} \times D \to R$ be a function. We consider the game shown in Figure 2.3. We say that F is a secure *dual PRF* or *2PRF* if, for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\mathrm{2prf}}_F(\mathscr{A}) := \max_{i \in \{1,2\}} \left| \Pr\left[\mathsf{PRF}_{F_i}(\mathscr{A}) \Rightarrow 1\right] - \frac{1}{2} \right| = \mathsf{negl}\,.$$

where F_i denotes F keyed in its i-th argument for $i \in \{1,2\}$.

Definition 7 (Triple PRF)**.** Let $F : \mathscr{K} \times \mathscr{K} \times \mathscr{K} \times D \to R$ be a function. We consider the game shown in Figure 2.3. We say that F is a secure *triple PRF* or *3PRF* if, for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\mathrm{3prf}}_F(\mathscr{A}) := \max_{i \in \{1,2,3\}} \left| \Pr\left[\mathsf{PRF}_{F_i}(\mathscr{A}) \Rightarrow 1\right] - \frac{1}{2} \right| = \mathsf{negl}\,.$$

where F_i denotes F keyed in its i-th argument for $i \in \{1,2,3\}$.

Observe the difference between our notion of dual PRF and the dual PRF introduced by Bellare [Bel06], which is a function $F : K_1 \times K_2 \to R$ and PRF when keyed from either K_1 or K_2 (the latter referred to as the *swap PRF* case).

Game $\mathsf{PRF}_F(\mathscr{A})$	Oracle $\mathsf{EVAL}(a_1,\ldots,a_\ell)$
1 : Sample random function $G : \mathcal{K}^{\ell-1} \times D \to R$	1 : **require** $a_\ell \notin X$
2 : $k \xleftarrow{\$} \mathcal{K}$	2 : $X \leftarrow X \cup \{a_\ell\}$
3 : $b \xleftarrow{\$} \{0,1\}$	3 : **if** $b = 0$
4 : $b' \xleftarrow{\$} \mathscr{A}^{\mathsf{EVAL}}$	4 : **return** $F_k(a_1,\ldots,a_\ell)$
5 : $X \leftarrow \emptyset$	5 : **else**
6 : **return** $\mathbb{1}[\![b' = b]\!]$	6 : **return** $G(a_1,\ldots,a_\ell)$

Figure 2.3: PRF game for function F_k taking ℓ arguments $a_1,\ldots,a_\ell$ as input where $a_1 \in \mathcal{K},\ldots,a_{\ell-1} \in \mathcal{K}$ and $a_\ell \in D$.

2.2.4 Hash Function

Definition 8. A hash function H consists of efficient algorithms KGen and Eval such that:

- $\mathsf{hk} \xleftarrow{\$} \mathsf{KGen}(1^\lambda)$: The key generation function outputs a hash key hk.
- $h \leftarrow \mathsf{Eval}(\mathsf{hk}, \mathsf{pt})$: The evaluation function takes as inputs a hash key hk and a message $\mathsf{pt} \in \{0,1\}^*$ and outputs digest h.

Definition 9 (Collision Resistance)**.** We consider the game in Figure 2.4. We say that a hash function H is collision resistant if, for all efficient adversaries, we have

$$\mathsf{Adv}^{\mathrm{cr}}_{\mathsf{H}}(\mathscr{A}) := \Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{A}) \Rightarrow 1] = \mathsf{negl}\,.$$

Game $\mathsf{CR}_{\mathsf{H}}(\mathscr{A})$
1 : $\mathsf{hk} \leftarrow \mathsf{KGen}(1^\lambda)$
2 : $(m_1, m_2) \leftarrow \mathscr{A}(\mathsf{hk})$
3 : **return** $\mathbb{1}[\![\mathsf{H.Eval}(\mathsf{hk}, m_1) = \mathsf{H.Eval}(\mathsf{hk}, m_2) \wedge m_1 \neq m_2]\!]$

Figure 2.4: Collision resistance of a hash function H.

2.2.5 Incremental Set Hash Function

Clarke et al. [CDv+03] define incremental *multiset* hash functions and multiset collision resistance. That is, a hash function which takes a set of elements as input, where the digest can be updated with an operation with complexity proportional to the number of elements added/removed. For our purposes (i.e., in Chapter 4), it suffices to consider an *incremental set hash function* and *set collision resistance*.

Definition 10. An incremental set hash function H consists of the following efficient algorithms:

- $\mathsf{hk} \xleftarrow{\$} \mathsf{IncGen}(1^\lambda)$: The key generation function outputs a hash key hk.
- $h_S \leftarrow \mathsf{IncEval}(\mathsf{hk}, S = \{m_1, \ldots, m_k\})$: The evaluation function for a set of messages takes a hash key hk and set S and outputs digest h_S.
- $h_{S \cup S_h} \leftarrow \mathsf{IncEval}(\mathsf{hk}, h, S_h = \{m'_1, \ldots, m'_{k'}\}, S = \{m_1, \ldots, m_k\})$: The evaluation function for a set of messages and digest takes as input a hash key hk, digest h, set S_h (associated with h) and set S and outputs a digest $h_{S_h \cup S}$.

An incremental hash function is *correct* if for all $\mathsf{hk} \xleftarrow{\$} \mathsf{IncGen}(1^\lambda)$ and non-empty $S \leftarrow \{m_1, \ldots, m_k\}$ it holds that

$$\mathsf{IncEval}(\mathsf{hk}, S) = h' ,$$

where $h' \leftarrow \mathsf{IncEval}(\mathsf{hk}, h, S_h, S')$, $S_h \cup S' = S$, $S_h \cap S' = \emptyset$ and h is the result of either 1) a call $\mathsf{IncEval}(\mathsf{hk}, S_h)$ or 2) a call $\mathsf{IncEval}(\mathsf{hk}, S_1)$ and one or more calls to $\mathsf{IncEval}(\mathsf{hk}, \cdot, S_2, S_3)$ for disjoint sets $S_1, S_2, S_3 \subset S_h$ s.t. each pair of sets S_2 and S_3 is strictly increasing in cardinality and the last call is s.t. $S_2 \cup S_3 = S_h$.

Definition 11 (Set Collision Resistance)**.** Consider the game defined in Figure 2.5. A family of incremental set hash function H is set collision resistant, if for any efficient adversary, we have

$$\mathsf{Adv}^{\mathsf{scr}}_{\mathsf{H}}(\mathscr{A}) := \Pr[\mathsf{SCR}_{\mathsf{H}}(\mathscr{A}) \Rightarrow 1] = \mathsf{negl} .$$

Game $\mathsf{SCR}_{\mathsf{H}}(\mathscr{A})$
1 : $\mathsf{hk} \xleftarrow{\$} \mathsf{H.IncGen}(1^\lambda)$
2 : $(S_1 = \{m_i\}_i, S_2 = \{m'_j\}_j) \xleftarrow{\$} \mathscr{A}(\mathsf{hk})$
3 : **return** $\mathbb{1}[\![\mathsf{H.IncEval}(\mathsf{hk}, S_1) = \mathsf{H.IncEval}(\mathsf{hk}, S_2) \wedge (S_1 \neq S_2)]\!]$

Figure 2.5: Set collision resistance of an incremental set hash function H.

We refer the reader to Clarke et al. [CDv+03] for more details about incremental set hash functions.

2.2.6 Symmetric Encryption

Definition 12 (Symmetric Encryption)**.** A symmetric encryption scheme $\mathsf{SymEnc} := (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is a tuple of efficient algorithms such that:

- $\mathsf{k} \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$: Given the security parameter 1^λ (encoded in unary) the generation algorithm returns a key $\mathsf{k} \in \mathscr{K}$.
- $c \xleftarrow{\$} \mathsf{Enc}(\mathsf{k}, m)$: Given a key k and a message m, the encryption algorithm returns a ciphertext c.

- $m \leftarrow \mathsf{Dec}(\mathsf{k}, c)$: Given a key k and a ciphertext c, the decryption algorithm returns a message m.

For simplicity of exposition, we model Enc as probabilistic rather than explicitly modelling input initialisation vectors (we therefore implicitly assume they are sampled randomly and appended to the ciphertext where relevant). We say that SymEnc is correct if for any message $m \in \mathscr{M}$ and any key $\mathsf{k} \in \mathscr{K}$ it holds that,

$$\Pr\left[\mathsf{Dec}(\mathsf{k}, c) = m \;\middle|\; \begin{array}{ll} \mathsf{k} & \xleftarrow{\$} \mathsf{Gen}(1^\lambda) \\ c & \xleftarrow{\$} \mathsf{Enc}(\mathsf{k}, m) \end{array}\right] = 1,$$

where the probability is taken over the random coins of Enc, and $\mathscr{M}$ and $\mathscr{K}$ denote the message space and key space respectively.

Definition 13 (SymEnc IND-CPA Security)**.** We consider the game in Figure 2.6. We say that a symmetric encryption scheme SymEnc is secure under chosen plaintext attacks, or is IND-CPA, if, for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathsf{SymEnc}}(\mathscr{A}) := \left|\Pr[\mathsf{IND\text{-}CPA}_{\mathsf{SymEnc},1}(\mathscr{A}) \Rightarrow 1] - \Pr[\mathsf{IND\text{-}CPA}_{\mathsf{SymEnc},0}(\mathscr{A}) \Rightarrow 1]\right| = \mathsf{negl}.$$

Game $\mathsf{IND\text{-}CPA}_{\mathsf{SymEnc},b}(\mathscr{A})$	Oracle $\mathsf{ENC}(m)$
1: $\mathsf{k} \xleftarrow{\$} \mathsf{SymEnc.Gen}(1^\lambda)$	1: $c \xleftarrow{\$} \mathsf{Enc}(k, m)$
2: $m_0, m_1 \leftarrow \perp$	2: **return** c
3: $(m_0, m_1, \mathsf{st}) \leftarrow \mathscr{A}^{\mathsf{ENC}}$	
4: **require** $\lvert m_0\rvert = \lvert m_1\rvert$	
5: $c^* \xleftarrow{\$} \mathsf{Enc}(k, m_b)$	
6: $b' \leftarrow \mathscr{A}(c^*, \mathsf{st})$	
7: **return** b'	

Figure 2.6: IND-CPA security for symmetric encryption scheme SymEnc.

Definition 14 (PRG Security of H)**.** We consider the game in Figure 2.7. Let H be a function $\mathsf{H} : \mathscr{S} \to \mathscr{W} \times \mathscr{K}$. We say that H is a secure *PRG* if, for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\mathrm{prg}}_{\mathsf{H}}(\mathscr{A}) := \left|\Pr[\mathsf{PRG}_{\mathsf{H}}(\mathscr{A}) \Rightarrow 1] - \frac{1}{2}\right| = \mathsf{negl}\,.$$

Definition 15 (Message Authentication Code)**.** A message authentication code (MAC.Gen,MAC.Tag,MAC.Vrfy) is a tuple of efficient algorithms as follows:

- $\mathsf{k} \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$: The generation algorithm takes as input the security parameter 1^λ (encoded in unary) and outputs a key $\mathsf{k} \in \mathscr{K}$.

Game $\mathsf{PRG}_{\mathsf{H}}(\mathscr{A})$	Oracle ROR
1 : $b \xleftarrow{\$} \{0,1\}$	1 : **if** $b = 0$:
2 : $b' \leftarrow \mathscr{A}^{\mathsf{ROR}}$	2 : $(w,k) \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^\lambda$
3 : **return** $\mathbb{1}[\![b = b']\!]$	3 : **if** $b = 1$:
	4 : $s \xleftarrow{\$} \{0,1\}^\lambda$
	5 : $(w,k) \leftarrow \mathsf{H}(s) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$
	6 : **return** (w,k)

Figure 2.7: Pseudorandom generator (PRG) security for H.

- $\tau \xleftarrow{\$} \mathsf{Tag}(\mathsf{k}, m)$: The tag generation takes as inputs a key k and a message m, and outputs a tag τ.
- $b \leftarrow \mathsf{Vrfy}(\mathsf{k}, m, \tau)$: The verification algorithm takes as inputs a key k, a message m and a tag τ and outputs a bit $b \in \{0,1\}$.

We say that MAC is correct if for any message $m \in \mathscr{M}$ and any key $\mathsf{k} \in \mathscr{K}$ it holds that

$$\Pr\left[\mathsf{Vrfy}(\mathsf{k}, m, \tau) = 1 \;\middle|\; \begin{array}{ll} \mathsf{k} & \xleftarrow{\$} \mathsf{Gen}(1^\lambda) \\ \tau & \xleftarrow{\$} \mathsf{Tag}(\mathsf{k}, m) \end{array}\right] = 1,$$

where the probability is taken over the random coins of Tag, and $\mathscr{M}$ and $\mathscr{K}$ denote the message space and key space respectively.

Definition 16 (MAC SUF-CMA Security)**.** We consider the game in Figure 2.8. Let MAC be a message authentication scheme. We say that MAC satisfies strong existential unforgeability if, for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\mathsf{suf\text{-}cma}}_{\mathsf{MAC}}(\mathscr{A}) := \Pr[\mathsf{SUF\text{-}CMA}_{\mathsf{MAC}}(\mathscr{A}) \Rightarrow 1] = \mathsf{negl}\,.$$

Game $\mathsf{SUF\text{-}CMA}_{\mathsf{MAC}}(\mathscr{A})$	Oracle $\mathsf{TAG}(m)$
1 : $k \xleftarrow{\$} \mathsf{MAC.Gen}(1^\lambda)$	1 : $\tau \xleftarrow{\$} \mathsf{MAC.Tag}(k, m)$
2 : $Q \leftarrow \emptyset$	2 : $Q \leftarrow Q \cup \{m, \tau\}$
3 : $(m,t) \leftarrow \mathscr{A}^{\mathsf{TAG}}$	3 : **return** τ
4 : **return** $\mathbb{1}[\![(m,t) \notin Q \wedge \mathsf{MAC.Vrfy}(k,m,t) = 1]\!]$	

Figure 2.8: SUF-CMA security for message authentication code MAC.

Two-Party Communication Part I

3 K-Waay: Fast and Deniable Post-Quantum X3DH Without Ring Signatures

In this chapter, we propose a deniable, post-quantum authenticated key exchange protocol that we call K-Waay that can be used in place of Signal's X3DH protocol [MP16b]. An extended abstract corresponding to this work appeared at USENIX Security 2024, and was joint work with Loïs Huguenin-Dumittan, Ngoc Khanh Nguyen, Nicolas Rolin and Serge Vaudenay [CHDN+24a].

The author of this thesis was the primary contributor to the following parts of this work: the deniable authenticated key exchange primitive, key indistinguishability security model and definition of deniability, protocol description and protocol security proofs. The author further contributed to several other parts of this work in various capacities. Moreover, this work was the result of many hours of discussion and collaboration, and so it is difficult to attribute precisely here. The contents of the full version of this paper is nonetheless included in this thesis for comprehensibility and completeness.

3.1 Contribution

Researchers for several years now have sought to build cryptographic primitives and protocols that are resistant to efficient quantum attacks [Sho94]. This is highly evidenced with the NIST Post-Quantum Cryptography competition for standardising quantum-safe key encapsulation mechanisms (KEM) and signatures, organised by the United States National Institute of Standards and Technology (NIST). Recently, four schemes were selected by NIST for standardisation, out of which three rely on algebraic lattices. Indeed, with the US National Security Agency releasing their new CNSA 2.0 Suite [US], which says that CRYSTALS-Kyber [BDK+18] and CRYSTALS-Dilithium [DKL+18] should be the main cryptographic force for communication security beginning from 2030, lattices are a natural candidate for building more advanced cryptographic primitives, such as secure messaging.

The widely used Signal protocol for secure messaging as currently deployed is not quantum-safe since it is based on Diffie-Hellman key exchange [DH76]. The protocol, used in ap-

plications like Signal and WhatsApp, comprises two components, namely (1) the X3DH key exchange [MP16b] which is used to bootstrap sessions of (2) the Double Ratchet messaging protocol [PM16]. The Double Ratchet has been investigated in a line of recent works [ACD19, BFG+22a, CJSV22] that each neatly abstract the protocol into primitives like so-called *continuous key agreement*. Fortunately, these primitives have post-quantum (PQ) instantiations that leaves the core structure and resulting security guarantees of the Double Ratchet in place.

In standard X3DH, parties use a mixture of ephemeral (one-time), semi-static (many-time but temporary) and long-term keys. First, parties upload their keying material to a central server or public key infrastructure in a so-called prekey bundle. A party can then derive a session key by downloading their partner's bundle and performing three (or four) Diffie-Hellman key exchanges with a mixture of ephemeral and long-term (resp. plus semi-static) keys, ensuring at least confidentiality if the ephemeral *or* long-term key of each party is corrupted.

Observe that X3DH does not use signatures after signed prekeys are uploaded: at that point, the DH exchanges provide (implicit) authentication guarantees. Consequently, the protocol provides a level of *deniability* [DNS04, DGK06, UG15] as was formalised by Vatandas et al. [VGIK20]: informally, a participant can deny having performed key exchange with its counterpart. This is an important privacy guarantee that prevents (at least on a cryptographic level) a conversation transcript from incriminating an unsuspecting party, particularly in situations like whistleblowing and protesting.

In 2023, Signal announced and rolled out their initial *hybrid* post-quantum key exchange solution called PQXDH [KS23]. Like in X3DH, several Diffie-Hellman key exchanges are performed at once, but in PQXDH, parties upload prekey bundles that also contain a Kyber-1024 public key that the initiator additionally encapsulates to the responder with. Moreover, prekey bundles are still signed with the same signature scheme as regular X3DH based on Curve25519 [Ber06]. Although PQXDH provides post-quantum confidentiality [BJKS23a], which is an important first step towards post-quantum security as it prevents "store-now-decrypt-later" attacks, it does not provide post-quantum *authentication* as an active quantum attacker can trivially forge pre-key bundles. It is thus prudent to design a suitable X3DH alternative that is *fully* post-quantum secure.

A natural direction for building such a protocol is to emulate X3DH's structure by replacing Diffie-Hellman key exchange with a cryptographic group action, such as CSIDH [CLM+18]. In order to broadly capture this protocol structure, Brendel et al. [BFG+20] introduce a primitive called *split-KEM* that captures the symmetry of e.g. Diffie-Hellman. In a split-KEM, a party A encapsulates to their partner B by using their own secret sk_A and their partner's public key pk_B to produce a ciphertext; B then decapsulates it using sk_B and pk_A. The authors define indistinguishability-based security notions and notice that Frodo [BCD+16] lattice-based key-exchange fulfills the split-KEM syntax and the weakest notion of indistinguishability they

define.[1] Although they present an X3DH-like protocol, they do not define a security model, and, looking ahead, their split-KEM security notions do not suffice to construct X3DH-like key exchange with authenticity and deniability.

In two recent works, Hashimoto et al. [HKKP21, HKKP22] and Brendel et al. [BFG+22b] concurrently proposed instead to construct X3DH-like key exchange using KEMs directly. Since a core feature of X3DH is its *asynchronicity*, a challenge-response protocol cannot be employed using KEMs alone to provide authentication [SSW20]. Thus to ensure deniability, two seemingly different approaches were proposed: Hashimoto et al. [HKKP21] apply ring signatures while Brendel et al. [BFG+22b] use a flavour of designated verifier signatures; these primitives were later shown to be equivalent [HKKP22].

As described in the aforementioned works, the currently most efficient post-quantum ring signatures [Beu20, ESZ22, LAZ19, LN22, YEL+21] are proven to be secure in the random oracle model (ROM) [BR93] and can enjoy signatures that are a handful of kilobytes large. Often, however, the constructions do not come with a security proof in the quantum random oracle model (QROM) [BDF+11]. In this vein, parameters are generally optimistically chosen as the security loss incurred by proofs in the ROM is not taken into account when setting them, without even mentioning QROM loss, which is usually much larger. Further, security notions can differ between works, making it less clear exactly when they are appropriate for use.

More generally, it is of interest to determine the cost (or overhead) that deniability incurs in (X3DH-like) key exchange. Towards this goal, Hashimoto et al. [HKKP22] provide benchmarks for their baseline, non-deniable X3DH-like protocol based on signatures and KEMs, and Brendel et al. [BFG+22b] consider parameter sizes for (but do not benchmark) existing ring and designated verifier signatures. As such, a more fine-grained and detailed evaluation will help inform practitioners on the overhead incurred by deniability in the post-quantum setting.

While the use of ring signatures to build PQ and deniable X3DH is at least theoretically understood, this far from exhausts the protocol design space. Motivated by this and the above discussion, we therefore ask the following research question:

- Can we design a provably-secure, efficient and deniable post-quantum X3DH alternative that does *not* require ring signatures?

3.1.1 Summary

In this chapter, we propose an efficient, deniable and post-quantum X3DH-like protocol without ring signatures that we call K-Waay. To summarise our contributions:

- Towards building our protocol, we revisit the split-KEM formalism proposed by Brendel

[1] The construction can conceptually be seen as instantiating the lattice-based cryptographic group action of Beullens [Beu20].

et al. [BFG+20] and deduce that several additional properties, namely notions of authenticity and deniability, are needed to construct a secure X3DH-like deniable authenticated key exchange protocol (DAKE) (Section 3.2).

- We propose K-Waay, an X3DH-like DAKE that uses deniable and unforgeable split-KEM at its core (Section 3.4). Our protocol uses signatures to sign prekeys, and then uses ephemeral KEM, long-term KEM and split-KEM for the final key exchange step. We compare the security of our protocol with the state-of-the-art in Table 3.1.

- The main drawback of a naive version of our protocol is that parties can run out of ephemeral keys, thus making the protocol synchronous if this happens (e.g. Bob needs to wait for Alice's fresh ephemeral key before sending a message). While such a problem would rarely occur in practice, given enough keys are uploaded on the server, we propose a simple trick that makes the reuse of ephemeral keys possible on the receiver's side for messages they received while offline. We think this trick could be of independent interest as it – perhaps surprisingly – allows for a specific kind of key reuse for a split-KEM that is *not* IND-CCA secure.

- We prove key indistinguishability in our model that captures ephemeral key reuse and session state exposure, and prove a variant of deniability that strengthens the notion of Brendel et al. [BFG+22b] by additionally leaking the victim's session state to the adversary in the security game.

- We instantiate a post-quantum split-KEM secure under our new security notions derived from the Frodo key exchange protocol (FrodoKEX) [BCD+16] based on the plain LWE assumption. The parameters we chose provide strong security guarantees, providing more than 192 bits of classical and quantum security for our core split-KEM security notions OW-CPA, decaps-CPA and deniability. We then use a transform in the (Q)ROM to prove it UNF-1KCA and IND-1BatchCCA (i.e., our new unforgeability and indistinguishability definitions for split-KEM). This construction incurs a security loss as usual in the (Q)ROM, but our final split-KEM still provides around 128 (resp. 64) bits of security in the ROM (resp. QROM) assuming the adversary is limited to 2^{64} (resp. quantum) random oracle queries.

- We benchmark our protocol K-Waay using our modified version of FrodoKEX (which we call FrodoKEX+) as the split-KEM, along with standard X3DH and the two previous proposals for PQ X3DH-like AKE [HKKP22, BFG+22b] (Section 3.6). We find that while K-Waay has larger prekeys, it is 6× faster compared to these. In addition, the only non-standard primitive we use in K-Waay (FrodoKEX+) is based on both an assumption (LWE) and a scheme (FrodoKEM) that have been thoroughly scrutinised by the cryptographic community. Overall, we believe our protocol is more mature and therefore suitable for short to medium-term integration compared to previous work based on ring signatures.

Protocol	PQ Conf	PQ Auth	KCI	FS	SSR	RR	Deniability
X3DH [MP16b, CCD+20]	✗	✗	✓	PFS	✓	✓	Malicious
PQXDH [KS23, BJKS23a]	✓	✗	✓	PFS	✗	✗	Semi-honest+
KEM+Sigs [HKKP22]	✓	✓	✓	PFS	✓	✗	✗
HKKP [HKKP22]	✓	✓	✓	WFS	✓	✗	Semi-honest
SPQR [BFG+22b]	✓	✓	✓	WFS	✗	✓	Semi-honest
K-Waay (Section 3.4)	✓	✓	✓	WFS	✓	✗	Semi-honest

Table 3.1: Comparison between different security properties proven for existing X3DH-like key exchange protocols, namely post-quantum confidentiality (PQ Conf), authentication (PQ Auth), resistance to key-compromise impersonation attacks when long-term keys are exposed (KCI), perfect forward secrecy (PFS) or weak forward secrecy (WFS) [Kra05], session state reveal (SSR), randomness reveal (RR) and deniability (where the judge/adversary is either honest-but-curious or is malicious and can inject messages). Protocols can be generically strengthened to handle randomness reveal by standard application of the so-called NAXOS trick [LLM07]. "KEM+Sigs" refers to the non-deniable baseline X3DH-like protocol proposed by Hashimoto et al. [HKKP21, HKKP22], and "HKKP" refers to their deniable X3DH protocol *without* NIZKs (their protcool with NIZKs implies maliciously-secure deniability w.r.t. a *classical* adversary). The security of PQXDH is based on the recent analysis of Bhargavan et al. [BJKS23a] except that Kret and Schmidt argue it also provides at least semi-honest deniability [KS23].

3.1.2 Technical Overview

X3DH-like Key Exchange. A quantum-secure X3DH-like protocol should satisfy certain properties. Apart from satisfying standard authenticated key exchange (AKE) properties like secrecy and authentication, it should also be *asynchronous*. That is, parties should be able to upload keying material to a central server, after which an initiating party can derive a session key immediately with their counterpart who may be offline. This also entails *receiver-obliviousness*, using the language of Hashimoto et al. [HKKP22], as the initial key upload should not depend on the keys of any other party. Another is *deniability*, allowing parties to claim that they plausibly did not participate in the key exchange. Note that we cannot possibly ensure that parties can claim that they never uploaded prekeys as they are signed (and using primitives like ring signatures would violate receiver-obliviousness). Finally, a deniable authenticated key exchange protocol, or *DAKE*, should, like X3DH, provide security guarantees even if the session state of a party is leaked.

Revisiting Split-KEM. In an attempt to model the primitive central to X3DH-like AKE, Brendel et al. [BFG+20] introduced *split-KEM*, which is similar to a standard KEM except the encapsulator can contribute to the derived key. However, we discovered that the accompanying security definitions were not sufficient to use such a primitive as the main component of a key exchange protocol. The reason is that their notions ensure that an encapsulated ciphertext will not leak information on its encapsulated key, but not that only the sender can send a "legitimate" ciphertext to the receiver (or that only the sender and receiver can derive a common key). In other words, there is no guarantee of implicit authentication. Therefore, we introduce

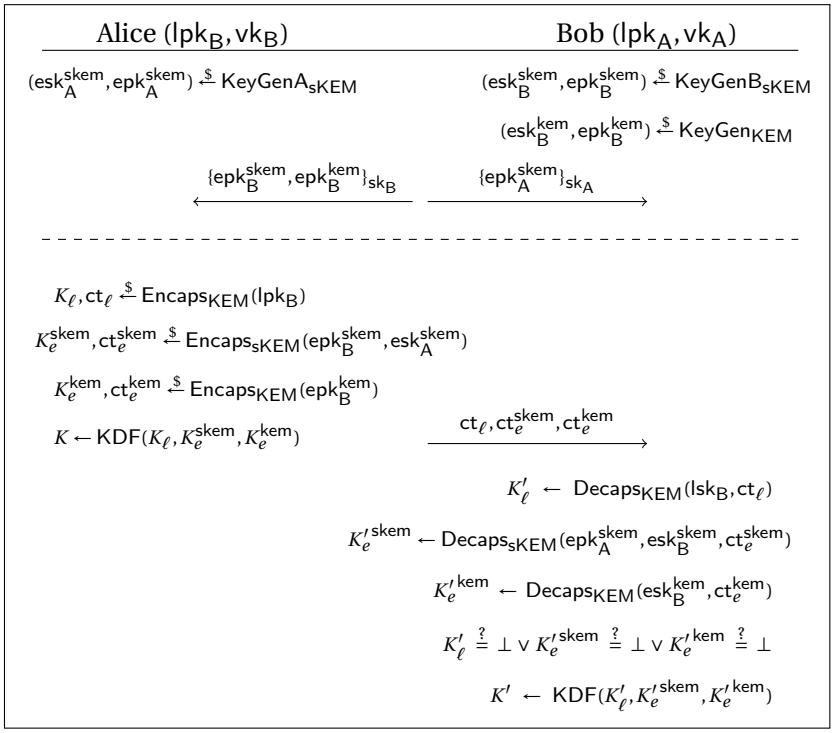

Figure 3.1: High-level overview of the K-Waay protocol. Values in brackets $\{\cdot\}_{\mathsf{sk}}$ are signed with sk and the signature is verified upon reception. For clarity, we omit the calculation and addition of session identifier sid to KDF.

the notion of unforgeability against one known-ciphertext attacks for split-KEM (UNF-1KCA), which ensures that if Alice receives a message allegedly sent by Bob, either Bob really sent it or the decapsulation will fail. Looking ahead, this will be used in the security proof of the protocol to argue that either the adversary relayed a legitimate split-KEM ciphertext to the receiver that the adversary cannot learn the decapsulation of, or the receiver aborts as the ciphertext is forged.

We also introduce an intermediary notion that we call decaps-CPA, which enforces that an adversary should not be able to recover a key *decapsulated* by some party without knowing the sender's or receiver's secret key. We will prove that our lattice-based split-KEM satisfies this notion, then we will apply some transform in the (Q)ROM to obtain a UNF-1KCA split-KEM.

Finally, we also define a notion capturing deniability for split-KEM, which states that no judge J can be convinced that a party A sent a given ciphertext to B, even knowing B's secret key but assuming both parties did not deviate from the protocol. This models a setting where B communicates with A and later tries to frame the latter by giving the transcript and their own secret key to J.

Construction. As any X3DH-like protocol, our construction works in 4 phases: long-term key generation, prekey generation, sending and receiving. The first observation we make is

that in X3DH, prekey bundles are signed with a long-term signing key before being uploaded to the server. This fact is often abstracted away in formal analyses as it hurts the claims one can make about the deniability of X3DH: as a signature is undeniable by definition, users cannot deny they *participated* in the protocol. Based on this, our goal was to achieve some level of peer-deniability [CF11], where parties can deny they communicated with someone in particular, and to leverage the fact that we use signatures to authenticate the prekeys. Our protocol works then as follows (see Figure 3.1 for a high-level overview). The long-term key pair consists of a KEM and signature key pair, the latter being used to sign the prekey, which consists of an ephemeral KEM key pair and ephemeral split-KEM key pair. The former is used for forward secrecy while the second is used for implicit authentication of the sender. Although usually ephemeral keys cannot be used for authentication as they are dynamic, in our case we can since they are authenticated (i.e., signed) by their owner. Then, the sender encapsulates against both KEM public keys of the receiver, and uses their own split-KEM secret key and the receiver's public key to derive a split-KEM ciphertext. Upon decapsulation, the receiver recovers the three encapsulated keys and combines them using a PRF to derive the shared key.

Ephemeral Split-KEM Key Reuse. The way our protocol is described above works perfectly well if the split-KEM satisfies the $\mathsf{UNF\text{-}1KCA}$ unforgeability notion mentioned above. However, in practice, it could happen that some party, say Bob, is offline for too long and all their ephemeral split-KEM keys have been used. If that occurs, another sender would have to wait for Bob to come online and upload new keys before they can send him a message.

We fix this issue by modifying the protocol as follows: when Bob's ephemeral public keys have run out on the server, a sender can simply reuse one of them. Then, when Bob is back online, he groups the ciphertexts corresponding to the same public key and decrypts all ciphertexts in a group *at once*. If one or more of the split-KEM decapsulations in a group fails, Bob outputs $\perp$ for *all* ciphertexts and, e.g., restarts the protocol. Otherwise, Bob proceeds as before (and *never* decapsulates again using the same split-KEM key).

DAKE Modelling. To formally capture ephemeral key reuse, our DAKE syntax includes an algorithm $\mathsf{BatchReceive}$ that takes as input a session state and one or more messages to be received. Our key indistinguishability notion therefore has to account for $\mathsf{BatchReceive}$, and so we extended the typical Bellare-Rogaway-style modelling [BR94] to this end (ignoring this, our model combines aspects of the models of Hashimoto et al. [HKKP21, HKKP22] and Brendel et al. [BFG+22b]). In particular, our notion of *partnering* between sending and receiving sessions is such that a given sending session can be partnered with several receiving sessions. Partnered sessions, in turn, are used when defining trivial attacks that our protocol (and often, but not always *any* protocol) cannot prevent (e.g., the adversary exposing both the session and long-term state of the receiver) as well as correctness checks. The adversary's challenge query, i.e., the query that returns to them either a real or random key, returns a single key, which in the case of $\mathsf{BatchReceive}$ corresponds to only part of its output.

Security. We show this version of the protocol is secure assuming the split-KEM satisfies a stronger notion than IND-CPA that we call IND-1BatchCCA (in addition to UNF-1KCA security). This definition is the same as traditional IND-CPA (adapted to the split-KEM syntax), except the adversary can query a decapsulation oracle *once* with multiple public keys and ciphertexts, and the oracle returns $\perp$ if *one or more* of the decapsulations failed, and the resulting keys otherwise. We show that one can easily build an IND-1BatchCCA split-KEM out of a CPA-secure one in the (Q)ROM, conveniently using the same transform mentioned above that builds a UNF-1KCA scheme out of a decaps-CPA one.

As in previous protocols [BFG+22b, HKKP22], the long-term KEM provides implicit authentication of the receiver as only they can decrypt. As mentioned above, the ephemeral KEM provides forward secrecy, and the UNF-1KCA/IND-1BatchCCA split-KEM provides implicit authentication of the sender, as it guarantees that only the sender could have sent a ciphertext that correctly decapsulates (unforgeability), and no adversary knows what is inside that ciphertext (indistinguishability), even after seeing the decapsulation of one batch of ciphertexts encapsulated against the same public key (given no decapsulation failed). We note that the sender-to-receiver authentication depends both on a long-term key (i.e., the signing key) and an ephemeral one (the split-KEM key). Consequently, our model (that allows session state exposure) is more restrictive than that of Hashimoto et al. [HKKP22], since in particular it suffices for the adversary to learn a receiver's ephemeral state during key exchange to forge a message that the receiver accepts. Intuitively, this is because split-KEM is effectively a symmetric primitive. Nevertheless, the security that we achieve is stronger than weak forward security without session state exposure.

Deniable Split-KEM from Lattices. We provide a lattice-based split-KEM which satisfies both deniability and UNF-1KCA security. Our starting point is the Frodo key-exchange (FrodoKEX) [BCD+16], which was identified (among other schemes) as a split-KEM by Brendel et al. [BFG+20], the security of which relies on the well-known learning-with-errors (LWE) problem [Reg05]. We highlight that the vanilla construction of FrodoKEX does not enjoy the aforementioned properties.[2] Indeed, when looking closely at the security games of deniability and UNF-1KCA, partial information about the secret keys are revealed - thus making a reduction to LWE completely non-trivial. We circumvent this problem in two ways.

First, we reduce deniability of our scheme to a so-called Extended-LWE problem [AP12], where in addition to a standard LWE instance, the adversary is given a short random combination of the secret coefficients. We show that deniability of our scheme reduces straightforwardly to Extended-LWE, and then follow the methodology of Alperin-Sheriff and Peikert [AP12] to reduce it further to plain LWE.

Towards UNF-1KCA security, we slightly modify the Frodo split-KEM by introducing masking terms. As the name suggests, they are used to hide the partial information about secret keys. In Section 3.6.2 we discuss the necessity of this (perhaps seemingly artificial) change.

[2] Nevertheless, we found no practical attack on deniability/UNF-1KCA for FrodoKEX.

3.1.3 Additional Related Work

The security of X3DH has been modelled in detail by Cohn-Gordon et al. [CCD+20]. Vatandas et al. [VGIK20] investigate the deniability of X3DH and similar key exchange protocols under the deniability notion of Di Raimondo et al. [DGK06], requiring strong knowledge-of-exponent-type assumptions to prove X3DH secure. Dobson and Galbraith [DG22] propose a SIDH-based X3DH-like protocol which is unfortunately now broken [CD23]. Very recently, Kiltz et al. [KPRR23] prove a simplified version of X3DH tightly-secure in the generic group model under a new multi-user assumption supporting corruptions although do not allow the adversary to expose parties' session states.

Unger and Goldberg build a number of different DAKEs [UG15, UG18]. However, the protocols do not provide post-quantum guarantees: only in their later paper [UG18] is it suggested to add a PQ KEM for post-quantum *confidentiality* and the authors do not propose a more comprehensive hybrid protocol. Nevertheless, the protocols provide relatively strong online deniability (where a judge and a party can communicate while trying to frame another party) at the expense of stronger primitives like dual-receiver encryption and non-committing encryption.

Alwen et al. [ABH+21] introduce the notion of authenticated key encapsulation mechanism (AKEM) and some security definitions. AKEM captures the same primitive as a split-KEM, but we opted for the syntax and language of the latter as it was meant to be used in an X3DH-like protocol.

Cremers and Feltz [CF11] introduce peer-deniability, which captures the kind of participation deniability property we are after, namely that a party cannot deny using a system but can deny communicating with a particular party. However, their security notion does not require the simulator to output the session key nor the judge/adversary to distinguish between the real and simulated key, and so composability issues may arise from using it.

Related to split-KEM is *signcryption* [Zhe97], which in syntax is roughly its encryption analogue. We are not aware of any works that consider deniable signcryption in the post-quantum setting.

3.2 Split-KEM

The primitive at the core of our protocol is split-KEM, which we present in this section. It was first defined by Brendel et al. [BFG+20].

Definition 17 (Split-KEM)**.** A split-KEM sKEM is a tuple of four efficient algorithms $(\mathsf{KeyGenA}, \mathsf{KeyGenB}, \mathsf{Encaps}, \mathsf{Decaps})$ defined as follows:

- $(\mathsf{pk_A}, \mathsf{sk_A}) \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$ (resp. $(\mathsf{pk_B}, \mathsf{sk_B}) \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$): The key generation function of the first/second party takes the security parameter λ as input, and outputs a pair

of public/secret keys $(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{A})$ (resp. $(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{B})$).

- $(K, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{B})$: The encapsulation function takes the public key pk_A of party A and party B's secret key sk_B as inputs, and outputs a ciphertext ct and a key K.
- $K/\perp \leftarrow \mathsf{Decaps}(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{A}, \mathsf{ct})$: The decapsulation function takes the secret key sk_A of party A, their counterpart's public key pk_B and a ciphertext ct as inputs, and outputs a key K or the error symbol $\perp$.

We say a split-KEM is $(1-\delta)$-correct if

$$\Pr\left[K \neq K' : \begin{array}{l}(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{A}) \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda);\\ (\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{B}) \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda);\\ (K, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{B});\\ K' \leftarrow \mathsf{Decaps}(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{A}, \mathsf{ct})\end{array}\right] \leq \delta\,.$$

Intuitively, a split-KEM is similar to a normal KEM except material from both participants is used for encapsulation (i.e., the final key will depend on both parties' secret/public keys). In an X3DH-like protocol, it can be used to implicitly authenticate the party encapsulating. In the language of Brendel et al. [BFG+20], our notion of split-KEM is "asymmetric", as it is assumed that B always encapsulates and A always decapsulates. This is sufficient for our purpose, but we note that all the results presented in this chapter can be adapted to a symmetric split-KEM where KeyGenA = KeyGenB.

3.2.1 Security

We will need several security properties from the split-KEM to prove our whole protocol secure. We first define one-wayness (OW-CPA) for sKEM, which is very similar to the usual one for KEM and another new notion called IND-1BatchCCA. Looking ahead, we will show that any OW-CPA split-KEM can easily be transformed into a IND-1BatchCCA one in the (quantum) random oracle model or (Q)ROM (c.f. Chapter 2).

Definition 18 (split-KEM OW-CPA)**.** We consider the OW-CPA game defined in Figure 3.2. A split-KEM scheme $\mathsf{sKEM} = (\mathsf{KeyGenA}, \mathsf{KeyGenB}, \mathsf{Encaps}, \mathsf{Decaps})$ is *OW-CPA* if for any efficient adversary $\mathscr{A}$ we have

$$\mathsf{Adv}^{\text{ow-cpa}}_{\mathsf{sKEM}}(\mathscr{A}) = \Pr[\mathsf{OW\text{-}CPA}_{\mathsf{sKEM}}(\mathscr{A}) \Rightarrow 1] = \mathsf{negl}\,.$$

Definition 19 (split-KEM IND-1BatchCCA)**.** We consider the IND-1BatchCCA game defined in Figure 3.2. Let $\mathscr{K}$ be a finite key space. A split-KEM scheme over $\mathscr{K}$ $\mathsf{sKEM} = (\mathsf{KeyGenA}, \mathsf{KeyGenB}, \mathsf{Encaps}, \mathsf{Decaps})$ is IND-1BatchCCA if for any efficient adversary $\mathscr{A}$ we have

$$\mathsf{Adv}^{\text{ind-1batchcca}}_{\mathsf{sKEM}}(\mathscr{A}) := \left|\Pr[\mathsf{IND\text{-}1BatchCCA}_{\mathsf{sKEM}}(\mathscr{A}) \Rightarrow 1] - \frac{1}{2}\right| = \mathsf{negl}\,.$$

Game IND-1BatchCCA$_{\mathsf{sKEM}}(\mathscr{A})$

1: $b \xleftarrow{\$} \{0,1\};\ q \leftarrow 0$
2: $\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{A} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
3: $\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{B} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
4: $K_0, \mathsf{ct}^* \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{B})$
5: $K_1 \xleftarrow{\$} \mathscr{K}$
6: $b' \xleftarrow{\$} \mathscr{A}^{\mathsf{BatchDEC}}(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}^*, K_b)$
7: **return** $\mathbb{1}[\![b' = b]\!]$

Oracle BatchDEC$(\{(\mathsf{pk}_i, \mathsf{ct}_i)\}_{i=1}^d)$

1: **if** $q = 1$: **return** $\perp$
2: **else** : $q \leftarrow q + 1$
3: **for** $i \in \{1, \ldots, d\}$
4: **if** $(\mathsf{pk}_i, \mathsf{ct}_i) = (\mathsf{pk}_\mathsf{B}, \mathsf{ct}^*)$: **return** $\perp$
5: $K'_i \leftarrow \mathsf{Decaps}(\mathsf{pk}_i, \mathsf{sk}_\mathsf{A}, \mathsf{ct}_i)$
6: **if** $K'_1 = \perp \vee \ldots \vee K'_d = \perp$: **return** $\perp$
7: **return** $(K'_1, \ldots, K'_d)$

Game OW-CPA$_{\mathsf{sKEM}}(\mathscr{A})$

1: $\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{A} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda);\ \mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{B} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
2: $K^*, \mathsf{ct}^* \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{B})$
3: $K' \xleftarrow{\$} \mathscr{A}(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}^*)$
4: **return** $\mathbb{1}[\![K' = K^*]\!]$

Figure 3.2: IND-1BatchCCA and OW-CPA games.

We also recall the different notions of indistinguishability for (asymmetric) split-KEM defined by Brendel et al. [BFG+20] that we do not use except to argue their insufficiency for AKE below:

Game xy-IND-CCA$_{\mathsf{sKEM}}(\mathscr{A})$

1: $b \xleftarrow{\$} \{0,1\}$
2: $n_\mathsf{x} \leftarrow 0; n_\mathsf{y} \leftarrow 0;$
3: $\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{A} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
4: $\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{B} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
5: $K_0, \mathsf{ct}^* \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{B})$
6: $K_1 \xleftarrow{\$} \mathscr{K}$
7: $b' \xleftarrow{\$} \mathscr{A}^{\mathsf{ENC},\mathsf{DEC}}(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}^*, K_b)$
8: **return** $\mathbb{1}[\![b' = b]\!]$

Oracle ENC(pk)

1: **if** $n_\mathsf{y} \geq \mathsf{y}$: **return** $\perp$
2: $n_\mathsf{y} \leftarrow n_\mathsf{y} + 1$
3: $K, \mathsf{ct} \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}, \mathsf{sk}_\mathsf{B})$
4: **if** $(\mathsf{pk}, \mathsf{ct}) = (\mathsf{pk}_\mathsf{A}, \mathsf{ct}^*)$: **return** $\perp$
5: **return** (K, ct)

Oracle DEC(pk, ct)

1: **if** $n_\mathsf{x} \geq \mathsf{x}$: **return** $\perp$
2: $n_\mathsf{x} \leftarrow n_\mathsf{x} + 1$
3: **if** $(\mathsf{pk}, \mathsf{ct}) = (\mathsf{pk}_\mathsf{B}, \mathsf{ct}^*)$: **return** $\perp$
4: **return** $\mathsf{Decaps}(\mathsf{pk}_{\overline{\mathsf{P}}}, \mathsf{sk}_\mathsf{P}, \mathsf{ct})$

Figure 3.3: xy-IND-CCA games for an "asymmetric" split-KEM from Brendel et al. [BFG+20], where $\mathsf{x}, \mathsf{y} \in \{\mathsf{n}, \mathsf{s}, \mathsf{m}\}$. When doing the comparison on the first line of both oracles, we assume $\mathsf{n} = 0$, $\mathsf{s} = 1$ and $\mathsf{m} = \infty$.

Definition 20 (split-KEM xy-IND-CCA)**.** We consider the xy-IND-CCA game defined in Figure 3.3. A split-KEM scheme sKEM = (KeyGenA, KeyGenB, Encaps, Decaps) is xy-IND-CCA,

with $\mathsf{x},\mathsf{y} \in \{\mathsf{n},\mathsf{s},\mathsf{m}\}$ if for any efficient adversary $\mathscr{A}$ we have

$$\mathsf{Adv}^{\text{xy-ind-cca}}_{\mathsf{sKEM}}(\mathscr{A}) := \left|\Pr\left[\mathsf{xy\text{-}IND\text{-}CCA}_{\mathsf{sKEM}}(\mathscr{A}) \Rightarrow 1\right] - \frac{1}{2}\right| = \mathsf{negl}\,.$$

These indistinguishability notions range from nn-IND-CCA, which is similar to a kind of IND-CPA security as the adversary has no access to encapsulation or decapsulation oracles, to mm-IND-CCA, which captures strong IND-CCA security for split-KEMs. More generally, all notions are of the form xy-IND-CCA, $\mathsf{x},\mathsf{y} \in \{\mathsf{n},\mathsf{s},\mathsf{m}\}$, where x (resp. y) specifies the number of queries an adversary can make to the decapsulation (resp. encapsulation) oracle (none, single, or many).

On the Original Split-KEM Security Notions. We recall that the advantage of split-KEMs over normal KEMs is that they capture the fact that the party encapsulating can contribute (static) keying material towards the shared key, whereas it is not the case with KEMs, as the encapsulation function only takes the receiving party's public key as input. In particular, this means that KEMs cannot be used for implicit authentication of the encapsulator, unlike split-KEMs. However, we argue that the original xy-IND-CCA definitions for split-KEMs [BFG+20] do not capture implicit authentication either and thus are not suited for their purpose (building an asynchronous DAKE). In fact, any IND-CPA (resp. IND-CCA) KEM can easily be converted to an (asymmetric) split-KEM satisfying nn-IND-CCA (resp. mm-IND-CCA).

More formally, imagine a setting where Alice and Bob know each other's public key, and Bob wants to implicitly authenticate to Alice using a split-KEM. In addition, we assume a mm-IND-CCA split-KEM sKEM_0 exists (note mm-IND-CCA security is the strongest so this holds for all weaker notions). We first modify sKEM_0 such that on a special ciphertext $\mathsf{ct}^\star$ not in the original ciphertext space, Decaps returns a constant key $K^\star$. Let's call this modified scheme sKEM. We observe that sKEM is still mm-IND-CCA secure as no adversary can break an honestly-generated challenge ciphertext. Now, implicit authentication means that if Alice decapsulates a ciphertext and obtains a key K, then only Alice knows K. However, in our case, any adversary can send $\mathsf{ct}^\star$ to Alice and set their own key to $K^\star$. Both the adversary and Alice will share the same key and implicit authentication does not hold. In a way, xy-IND-CCA security does not prevent forgeries.

UNF-1KCA. This leads us to define our notion of UNF-1KCA security for split-KEMs below which, along with OW-CPA (which can be turned into IND-1BatchCCA), guarantees that only Bob (and obviously Alice) can know the result of Alice's decapsulation on some ciphertext. More precisely, UNF-1KCA ensures that no adversary can forge a valid split-KEM ciphertext for A even knowing a ciphertext that was computed with respect to a public key chosen by the adversary[3], under the condition that the public key used for encapsulation and the known ciphertext are different from the pair made of A's public key and the ciphertext output

[3] Looking ahead, the fact that the public key is adversarially-chosen will be useful for proving security under key-compromise impersonation attacks for our full protocol.

by the adversary. We also define a security notion called decaps-CPA that will serve as a building block to build UNF-1KCA. The decaps-CPA notion ensures that it is hard for an adversary knowing a ciphertext ct (under an adversarially-chosen public key) to come up with a ciphertext ct' (possibly equal to ct) and a key K' such that the decapsulation of ct' returns K'.

Definition 21 (split-KEM UNF-1KCA)**.** We consider the UNF-1KCA game defined in Figure 3.4. A split-KEM scheme $\mathsf{sKEM} = (\mathsf{KeyGenA}, \mathsf{KeyGenB}, \mathsf{Encaps}, \mathsf{Decaps})$ is UNF-1KCA if for any efficient adversary $\mathscr{A}$ we have

$$\mathsf{Adv}^{\text{unf-1kca}}_{\mathsf{sKEM}}(\mathscr{A}) := \Pr\left[\mathsf{UNF\text{-}1KCA}_{\mathsf{sKEM}}(\mathscr{A}) \Rightarrow 1\right] = \mathsf{negl}\,.$$

Definition 22 (split-KEM decaps-CPA)**.** We consider the decaps-CPA game defined in Figure 3.4. A split-KEM scheme $\mathsf{sKEM} = (\mathsf{KeyGenA}, \mathsf{KeyGenB}, \mathsf{Encaps}, \mathsf{Decaps})$ is decaps-CPA if for any efficient adversary $\mathscr{A}$ we have

$$\mathsf{Adv}^{\text{decaps-cpa}}_{\mathsf{sKEM}}(\mathscr{A}) := \left|\Pr\left[\mathsf{decaps\text{-}CPA}_{\mathsf{sKEM}}(\mathscr{A}) \Rightarrow 1\right] - \frac{1}{2}\right| = \mathsf{negl}\,.$$

Game $\mathsf{UNF\text{-}1KCA}_{\mathsf{sKEM}}(\mathscr{A})$	Game $\mathsf{decaps\text{-}CPA}_{\mathsf{sKEM}}(\mathscr{A})$
1 : $\mathsf{pk_A}, \mathsf{sk_A} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$	1 : $\mathsf{pk_A}, \mathsf{sk_A} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
2 : $\mathsf{pk_B}, \mathsf{sk_B} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$	2 : $\mathsf{pk_B}, \mathsf{sk_B} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
3 : $\mathsf{pk}, \mathsf{st} \xleftarrow{\$} \mathscr{A}(\mathsf{pk_A}, \mathsf{pk_B})$	3 : $\mathsf{pk}, \mathsf{st} \xleftarrow{\$} \mathscr{A}(\mathsf{pk_A}, \mathsf{pk_B})$
4 : $K_\mathsf{B}, \mathsf{ct} \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}, \mathsf{sk_B})$	4 : $K_\mathsf{B}, \mathsf{ct} \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}, \mathsf{sk_B})$
5 : $\mathsf{ct}' \xleftarrow{\$} \mathscr{A}(\mathsf{pk_A}, \mathsf{pk_B}, \mathsf{ct}, K_\mathsf{B}, \mathsf{st})$	5 : $K'_\mathsf{A}, \mathsf{ct}' \xleftarrow{\$} \mathscr{A}(\mathsf{pk_A}, \mathsf{pk_B}, \mathsf{ct}, \mathsf{st})$
6 : **if** $(\mathsf{ct}, \mathsf{pk}) = (\mathsf{ct}', \mathsf{pk_A})$: **return** 0	6 : $K_\mathsf{A} \leftarrow \mathsf{Decaps}(\mathsf{pk_B}, \mathsf{sk_A}, \mathsf{ct}')$
7 : $K_\mathsf{A} \leftarrow \mathsf{Decaps}(\mathsf{pk_B}, \mathsf{sk_A}, \mathsf{ct}')$	7 : **if** $K_\mathsf{A} = \perp$: **abort**
8 : **if** $K_\mathsf{A} = \perp$: **return** 0	8 : **return** $\mathbb{1}[\![K_\mathsf{A} = K'_\mathsf{A}]\!]$
9 : **return** 1	

Figure 3.4: Games UNF-1KCA and decaps-CPA.

3.2.2 Deniability

Finally, we state the notion of split-KEM deniability we would like to achieve.

Definition 23 (Deniability)**.** We consider the game shown in Figure 3.5. We say a split-KEM sKEM is *deniable* if there exists a simulator Sim s.t. for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\text{deny}}_{\mathsf{sKEM},\mathsf{Sim}}(\mathscr{A}) := \left|\Pr[\mathsf{DENY}^{\mathsf{REAL}}_{\mathsf{sKEM},\mathsf{Sim}}(\mathscr{A}) \Rightarrow 1] - \Pr[\mathsf{DENY}^{\mathsf{SIM}}_{\mathsf{sKEM},\mathsf{Sim}}(\mathscr{A}) \Rightarrow 1]\right| = \mathsf{negl}\,.$$

Informally, the setting considered is the following. Alice and Bob use the split-KEM to establish a shared key (we assume the public keys are only used for this one exchange), and Alice (while

Game $\mathsf{DENY}^{\mathsf{REAL}}_{\mathsf{sKEM},\mathsf{Sim}}(\mathscr{A})$	Game $\mathsf{DENY}^{\mathsf{SIM}}_{\mathsf{sKEM},\mathsf{Sim}}(\mathscr{A})$
1 : $(\mathsf{pk}_\mathsf{A},\mathsf{sk}_\mathsf{A}) \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$	1 : $(\mathsf{pk}_\mathsf{A},\mathsf{sk}_\mathsf{A}) \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
2 : $(\mathsf{pk}_\mathsf{B},\mathsf{sk}_\mathsf{B}) \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$	2 : $(\mathsf{pk}_\mathsf{B},\mathsf{sk}_\mathsf{B}) \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
3 : $K,\mathsf{ct} \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}_\mathsf{A},\mathsf{sk}_\mathsf{B})$	3 : $K,\mathsf{ct} \xleftarrow{\$} \mathsf{Sim}(\mathsf{pk}_\mathsf{B},\mathsf{sk}_\mathsf{A})$
4 : $b \xleftarrow{\$} \mathscr{A}(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B},\mathsf{sk}_\mathsf{A},K,\mathsf{ct})$	4 : $b \xleftarrow{\$} \mathscr{A}(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B},\mathsf{sk}_\mathsf{A},K,\mathsf{ct})$
5 : **return** b	5 : **return** b

Figure 3.5: Deniability game (we assume w.l.o.g. that B encapsulates and A simulates).

following the protocol) wants to frame Bob and prove that he did communicate with her. Therefore, after receiving Bob's ciphertext and deriving the key, Alice gives both public keys, the derived key, the ciphertext and her own secret key to a judge (i.e., the adversary) that must decide whether Bob actually sent the ciphertext that was used to derive the key or not. The scheme is deniable if there is a simulator that, given Alice's view, outputs a ciphertext and a key indistinguishable from the ones output by Bob. We discuss deniability further after introducing our key exchange deniability notion (Section 3.3.3) and in Section 3.6.

3.3 Deniable Authenticated Key Exchange

In this section, we describe our model for deniable authenticated key exchange (DAKE) that we tailor to the semantics and flow of X3DH.

3.3.1 Syntax

Definition 24. A DAKE DAKE is a tuple of four efficient algorithms $(\mathsf{KeyGen},\mathsf{Init},\mathsf{Send},\mathsf{BatchReceive})$ defined as follows:

- $(\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$. This function takes as input the security parameter λ and outputs the long-term public/secret key pair of the caller.
- $(\mathsf{st}_i,\mathsf{prek}_i) \xleftarrow{\$} \mathsf{Init}(\mathsf{sk}_i,\mathsf{role})$. This function takes as inputs a long-term secret key sk_i and a role $\mathsf{role} \in \{\mathsf{sender},\mathsf{receiver}\}$ and outputs a session state st_i and a prekey bundle prek_i. Init models the creation of key material that will be uploaded to the public key infrastructure by both parties (e.g., a prekey bundle in X3DH). The output values depend only on the public key of party i executing the function.
- $(\mathsf{k},m) \xleftarrow{\$} \mathsf{Send}(\mathsf{sk}_i,\mathsf{pk}_j,\mathsf{st}_i,\mathsf{prek}_j)$. This function takes as inputs the secret key of the executing party i, the public key of the intended recipient pk_j, party i's session state st_i and the (claimed) prekey bundle of the intended recipient prek_j, and outputs a key k and a message m.
- $\{k_s\}_s \leftarrow \mathsf{BatchReceive}(\mathsf{sk}_i,\mathsf{st}_i,\{\mathsf{pk}_j,\mathsf{prek}_j,m_j\}_j)$. This function takes as inputs the secret

key of the executing party i, an ephemeral state of party i st_i and a vector of size $d \geq 1$ of the form $(\mathsf{pk}_j, \mathsf{prek}_j, m_j)$ for party i's session with the public key of the (claimed) sender pk_j, the (claimed) prekey bundle of party j prek_j and a message m_j, and outputs a vector of d keys $(\mathsf{k}_1, \ldots, \mathsf{k}_d)$, some or all of which may be $\perp$.

Init explicitly captures parties uploading ephemeral keys to a central server in the first protocol step. This contrasts with the formal modelling in some previous works on X3DH-like key exchange [BFG+22b, HKKP22] that model a three-move key exchange with a single initiator. As Init is independent of keying material from the caller's counterpart, our definition captures so-called *receiver obliviousness* [HKKP22] (sometimes *post-specified peers* [CK02]), corresponding to some, but not all, key exchange protocols in the literature.

The most novel part of our primitive is BatchReceive which in particular captures ephemeral key reuse when uploaded ephemeral keys are exhausted. In the case of key exhaustion, when a party comes back online, they execute BatchReceive several times (once per ephemeral state st_i), where the number of inputs of the form $(\mathsf{pk}_j, \mathsf{prek}_j, m_j)$ in a given BatchReceive call corresponds to how many times st_i is re-used. Otherwise, BatchReceive can be used as in standard AKE with a single value $(\mathsf{pk}_j, \mathsf{prek}_j, m_j)$ as input. Note that BatchReceive takes sender prekey bundles prek_j as input to emphasise that they are generated via Init and not Send, although each bundle could in principle be contained in each m_j.

3.3.2 Security Model

We now describe the security model we consider for our DAKE, which extends existing models in some ways to support BatchReceive.

Parties and Sessions. We assume that there are n parties $P_1, \ldots P_n$ (or $1, \ldots, n$) where party P_i (resp. or i) is associated with long-term key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ output by KeyGen. Each party runs one or more *sessions* (sometimes called *oracles* [BFG+20]), where the s-th session of P_i is denoted by π_i^s. Each session π_i^s is associated with the following local fields:

- sid, the session identifier or session id.
- pid, the partner identifier.
- role $\in \{\perp, \mathsf{sender}, \mathsf{receiver}\}$, the role of P_i.
- status $\in \{\perp, \mathsf{accept}, \mathsf{reject}\}$, the status of π_i^s.
- k, the session key.
- st, the session state.
- r, the session randomness.

All fields are initialised to $\perp$ except r which is initialised to uniform randomness. A session either has role sender or receiver, and its counterpart, its *partner* pid, has the other role; note a receiver may have several counterparts (capturing ephemeral key reuse).

Fields pid, role, status and r in session π_i^s are set directly by the challenger, and the rest are (sometimes implicitly) set by the underlying DAKE algorithms called by the challenger. Moreover, in light of the definition of BatchReceive, sid, pid and k are *vectors* for a receiver (role = receiver); we sometimes write $\vec{\text{sid}}$, $\vec{\text{pid}}$ and $\vec{\text{k}}$ for clarity to indicate this.

Suppose P_i is acting as a receiver. Initially, P_i calls Init, and then eventually calls BatchReceive. Before this point, one or more senders P_j (i.e., parties with role = sender) may call Init and then Send with respect to the output prek from P_i's Init call (assuming honest message delivery), which output messages of the form m_j. Finally, P_i invokes BatchReceive with one or more m_j values as input. A party has status = accept if and only if $\text{k} \neq \perp$[4], and stores any session state after calling Init and before setting status $\neq \perp$ due to a Send or BatchReceive call in st.

Partnering. We define partnering between two sessions to capture security using session identifiers:

Definition 25 (Partnering)**.** For any (i, P_j, s, t), we say that sessions π_i^s and π_j^t are *partners* if

1. $\pi_i^s.\text{role} \neq \pi_j^t.\text{role}$.
2. If $\pi_i^s.\text{role} = \text{sender}$, then $\pi_i^s.\text{pid} = j$ and $i \in \pi_j^t.\vec{\text{pid}}$. If $\pi_i^s.\text{role} = \text{receiver}$, then $j \in \pi_i^s.\vec{\text{pid}}$ and $i = \pi_j^t.\text{pid}$.
3. If $\pi_i^s.\text{role} = \text{sender}$, then $\pi_i^s.\text{sid} \in \pi_j^t.\vec{\text{sid}}$ and $\pi_i^s.\text{sid} \neq \perp$. If $\pi_i^s.\text{role} = \text{receiver}$, then $\pi_j^t.\text{sid} \in \pi_i^s.\vec{\text{sid}}$ and $\pi_j^t.\text{sid} \neq \perp$.

Looking ahead, this definition ensures that two sessions can only be partners if they both have set status = accept. Our definition mainly differs from previous work in that there can be many senders (and thus partnered sessions) for a given receiver. Ignoring this aspect, our definition is only slightly different from that of Hashimoto et al. [HKKP22] in that we restrict sid to be not equal to $\perp$; this is an artifact of the fact we model 'four-move' key exchange (including prekey uploading).

KIND **Security Game.** We first define key indistinguishability (KIND) and then define deniability separately. Following previous work, we define a KIND experiment played between a challenger C and adversary $\mathscr{A}$ in text below. The experiment $\text{KIND}^n_{\text{DAKE}}(\mathscr{A})$ is parameterised by the DAKE DAKE and integer n, the number of parties (honest or otherwise) in the lifetime of the game's execution. The game is divided into distinct phases defined as follows.

[4] In particular, BatchReceive may output several keys; as long as at least one of them is not $\perp$, the calling party accepts.

Setup. C first uniformly samples challenge bit $b \in \{0,1\}$. Then, for each party P_i, C calls $(\mathsf{pk}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ and provides $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ and 1^λ as input to $\mathscr{A}$.

Phase 1. $\mathscr{A}$ adaptively makes any number of the following queries in any order:

- $\mathsf{EXEC}(i, s, \mathsf{prek}, m)$: $\mathscr{A}$ starts or runs the next step of execution in session π_i^s. In each call, C uses randomness tape $\pi_i^s.\mathsf{r}$ as needed.
  - To start the execution in session π_i^s not previously started, $\mathscr{A}$ calls $\mathsf{EXEC}(i, s, \mathsf{prek}, m)$ with special input $m = (\mathtt{start}, \mathsf{sender}, j)$ (resp. $(\mathtt{start}, \mathsf{receiver}, \vec{j})$) (where $\mathtt{start}$ is defined only in the context of this game) that, if not previously called, sets $\pi_i^s.\mathsf{pid} = j$ (resp. $\pi_i^s.\mathsf{pid} = \vec{j}$) and $\pi_i^s.\mathsf{role} = \mathsf{sender}$ (resp. $\pi_i^s.\mathsf{role} = \mathsf{receiver}$); observe input prek is ignored by C. Then, C invokes $(\mathsf{st}_i, \mathsf{prek}_i) \xleftarrow{\$} \mathsf{Init}(\mathsf{sk}_i, \mathsf{role})$ and outputs prek_i to $\mathscr{A}$.
  - Given that P_i has started in π_i^s, $\pi_i^s.\mathsf{status} = \perp$ and $\pi_i^s.\mathsf{role} = \mathsf{sender}$, when $\mathscr{A}$ calls $\mathsf{EXEC}(i, s, \mathsf{prek}, \perp)$, C invokes $(\mathsf{k}, m) \xleftarrow{\$} \mathsf{Send}(\mathsf{sk}_i, \mathsf{pk}_j, \mathsf{st}_i, \mathsf{prek})$ (where $j = \pi_i^s.\mathsf{pid}$), returns output m to $\mathscr{A}$ and sets $\pi_i^s.\mathsf{status}$ to reject (resp. accept) if $\mathsf{k} = \perp$ (resp. $\mathsf{k} \neq \perp$).
  - If $\pi_i^s.\mathsf{role} = \mathsf{receiver}$ and $\pi_i^s.\mathsf{status} = \perp$, when $\mathscr{A}$ calls $\mathsf{EXEC}(i, s, \{s_j, \mathsf{prek}_j, m_j\}_{j \in \vec{j}'})$, C aborts if $\vec{j}' \neq \pi_i^s.\mathsf{pid}$ and otherwise invokes $\mathsf{k} \leftarrow \mathsf{BatchReceive}(\mathsf{sk}_i, \mathsf{st}_i, \{\mathsf{pk}_j, \mathsf{prek}_j, m_j\}_j)$ and outputs to $\mathscr{A}$ $\perp$ if BatchReceive fails (resp. nothing otherwise) and sets $\pi_i^s.\mathsf{status}$ to reject (resp. accept).
- $\mathsf{LTK}(i)$ outputs sk_i. P_i is hereafter *corrupted*.
- $\mathsf{REGISTER}(\mathsf{pk}_i, i)$ registers a new party P_i for $i > n$ not previously registered, sets their long-term public key to pk_i and distributes pk_i to all other oracles; P_i is immediately marked as *corrupted*.
- $\mathsf{STATE}(i, s)$ outputs $\pi_i^s.\mathsf{st}$, which is hereafter *revealed*.
- $\mathsf{KEY}(i, s, j)$ outputs $\pi_i^s.\mathsf{k}_j$ if $\pi_i^s.\mathsf{role} = \mathsf{receiver}$ and $\pi_i^s.\mathsf{status} \neq \perp$ and otherwise outputs $\pi_i^s.\mathsf{k}$.

Test. When $\mathscr{A}$ decides to move to the next phase, it issues the following query TEST which (if successful) returns either a real or random key:

- $\mathsf{TEST}(i, s, j)$: If $\pi_i^s.\mathsf{status} \neq \mathsf{accept}$, C returns $\perp$. Otherwise:
  - If $\pi_i^s.\mathsf{role} = \mathsf{sender}$, C aborts if $j \neq \pi_i^s.\mathsf{pid}$, and otherwise returns either $\pi_i^s.\mathsf{k}$ if $b = 0$ or a uniformly sampled key k if $b = 1$;
  - If $\pi_i^s.\mathsf{role} = \mathsf{receiver}$, C aborts if $j \notin \pi_i^s.\vec{\mathsf{pid}}$ or $\pi_i^s.\mathsf{k}_j = \perp$, and otherwise returns either $\pi_i^s.\mathsf{k}_j$ if $b = 0$ or a uniformly sampled key k if $b = 1$.

  At this point, π_i^s (which we say is with respect to key j if $\pi_i^s.\mathsf{role} = \mathsf{receiver}$) is said to be the *test session*.

Phase 2. $\mathscr{A}$ adaptively issues queries as in Phase 1.

Guess, Freshness and Correctness. After Phase 2, $\mathscr{A}$ outputs bit b'. Suppose that $\mathscr{A}$ made query $\mathsf{TEST}(i,s,j)$, i.e., π_i^s is the test session with respect to key j and $j \in \pi_i^s.\mathsf{pid}$ (with equality at least when $\pi_i^s.\mathsf{role} = \mathsf{sender}$). The following *freshness* conditions are checked by C; if any condition *is not satisfied*, C sets b' to a uniform bit (i.e., $\mathscr{A}$ gains no advantage):

1. $\mathsf{KEY}(i,s,j')$ has not been queried, where j' is arbitrary if $\pi_i^s.\mathsf{role} = \mathsf{sender}$ and $j' = j$ if $\pi_i^s.\mathsf{role} = \mathsf{receiver}$.
2. If π_i^s and π_j^t are partners, then $\mathsf{KEY}(j,t,i')$ has not been queried, where i' is arbitrary if $\pi_i^s.\mathsf{role} = \mathsf{sender}$ and $i' = i$ if $\pi_i^s.\mathsf{role} = \mathsf{receiver}$.
3. P_i is not corrupted or $\pi_i^s.\mathsf{st}$ has not been revealed.
4. If π_i^s and π_j^t are partners, then P_j is not corrupted or $\pi_j^t.\mathsf{st}$ has not been revealed.
5. If π_i^s has no partner session, then P_j is not corrupted when $\pi_i^s.\mathsf{status} = \perp$.
6. If π_i^s has no partner session, then if $\pi_i^s.\mathsf{role} = \mathsf{sender}$, for any session π_j^t such that prek_j was both output by $\mathsf{Init}(\mathsf{sk}_j, \mathsf{receiver})$ and input to Send in π_i^s by C, P_j is not corrupted or $\pi_j^t.\mathsf{st}$ is not revealed.
7. If π_i^s has no partner session, then if $\pi_i^s.\mathsf{role} = \mathsf{receiver}$, for any session π_j^t such that prek_j was both output by $\mathsf{Init}(\mathsf{sk}_j, \mathsf{sender})$ and input to $\mathsf{BatchReceive}$ in π_i^s by C, $\pi_j^t.\mathsf{st}$ is not revealed and $\pi_i^s.\mathsf{st}$ is not revealed.

Then, the following *correctness* conditions are checked by C which, iterating over all relevant parties i, j, k, only consider the subset of sessions corresponding to *honest* protocol runs where $\mathscr{A}$ faithfully follows the protocol specification. If any condition *is satisfied*, C sets $b = b'$ (i.e., $\mathscr{A}$ wins):

1. There exist distinct sessions π_i^s and π_j^t such that $\pi_i^s.\mathsf{role} = \pi_j^t.\mathsf{role}$ and either (1) $\pi_i^s = \mathsf{receiver}$ and $\pi_i^s.\mathsf{sid}_j = \pi_j^t.\mathsf{sid}_i$ or (2) $\pi_i^s.\mathsf{sid} = \pi_j^t.\mathsf{sid}$.
2. Assuming $\pi_i^s.\mathsf{role} = \mathsf{receiver}$, there exist sessions π_i^s with respect to key j and π_j^t that are partners such that $\pi_i^s.\mathsf{k}_j \neq \pi_j^t.\mathsf{k}$ (analogously when $\pi_i^s.\mathsf{role} = \mathsf{sender}$).
3. There exist distinct sessions π_i^s, π_j^t and π_k^u such that $\pi_i^s.\mathsf{status} = \pi_j^t.\mathsf{status} = \pi_k^u.\mathsf{status} = \mathsf{accept}$ and $\pi_i^s.\mathsf{sid}_k = \pi_j^t.\mathsf{sid}_k = \pi_k^u.\mathsf{sid}$ (assuming i, j are receivers here but analogously in other cases).

Finally, the game outputs 1 if and only if $b = b'$.

Security is formally defined in Definition 26.

Definition 26 (DAKE Key Indistinguishability)**.** We consider the KIND game described above. We say a DAKE DAKE is key indistinguishable if for all efficient adversaries $\mathscr{A}$ and polynomially-bounded n (the total number of parties), we have

$$\mathsf{Adv}^{\mathrm{kind}}_{\mathsf{DAKE},n}(\mathscr{A}) := \left|\Pr[\mathsf{KIND}^{n}_{\mathsf{DAKE}}(\mathscr{A}) \Rightarrow 1] - \frac{1}{2}\right| = \mathsf{negl}\,.$$

Discussion. Following previous work, we define freshness conditions to prevent the adversary from mounting trivial attacks. Conditions 1 to 5 correspond exactly to the forward-secure variant of security in [HKKP22]. Due to the design of our DAKE K-Waay, we additionally restrict the adversary via conditions 6 and 7. The clauses in these conditions are essentially due to the fact that in K-Waay the only secret keying material required to call Send is an ephemeral split-KEM secret. For example, suppose that the tester π_i^s is the receiver. Due to the 'symmetric' nature of split-KEM, without these restrictions, revealing π_i^s.st allows the adversary to inject to P_i by simulating Send (akin to a key-compromise impersonation (KCI) attack using P_i's ephemeral state) and trivially distinguish. Consequently, we restrict session state exposure in this case.

Our model does not support randomness exposure or manipulation. As is standard, however, one can employ the NAXOS trick [LLM07] to obtain security given, e.g., randomness but not long-term keying material is exposed. Note also that we do not force Init to be called, e.g., by the sender or senders first or Init to be called by both the sender or senders and receiver before a party calls Send or BatchReceive.

Apart from the fact we make several extensions to typical AKE modelling to capture BatchReceive, the game is closest to that of Hashimoto et al. [HKKP22] except that we additionally enforce correctness checks as Brendel et al. [BFG+22b] do. To capture partnering, we consider partner and key identifiers that may be *vectors* for a receiver, such that several sender sessions may be partnered with a receiver session if, for a given sender session, it partners with a part/component of the receiver session. We do not capture semi-static keys explicitly as in Brendel et al.'s work [BFG+22b], although in principle they could be captured in Init. Like Hashimoto et al.'s [HKKP22], our game supports message injection, session state exposure or revealing (unlike Brendel et al.), session key exposure, long-term key exposure (corruption) and adversarial long-term key registration (also considered corruption). Following previous work, we allow the adversary to expose session states individually, rather than expose all states of a given party at once; security w.r.t. this weaker class of attacks is thus tightly implied by our notion. During execution, a single challenge test query is made by the adversary that reveals a real or random key output in some session. For BatchReceive which can output several keys, just one of the output keys are tested.

Trivial Attacks. We restrict the adversary's behaviour to prevent 'trivial' attacks (e.g. directly revealing the challenge key) by defining *freshness* predicates. Due to our protocol's design, our notion restricts more than the full forward security notion under session state exposure

defined by Hashimoto et al. [HKKP22]. Our freshness predicates imply weak forward secrecy and implicit authentication given session state exposure is not allowed (enforced in some recent works like [BHJ+15, CCG+19]). Brendel et al.'s model provide these guarantees but additionally protect against randomness exposure [BFG+22b], whereas we allow exposures on session states under some conditions unlike them.

3.3.3 Deniability

We next introduce our security notion for a deniable DAKE. To this end, we introduce security game $\mathsf{DENY}^{exp}_{\mathsf{DAKE},\mathsf{Sim}}$ in Figure 3.6.

Game $\mathsf{DENY}^{exp}_{\mathsf{DAKE},n,\mathsf{Sim}}(\mathscr{A})$	Oracle $\mathsf{CHAL}(i,j)$
1: $b \xleftarrow{\$} \{0,1\}$	1: **require** $i \in [n] \wedge j \in [n]$
2: $L \leftarrow \emptyset$	2: $(k,m) \leftarrow (\bot,\bot)$
3: **for** $i \in [n]$:	3: $(\mathsf{st}_i, \mathsf{prek}_i) \xleftarrow{\$} \mathsf{Init}(\mathsf{sk}_i, \mathsf{sender})$
4: $\quad (\mathsf{pk}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$	4: $(\mathsf{st}_j, \mathsf{prek}_j) \xleftarrow{\$} \mathsf{Init}(\mathsf{sk}_j, \mathsf{receiver})$
5: $\quad L \leftarrow L \cup \{(\mathsf{pk}_i, \mathsf{sk}_i)\}$	5: **if** $b = 0$: $(\mathsf{k}, m) \xleftarrow{\$} \mathsf{Send}(\mathsf{sk}_i, \mathsf{pk}_j, \mathsf{st}_i, \mathsf{prek}_j)$
6: $b' \xleftarrow{\$} \mathscr{A}^{\mathsf{CHAL}}(L)$	6: **else** : $(k,m) \xleftarrow{\$} \mathsf{Sim}(\mathsf{sk}_j, \mathsf{pk}_i, \mathsf{st}_j, \mathsf{prek}_i, \mathsf{prek}_j)$
7: **return** $1[\![b' = b]\!]$	7: $T \leftarrow (\mathsf{prek}_i, \mathsf{prek}_j, m)$
	8: **if** $exp = \mathsf{true}$: **return** $(\mathsf{k}, T, \mathsf{st}_j)$
	9: **else** : **return** (k, T)

Figure 3.6: Deniability game.

Definition 27 (DAKE deniability)**.** We consider the game shown in Figure 3.6. We say a DAKE DAKE is DENY^{exp} for $exp \in \{\mathsf{true}, \mathsf{false}\}$ if there exists an efficient simulator Sim s.t. for all efficient adversaries $\mathscr{A}$ and polynomially-bounded n, we have

$$\mathsf{Adv}^{\mathrm{deny}}_{\mathsf{DAKE},\mathsf{Sim},\mathsf{exp}}(\mathscr{A}) := \left| \Pr[\mathsf{DENY}^{exp}_{\mathsf{DAKE},n,\mathsf{Sim}}(\mathscr{A}) \Rightarrow 1] - \frac{1}{2} \right| = \mathsf{negl}\,.$$

Our definition captures the following deniability property. Initially, the judge $\mathscr{A}$ is given the long-term keys of all parties. $\mathscr{A}$ then observes honest protocol runs between pairs of parties (via CHAL). Depending on the challenge bit b, either Send or a simulator Sim that takes as input the secret keying material of the receiver trying to frame the sender is executed in each run. Moreover, $\mathscr{A}$ is given the prekey messages independent of b and, if the parameter exp is set to true, also the session state of the receiver in each protocol run. The goal of the adversary is to distinguish whether Send or Sim is being called.

Our notion $\mathsf{DENY}^{\mathsf{false}}$ corresponds most closely with that of Brendel et al. [BFG+22b] which was also adopted by Cremers et al. [CZ24]. Due to how Brendel et al.'s AKE primitive is defined, they also consider semi-static key pairs which are also given to the adversary. $\mathsf{DENY}^{\mathsf{true}}$

provides stronger deniability, corresponding in practice to a receiver who co-operates with a judge by handing over the entire contents of their device. Although incomparable formally, our DAKE would not be considered deniable under a notion like that of Brendel et al. [BFG+22b] since their protocol does not formally model long-term signatures. Note that our definition, like Brendel et al.'s [BFG+22b], can be straightforwardly converted to a "simulation-based" notion like Definition 23.

Finally, observe that our definition, like that of Brendel et al. [BFG+22b] does not consider deniability for the receiver but only for the sender. One could define such a notion, in which the goal is for the judge (adversary) to distinguish between the output of BatchReceive and a simulator Sim that has access to the long-term and ephemeral states of all corresponding senders and is given (honest) ciphertexts output by Send as input. Here, one could argue deniability for K-Waay using the security of the ephemeral KEM and then the KDF. A weaker definition would require the judge to distinguish between the output of Send *and* BatchReceive calls, and the output of a simulator Sim given the senders' states, which is trivial to satisfy (and thus we did not capture it) but is closer to offline semi-honest deniability (whereas the first notion sketched above is more 'online').

3.4 K-Waay: Post-Quantum X3DH from Split-KEM

In this section, we present our split-KEM-based protocol K-Waay before proving it satisfies our key indistinguishability and deniability security notions formalised in Section 3.3.

3.4.1 Construction

We present our DAKE K-Waay (Key-exchange With asynchrony, authentication and peer-deniability) in Figure 3.7.

Each party is associated with a long-term public/secret key pair which in K-Waay comprises of a signature and KEM key pair generated in KeyGen. In Init, ephemeral KEM and split-KEM keys for both parties are generated and the public keys are signed with the long-term signature key.

After initialisation, the sender P_i (sometimes called the initiator) invokes Send that takes the prekey prek_j output by the receiver P_j's Init call as input. After verifying the signature in prek_j, P_i encapsulates to (1) the long-term KEM key of P_j; (2) the ephemeral KEM key contained in prek_j; and (3) the ephemeral split-KEM key contained in prek_j. Note that the split-KEM provides implicit authentication (without it, Send could be simulated without secrets). Moreover, the sender's prekey bundle and in particular the split-KEM public key is generated *before* Send is called for deniability. P_i then combines the encapsulated keys using a KDF and outputs the key and its message for P_j consisting of the three encapsulation ciphertexts.

KeyGen(1^λ)

1: // Long-term key generation
2: $(\mathsf{kpk},\mathsf{ksk}) \xleftarrow{\$} \mathsf{KeyGen}_{\mathsf{LKEM}}(1^\lambda)$
3: $(\mathsf{spk},\mathsf{ssk}) \xleftarrow{\$} \mathsf{KeyGen}_{\mathsf{Sig}}(1^\lambda)$
4: $\mathsf{pk} \leftarrow (\mathsf{spk},\mathsf{kpk})$
5: $\mathsf{sk} \leftarrow (\mathsf{ssk},\mathsf{ksk})$
6: **return** $(\mathsf{pk},\mathsf{sk})$

Init(sk_i, role)

1: // Prekey generation/upload
2: **if** role = sender:
3: $\quad(\mathsf{espk}_i,\mathsf{essk}_i) \xleftarrow{\$} \mathsf{KeyGenB}_{\mathsf{sKEM}}(1^\lambda)$
4: $\quad\mathsf{ekpk}_i \leftarrow \perp$
5: **else**:
6: $\quad(\mathsf{espk}_i,\mathsf{essk}_i) \xleftarrow{\$} \mathsf{KeyGenA}_{\mathsf{sKEM}}(1^\lambda)$
7: $\quad(\mathsf{ekpk}_i,\mathsf{eksk}_i) \xleftarrow{\$} \mathsf{KeyGen}_{\mathsf{EKEM}}(1^\lambda)$
8: $\sigma_i \xleftarrow{\$} \mathsf{Sign}_{\mathsf{Sig}}(\mathsf{sk}_i.\mathsf{ssk},(\mathsf{espk}_i,\mathsf{ekpk}_i))$
9: $\mathsf{prek}_i \leftarrow (\mathsf{espk}_i,\mathsf{ekpk}_i,\sigma_i)$
10: **return** $(\mathsf{st}_i = (\mathsf{essk}_i,\mathsf{eksk}_i,\mathsf{prek}_i),\mathsf{prek}_i)$

Send($\mathsf{sk}_i,\mathsf{pk}_j,\mathsf{st}_i,\mathsf{prek}_j$)

1: $(\mathsf{essk}_i,\mathsf{eksk}_i,\mathsf{prek}_i) \leftarrow \mathsf{st}_i$
2: $(\mathsf{espk}_j,\mathsf{ekpk}_j,\sigma_j) \leftarrow \mathsf{prek}_j$
3: $msg \leftarrow (\mathsf{espk}_j,\mathsf{ekpk}_j)$
4: **require** $\mathsf{Vrfy}_{\mathsf{Sig}}(\mathsf{pk}_j.\mathsf{spk},msg,\sigma_j)$
5: $(K_\ell,\mathsf{ct}_\ell) \xleftarrow{\$} \mathsf{Encaps}_{\mathsf{LKEM}}(\mathsf{pk}_j.\mathsf{kpk})$
6: $(K_k,\mathsf{ct}_k) \xleftarrow{\$} \mathsf{Encaps}_{\mathsf{EKEM}}(\mathsf{ekpk}_j)$
7: $(K_s,\mathsf{ct}_s) \xleftarrow{\$} \mathsf{Encaps}_{\mathsf{sKEM}}(\mathsf{espk}_j,\mathsf{essk}_i)$
8: $m \leftarrow (\mathsf{ct}_\ell,\mathsf{ct}_k,\mathsf{ct}_s)$
9: $\mathsf{sid} \leftarrow P_i||P_j||\mathsf{pk}_i||\mathsf{pk}_j||\mathsf{prek}_i||\mathsf{prek}_j||m$
10: $\mathsf{k} \leftarrow \mathsf{KDF}(K_\ell,K_k,K_s,\mathsf{sid})$
11: **return** (k,m)

BatchReceive($\mathsf{sk}_i,\mathsf{st}_i,S=\{\mathsf{pk}_j,\mathsf{prek}_j,m_j\}_j$)

1: $(\mathsf{essk}_i,\mathsf{eksk}_i,\mathsf{prek}_i) \leftarrow \mathsf{st}_i$
2: $\mathsf{fail} \leftarrow \mathsf{false}$; $\mathsf{k}_j \leftarrow \perp$
3: **for** j : $(\mathsf{pk}_j,\mathsf{prek}_j,m_j) \in S$:
4: $\quad(\mathsf{ct}_\ell,\mathsf{ct}_k,\mathsf{ct}_s) \leftarrow m_j$
5: $\quad(\mathsf{espk}_j,\mathsf{ekpk}_j,\sigma_j) \leftarrow \mathsf{prek}_j$
6: $\quad$**if** $\neg\mathsf{Vrfy}_{\mathsf{Sig}}(\mathsf{pk}_j.\mathsf{spk},(\mathsf{espk}_j,\mathsf{ekpk}_j),\sigma_j)$:
7: $\quad\quad\mathsf{k}_j \leftarrow \perp$
8: $\quad\quad$**continue**
9: $\quad K_\ell \leftarrow \mathsf{Decaps}_{\mathsf{LKEM}}(\mathsf{sk}_i.\mathsf{ksk},\mathsf{ct}_\ell)$
10: $\quad K_k \leftarrow \mathsf{Decaps}_{\mathsf{EKEM}}(\mathsf{eksk}_i,\mathsf{ct}_k)$
11: $\quad K_s \leftarrow \mathsf{Decaps}_{\mathsf{sKEM}}(\mathsf{espk}_j,\mathsf{essk}_i,\mathsf{ct}_s)$
12: $\quad\mathsf{sid} \leftarrow P_j||P_i||\mathsf{pk}_j||\mathsf{pk}_i||\mathsf{prek}_j||\mathsf{prek}_i||m_j$
13: $\quad$**if** $K_s = \perp$: $\mathsf{fail} \leftarrow \mathsf{true}$
14: $\quad$**if** $(K_\ell = \perp) \vee (K_k = \perp) \vee (K_s = \perp)$: $\mathsf{k}_j \leftarrow \perp$
15: $\quad$**else** : $\mathsf{k}_j \leftarrow \mathsf{KDF}(K_\ell,K_k,K_s,\mathsf{sid})$
16: **if** fail: **return** $\perp^{|S|}$
17: **else** : **return** $\{\mathsf{k}_j\}_j$

Figure 3.7: K-Waay: An X3DH-like DAKE from IND-CCA KEMs EKEM and LKEM, SUF-CMA signature scheme Sig and IND-1BatchCCA and UNF-1KCA split-KEM sKEM.

Receiving is analogous: receiver P_i verifies P_j's prekey, decapsulates using its three respective secret keys and derives the session key (recall though that sender prekey bundles were not required in previous X3DH protocols). If P_i's prekeys have run out, it is possible that multiple P_j's have sent using the same prekey prek_i. In that case, P_i decapsulates for all sessions using the same secret keys but aborts if *any* split-KEM decapsulations failed in *any* of the sessions (a signature check failing does not however lead to the receiver aborting). We assume that for a

given BatchReceive($\mathsf{sk}_i, \mathsf{st}_i, S$) call, each element of S corresponds to a different party.

3.4.2 Security

Theorem 1. Consider $(1-\delta_{\mathsf{EKEM}})$-correct IND-CCA KEM EKEM, $(1-\delta_{\mathsf{LKEM}})$-correct IND-CCA KEM LKEM, $(1-\delta_{\mathsf{Sig}})$-correct SUF-CMA signature scheme Sig and $(1-\delta_{\mathsf{sKEM}})$-correct IND-1BatchCCA, UNF-1KCA split-KEM sKEM and triple PRF KDF used to build K-Waay (Figure 3.7). Then, we have that for polynomially-bounded n and every efficient adversary $\mathscr{A}$ that makes at most q oracle queries, one can build an adversary $\mathscr{B}$ such that

$$\begin{aligned}\mathsf{Adv}^{\mathrm{kind}}_{\mathsf{K\text{-}Waay},n}(\mathscr{A}) \leq \frac{q}{3}\cdot\left(\delta_{\mathsf{Sig}}+\delta_{\mathsf{LKEM}}+\delta_{\mathsf{EKEM}}+\delta_{\mathsf{sKEM}}\right) + \\ 2q^2\cdot(\epsilon_{\mathsf{EKEM}}+\epsilon_{\mathsf{LKEM}}+2\epsilon_{\mathsf{KDF}}+2\epsilon_{\mathsf{Sig}}) + q^3\cdot\left(\epsilon_{\mathsf{EKEM}}+\epsilon_{\mathsf{LKEM}}+\epsilon_{\mathsf{sKEM}}+3\epsilon_{\mathsf{KDF}}\right),\end{aligned}$$

where $\epsilon_{\mathsf{EKEM}} = \mathsf{Adv}^{\mathrm{ind-cca}}_{\mathsf{EKEM}}(\mathscr{B})$, $\epsilon_{\mathsf{LKEM}} = \mathsf{Adv}^{\mathrm{ind-cca}}_{\mathsf{LKEM}}(\mathscr{B})$, $\epsilon_{\mathsf{Sig}} = \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{B})$, $\epsilon_{\mathsf{sKEM}} = \mathsf{Adv}^{\mathrm{ind-1batchcca}}_{\mathsf{sKEM}}(\mathscr{B}) + \mathsf{Adv}^{\mathrm{unf-1kca}}_{\mathsf{sKEM}}(\mathscr{B})$ and $\epsilon_{\mathsf{KDF}} = \mathsf{Adv}^{\mathrm{3prf}}_{\mathsf{KDF}}(\mathscr{B})$.

Proof. Our proof proceeds by constructing sequences of hybrids, which we first summarise. Let Game Γ_1 be exactly the KIND game played with respect to DAKE K-Waay (Figure 3.7). We first transition to Game Γ_2, which differs from Game Γ_1 in that honest protocol runs, all $\mathsf{Vrfy}_{\mathsf{Sig}}$ checks in BatchReceive calls are removed and Decaps calls are replaced by the output of the Encaps calls in the corresponding Send calls whenever they are consistent. To this end, we invoke the correctness of K-Waay's building blocks. Then, we transition to Game Γ_3 in which the challenger immediately outputs the session π_i^s that the adversary makes real-or-random challenge query $\mathsf{TEST}(i,s,j^*)$ with respect to. We then partition $\mathscr{A}$'s possible executions of Game Γ_3 into several events.

Suppose π_i^s has a partner session (with respect to key j^* if $\pi_i^s.\mathsf{role} = \mathsf{receiver}$) (event E_p), say π_j^t. Observe that by definition of partnering and construction of the protocol (in particular by definition of sid), it follows that partnered sessions correspond to *honest* protocol runs. Then, considering π_i^s and π_j^t, if the receiver's session state, say $\pi_j^t.\mathsf{st}$, is revealed (event $E_p \wedge E_{c1}$), we reduce to the IND-CCA security of the long-term KEM LKEM, since the freshness conditions imply P_j must not have been corrupted. Otherwise (event $E_p \wedge \neg E_{c1}$), we reduce to the IND-CCA security of the ephemeral KEM EKEM. After both cases, we transition to an unwinnable game by keying KDF with the now uniformly random key output by the respective KEM call, a transition we perform repeatedly and omit from this description hereafter. Otherwise (event $\neg E_p$), we consider whether party P_i in test session π_i^s has the role sender or receiver:

- $\pi_i^s.\mathsf{role} = \mathsf{sender}$ (event $\neg E_p \wedge E_s$): As P_j can only be corrupted after P_i accepts, we first use the SUF-CMA security of Sig to argue that P_i's Send call in the test session must be with honestly-generated input (prek). Then, let E_{c2} be the event that P_j is corrupted. Given $\neg E_p \wedge E_s \wedge E_{c2}$, we reduce to the security of EKEM, since by freshness the state

π_j^t.st associated with prek must not have been exposed. Otherwise ($\neg E_p \wedge E_s \wedge \neg E_{c2}$) we reduce to the security of LKEM.

- π_i^s.role = receiver (event $\neg E_p \wedge \neg E_s$): As above, we first argue using SUF-CMA security that input prek_j used in the test session's BatchReceive call must have been honestly generated. Then by freshness, we know that neither π_i^s.st nor π_j^t.st associated with prek_j are revealed, in which case we first reduce to the UNF-1KCA security of sKEM to prevent injections on the split-KEM ciphertext, after which we reduce to the IND-1BatchCCA security of sKEM.

Let $\mathsf{Adv}^{\mathrm{gi}}_{\mathsf{DAKE},n}(\mathscr{A})$ be the advantage of adversary $\mathscr{A}$ in winning game Game Γ_i for relevant i which we introduce below. Furthermore, let $\mathsf{Adv}^{\mathrm{gi}}_{\mathsf{DAKE},n}(\mathscr{A}, E)$ be the same advantage except restricted to event E, so in particular if $\mathsf{Adv}^{\mathrm{gi}}_{\mathsf{DAKE},n}(\mathscr{A})$ is of the form $\left|\Pr[X] - \frac{1}{2}\right|$, $\mathsf{Adv}^{\mathrm{gi}}_{\mathsf{DAKE},n}(\mathscr{A}, E)$ is of the form $\left|\Pr[X \wedge E] - \frac{1}{2}\right|$.

Game Γ_1: This is the original key indistinguishability game.

Game Γ_2: This differs from Game Γ_1 in that, in honest protocol runs, all signature verification calls in BatchReceive calls are removed and the output of Decaps calls are replaced with the output of the corresponding Encaps call in Send. It follows at this point that the three correctness checks in the KIND game evaluate to true. Since for a given $\mathsf{BatchReceive}(\cdot,\cdot,S)$ call there must be $|S|$ corresponding Send and Init calls, there are at most $q/3$ iterations of the **for** loop in BatchReceive (counting over all such calls in a given execution of Game Γ_1). It then follows from a standard hybrid argument and the correctness of Sig, LKEM, EKEM and sKEM that:

$$\mathsf{Adv}^{\mathrm{g1}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g2}}_{\mathsf{DAKE},n}(\mathscr{A}) + \frac{q}{3} \cdot \left(\delta_{\mathsf{Sig}} + \delta_{\mathsf{LKEM}} + \delta_{\mathsf{EKEM}} + \delta_{\mathsf{sKEM}}\right).$$

Game Γ_3: This differs from Game Γ_3 in that the challenger immediately outputs the session π_i^s that the adversary $\mathscr{A}$ calls $\mathsf{TEST}(i, s, j^*)$ with respect to. Noting that there are at most q such possible sessions and applying a standard argument, it follows that:

$$\mathsf{Adv}^{\mathrm{g2}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq q \cdot \mathsf{Adv}^{\mathrm{g3}}_{\mathsf{DAKE},n}(\mathscr{A}).$$

Case 1: Test session π_i^s is partnered (Game Γ_{3a} and Game Γ_{3b}):

Game $\Gamma_{3a.1}$: Let E_p be the event that test session π_i^s has a partner, say π_j^t. Let E_{c1} be the event that the ephemeral state st of the receiver (in π_i^s and π_j^t) is revealed. Games Game $\Gamma_{3a.i}$ are defined given $E_p \wedge E_{c1}$. Game $\Gamma_{3a.1}$ differs from Game Γ_3 in that the game initially outputs π_j^t, the partner of π_i^s (observe that $j = j^*$ where j^* is defined in the previous hop), as well as a bit indicting whether π_i^s is the sender or receiver. By the same reasoning as

above, we have

$$\mathsf{Adv}^{\mathrm{g3}}_{\mathsf{DAKE},n}(\mathscr{A}, E_p \wedge E_{c1}) \leq 2q \cdot \mathsf{Adv}^{\mathrm{g3a.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \ .$$

Game $\Gamma_{3a.2}$: Game $\Gamma_{3a.2}$ differs from Game $\Gamma_{3a.1}$ in that the output key K in the call to LKEM.Encaps and the corresponding LKEM.Decaps call or calls (which are guaranteed to exist given E_p, and K is identical by definition of Game Γ_2) made in the test and partner sessions with respect to the receiver's public key and secret key, respectively, are replaced with a key k uniformly sampled by the challenger. Observe that since E_{c1} holds, by freshness, P_j cannot be corrupted, and thus we reduce to the security of LKEM.

Let $\mathscr{A}'$ be a IND-CCA adversary who simulates for Game $\Gamma_{3a.1}$/Game $\Gamma_{3a.2}$ adversary $\mathscr{A}$ as follows. Let pk be the IND-CCA challenge public key, (ct^*, K^*) be the challenge ciphertext and key respectively.

In the Setup phase, $\mathscr{A}'$ uniformly samples bit b_{sim}, calls $(\mathsf{pk}_\ell, \mathsf{sk}_\ell) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ locally for $\ell \neq k$ where k is the sender, sets $\mathsf{pk}_k \leftarrow \mathsf{pk}$, and returns $\{\mathsf{pk}_1, \ldots \mathsf{pk}_n\}$ and 1^λ to $\mathscr{A}$. Observe here (and later for Game $\Gamma_{3b.2}$) that, since E_p holds, we have matching sid values for test session π_i^s and partner π_j^t. Note by construction of sid, the presence of substring $\mathsf{prek}_i || \mathsf{prek}_j$ and m in the common value sid implies that Send must have been called honestly in π_i^s and also in BatchReceive for tuple $(\mathsf{pk}_j, \mathsf{prek}_j, m_j)$ in π_j^t for E_p to hold. Thus, we do not need to consider injections in the test session itself (although we have to in general in the BatchReceive call).

Before proceeding, we argue that $\mathscr{A}'$ can simulate on behalf of parties with a maliciously-registered long-term key locally, which applies here and in the rest of the proof. Since π_i^s is partnered, as argued above, π_i^s and π_j^t correspond to honest (completed) executions, and so neither P_i and P_j can be malicious. For unpartnered sessions, since Send and BatchReceive cannot be called by the game, the test session π_i^s cannot be corrupted itself (since testing requires $\pi_i^s.\mathsf{status} \neq \perp$), and otherwise condition 5 restricts the non-tested party P_j from being corrupted, thus precluding its key from being registered maliciously. Finally, computation involving messages or prekey bundles from maliciously-registered parties does not require any secret material not already known to $\mathscr{A}'$.

In Phase 1, when $\mathscr{A}$ calls $\mathsf{EXEC}(k', \cdot, \cdot, \cdot)$ where k' corresponds to the sender in π_i^s and π_j^t and the challenger is supposed to invoke Send, $\mathscr{A}'$ replaces the call to $\mathsf{Encaps}_{\mathsf{LKEM}}$ with the output (ct^*, K^*), and otherwise simulates locally. When $\mathscr{A}$ calls $\mathsf{EXEC}(k, \cdot, \cdot, \cdot)$ corresponding to the receiver in π_i^s and π_j^t and the challenger is supposed to invoke BatchReceive, $\mathscr{A}'$ replaces the output of the relevant $\mathsf{Decaps}_{\mathsf{LKEM}}$ calls corresponding to either i or j, depending on who is the receiver, with K^* and the output of other calls $\mathsf{Decaps}_{\mathsf{LKEM}}$ with the output obtained from $\mathsf{DEC}(\cdot)$; $\mathscr{A}'$ otherwise simulates locally.

In the Test phase, i.e., when $\mathscr{A}$ calls $\mathsf{TEST}(i, s, j)$, $\mathscr{A}'$ simulates with respect to bit b_{sim}. $\mathscr{A}'$ then simulates Phase 2 as above and the rest of the game locally, ultimately outputting the same bit as $\mathscr{A}$; observe that $\mathscr{A}'$ can efficiently evaluate the freshness conditions. Since $\mathscr{A}'$ perfectly simulates Game $\Gamma_{3a.1}$ when playing with respect to challenge bit 0 and Game $\Gamma_{3a.2}$

when it is 1, it follows that:

$$\mathsf{Adv}^{\text{g3a.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\text{g3a.2}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\text{ind-cca}}_{\mathsf{LKEM}}(\mathscr{A}') \,.$$

Game $\Gamma_{3a.3}$: This differs from Game $\Gamma_{3a.2}$ in that, for the test and partner sessions, the call to KDF made in Send and the corresponding calls made in BatchReceive with respect to ciphertext ct_ℓ output by Send are replaced with uniformly sampled keys. Let $\mathscr{A}'$ be a PRF adversary playing with respect to KDF keyed in its first argument simulating for Game $\Gamma_{3a.2}$/Game $\Gamma_{3a.3}$ adversary $\mathscr{A}$ as follows. $\mathscr{A}'$ simulates locally all calls except the Send and BatchReceive calls made in π_i^s and π_j^t, where it replaces the relevant calls $\mathsf{KDF}(K_\ell, K_k, K_s, \mathsf{sid})$ with the call $\mathsf{EVAL}(K_k, K_s, \mathsf{sid})$. Since K_ℓ is uniform (by definition of Game $\Gamma_{3a.2}$) and, by definition of freshness, K_ℓ is not revealed to $\mathscr{A}$, the simulation is perfect and we have:

$$\mathsf{Adv}^{\text{g3a.2}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\text{g3a.3}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\text{3prf}}_{\mathsf{KDF}}(\mathscr{A}') \,.$$

Finally, we have $\mathsf{Adv}^{\text{g3a.3}}_{\mathsf{DAKE},n}(\mathscr{A}) = 0$ since the output of TEST is identical regardless of the challenge bit and it is not otherwise used by the challenger or leaked to the adversary.

Game $\Gamma_{3b.1}$: We now consider the case when $E_p \wedge \neg E_{c1}$, i.e., the case where the receiver's session state st in π_i^s and π_j^t is not revealed. Game $\Gamma_{3b.1}$ differs from Game Γ_3 in that the game initially outputs π_j^t, the partner of π_i^s, as well as a bit indicating whether π_i^s is the sender or receiver. Since Game $\Gamma_{3b.1}$ is exactly Game $\Gamma_{3a.1}$, we have

$$\mathsf{Adv}^{\text{g3}}_{\mathsf{DAKE},n}(\mathscr{A}, E_p \wedge \neg E_{c1}) \leq q \cdot \mathsf{Adv}^{\text{g3b.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \,.$$

Game $\Gamma_{3b.2}$: In Game $\Gamma_{3b.2}$, the output of $\mathsf{Encaps}_{\mathsf{EKEM}}$ and the corresponding $\mathsf{Decaps}_{\mathsf{EKEM}}$ call or calls in the test session are replaced with a uniformly random key k. IND-CCA adversary $\mathscr{A}'$ simulates for Game $\Gamma_{3b.1}$/Game $\Gamma_{3b.2}$ adversary $\mathscr{A}$ as follows. $\mathscr{A}'$ follows the same broad approach as the adversary defined in the hop between Game $\Gamma_{3a.1}$ and Game $\Gamma_{3a.2}$. In particular, $\mathscr{A}'$ simulates the receiver in their session's call to Init except it uses the IND-CCA challenge public key pk, replaces the output of EKEM in the test session Encaps and the corresponding Decaps calls with the challenge ciphertext and key and replaces other Decaps calls with calls to oracle DEC. By the same reasoning as before, it follows that:

$$\mathsf{Adv}^{\text{g3b.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\text{g3b.2}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\text{ind-cca}}_{\mathsf{EKEM}}(\mathscr{A}') \,.$$

Game $\Gamma_{3b.3}$: This replaces the relevant outputs of KDF in the test session with a uniformly random key. As in Game $\Gamma_{3a.3}$, this game is now unwinnable, i.e., $\mathsf{Adv}^{\text{g3b.3}}_{\mathsf{DAKE},n}(\mathscr{A}) = 0$. As before, we reduce to the security of KDF, except now we key KDF in the PRF game with the second

argument K_k. We then arrive at:

$$\mathsf{Adv}^{\mathrm{g3b.2}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3b.3}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{3prf}}_{\mathsf{KDF}}(\mathscr{A}') \ .$$

Case 2: Test session π^s_i is unpartnered and $\pi^s_i.\mathsf{role} = \mathsf{sender}$ (Game Γ_{3c}):

Game $\Gamma_{3c.1}$: Let $\pi^s_i.\mathsf{pid} = j$ and E_s be the event that $\pi^s_i.\mathsf{role} = \mathsf{sender}$. Game $\Gamma_{3c.1}$ differs from Game Γ_3 in that the challenger immediately outputs j. By a standard argument, we have:

$$\mathsf{Adv}^{\mathrm{g3}}_{\mathsf{DAKE},n}(\mathscr{A}, \neg E_p \wedge E_s) \leq \min\{q, n\} \cdot \mathsf{Adv}^{\mathrm{g3c.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \ .$$

Game $\Gamma_{3c.2}$: This differs from Game $\Gamma_{3c.1}$ in that the challenger aborts if the call $\mathsf{Send}(\mathsf{sk}_i, \mathsf{pk}_j, \mathsf{st}_i, \mathsf{prek}_j)$ in the test session is such that prek_j was not previously output by a call to $\mathsf{Init}(\mathsf{sk}_j, \mathsf{receiver})$. Note that by freshness condition 5 that P_j must not be corrupted until after $\pi^s_i.\mathsf{status}$ is changed from $\perp$, which, by definition of E_s, means until after it is set to accept. In order for Send to accept on input $\mathsf{prek}_j = (\mathsf{espk}_j, \mathsf{ekpk}_j, \sigma_j)$ not previously output by $\mathsf{Init}(\mathsf{sk}_j, \mathsf{receiver})$ (and thus for the game to abort), $\mathscr{A}$ needs to find a different prek_j such that $\mathsf{Vrfy}(\mathsf{pk}_j.\mathsf{spk}, (\mathsf{espk}_j, \mathsf{ekpk}_j), \sigma_j)$ (by construction of Send). Using this observation, we reduce to the SUF-CMA security of Sig.

Let $\mathscr{A}'$ be a SUF-CMA adversary simulating for Game $\Gamma_{3c.1}$ adversary $\mathscr{A}$. Let pk be the SUF-CMA challenge public key. In the Setup phase, $\mathscr{A}'$ sets $\mathsf{pk}_j = \mathsf{pk}$ and otherwise simulates locally. In particular, unlike in previous hops, $\mathscr{A}'$ also samples the random Game $\Gamma_{3c.1}$ bit. In each subsequent phase, for each call $\mathsf{EXEC}(j, u, \cdot, m)$ such that $m = (\mathtt{start}, \mathsf{role}, \cdot)$, $\mathscr{A}'$ replaces the $\mathsf{Sign}_{\mathsf{Sig}}(\mathsf{sk}_j.\mathsf{ssk}, (\mathsf{espk}_j, \mathsf{ekpk}_j))$ call in $\mathsf{Init}(\mathsf{sk}_j, \mathsf{receiver})$ by a call to $\mathsf{SIGN}((\mathsf{espk}_j, \mathsf{ekpk}_j))$, and otherwise simulates the call locally. When the challenger calls $\mathsf{Send}(\cdot, \mathsf{pk}_j, \cdot, \mathsf{prek}_j)$ where $\mathsf{prek}_j = (\mathsf{espk}_j, \mathsf{ekpk}_j, \sigma_j)$, $\mathscr{A}'$ checks whether (1) $(\mathsf{espk}_j, \mathsf{ekpk}_j)$ was previously queried to SIGN which output σ_j and (2) $\mathsf{Vrfy}_{\mathsf{Sig}}(\mathsf{pk}, (\mathsf{espk}_j, \mathsf{ekpk}_j), \sigma_j) = 1$. Given (1) and (2) both hold, $\mathscr{A}'$ returns $(m, \sigma) = ((\mathsf{espk}_j, \mathsf{ekpk}_j), \sigma_j)$ to its challenger. $\mathscr{A}'$ otherwise simulates locally, aborting if $\mathscr{A}$ outputs a bit. The simulation is perfect and it follows that:

$$\mathsf{Adv}^{\mathrm{g3c.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3c.2}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{A}') \ .$$

Game $\Gamma_{3c.3}$: In Game $\Gamma_{3c.3}$, the challenger initially outputs π^t_j, where π^t_j is the session that prek is output by $\mathsf{Init}(\mathsf{sk}_j, \mathsf{receiver})$ and input to the Send call in test session π^s_i. By a standard failure event argument, we have:

$$\mathsf{Adv}^{\mathrm{g3c.2}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq q \cdot \mathsf{Adv}^{\mathrm{g3c.3}}_{\mathsf{DAKE},n}(\mathscr{A}) \ .$$

Game $\Gamma_{3c.4a.1}$: Let E_{c2} be the event that P_j is corrupted. We construct hybrid sequence Game $\Gamma_{3c.4a}$ (resp. Game $\Gamma_{3c.4b}$) to deal with the case that E_{c2} holds (resp. does not hold).

Game $\Gamma_{3c.4a.1}$ differs from Game $\Gamma_{3c.3}$ in that the output of $\mathsf{Encaps}_{\mathsf{EKEM}}$ in the Send call in test session π_i^s and of the (possible) corresponding $\mathsf{Decaps}_{\mathsf{EKEM}}$ calls in π_j^t are replaced with uniformly random output. By freshness, P_j's session state π_j^t.st associated with prek input to the test Send call is not revealed.

IND-CCA adversary $\mathscr{A}'$ simulates for Game $\Gamma_{3c.4a.1}$ adversary $\mathscr{A}$ as follows. Let $(\mathsf{pk}, k, \mathsf{ct})$ the challenge public key, key and corresponding ciphertext (respectively) of $\mathscr{A}'$. $\mathscr{A}'$ embeds pk in session π_j^t by replacing the public key output by $\mathsf{KeyGen}_{\mathsf{EKEM}}$ in $\mathsf{Init}(\mathsf{sk}_j, \mathsf{receiver})$ with pk, which outputs prek_j. Upon prek_j being input to Send in the test session, $\mathscr{A}'$ replaces the output of $\mathsf{Encaps}_{\mathsf{EKEM}}$ with (k, ct). When the challenger calls $\mathsf{BatchReceive}(\mathsf{sk}_j, \cdot, \{\cdot, \cdot, m_{j'} = (\cdot, \mathsf{ct}', \cdot)\}_{j'})$ in session π_j^t, if $\mathsf{ct}' = \mathsf{ct}$, $\mathscr{A}'$ replaces the output of $\mathsf{Decaps}_{\mathsf{EKEM}}$ with k; else, $\mathscr{A}'$ replaces the call $\mathsf{Decaps}_{\mathsf{EKEM}}(\cdot, \mathsf{ct}')$ with the call $\mathsf{DEC}(\mathsf{ct}')$. $\mathscr{A}'$ otherwise simulates locally and outputs the same bit as $\mathscr{A}$. By similar reasons to before, we have:

$$\mathsf{Adv}^{\mathrm{g3c.3}}_{\mathsf{DAKE},n}(\mathscr{A}, E_{c2}) \leq \mathsf{Adv}^{\mathrm{g3c.4a.1}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{ind-cca}}_{\mathsf{EKEM}}(\mathscr{A}') \,.$$

Game $\Gamma_{3c.4a.2}$: This replaces the output of KDF in the Send and BatchReceive calls as before in the test session and π_j^t with uniformly random keys. Otherwise, by the exact same argument as for Game $\Gamma_{3b.3}$ (where since the two sessions are unpartnered, they must diverge in sid, and so both EVAL queries are allowed in the simulation), we have $\mathsf{Adv}^{\mathrm{g3c.4a.2}}_{\mathsf{DAKE},n}(\mathscr{A}) = 0$ and

$$\mathsf{Adv}^{\mathrm{g3c.4a.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3c.4a.2}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{3prf}}_{\mathsf{KDF}}(\mathscr{A}') \,.$$

Game $\Gamma_{3c.4b.1}$: We assume $\neg E_{c2}$, i.e., that P_j is not corrupted. We reduce to the IND-CCA security of LKEM. The reduction follows the same high-level strategy as previous hops (embedding the challenge pk in pk_j and the challenge in the test Send call and possibly the corresponding BatchReceive call), noting that non-challenge $\mathsf{Decaps}_{\mathsf{LKEM}}(\mathsf{sk}_j, \cdot)$ queries are replaced with calls to DEC. We then have:

$$\mathsf{Adv}^{\mathrm{g3c.3}}_{\mathsf{DAKE},n}(\mathscr{A}, \neg E_{c2}) \leq \mathsf{Adv}^{\mathrm{g3c.4b.1}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{ind-cca}}_{\mathsf{LKEM}}(\mathscr{A}') \,.$$

Game $\Gamma_{3c.4b.2}$: As in Game $\Gamma_{3c.4a.2}$, this replaces the output of KDF in the Send and BatchReceive calls in π_i^s and π_j^t with a uniformly random key. As argued several times above, it follows that $\mathsf{Adv}^{\mathrm{g3c.4b.2}}_{\mathsf{DAKE},n}(\mathscr{A}) = 0$ and

$$\mathsf{Adv}^{\mathrm{g3c.4b.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3c.4b.2}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{3prf}}_{\mathsf{KDF}}(\mathscr{A}') \,.$$

Case 3: Test session π_i^s is unpartnered and π_i^s.role = receiver (Game Γ_{3d}):

Game $\Gamma_{3d.1}$: Game $\Gamma_{3d.1}$ differs from Game Γ_3 in that the challenger immediately out-

puts j, the third argument in $\mathscr{A}$'s $\mathsf{TEST}(i,s,j)$ call. As for Game $\Gamma_{3c.1}$, we have

$$\mathsf{Adv}^{\mathrm{g3}}_{\mathsf{DAKE},n}(\mathscr{A},\neg E_s) \leq \min\{q,n\}\cdot \mathsf{Adv}^{\mathrm{g3d.1}}_{\mathsf{DAKE},n}(\mathscr{A}) .$$

Game $\Gamma_{3d.2}$: This differs from Game $\Gamma_{3d.1}$ in that the challenger aborts if the call $\mathsf{BatchReceive}(\mathsf{sk}_i,\mathsf{st}_i,\{\mathsf{pk}_{j'},\mathsf{prek}_{j'},m\}_{j'})$ in the test session is such that prek_j was not previously output by a call to $\mathsf{Init}(\mathsf{sk}_j,\mathsf{sender})$. As in Game $\Gamma_{3c.2}$, P_j must not be corrupted until after π_i^s.status is set to accept. By reducing to SUF-CMA security essentially as in Game $\Gamma_{3c.2}$, it follows that:

$$\mathsf{Adv}^{\mathrm{g3d.1}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3d.2}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{A}) .$$

Game $\Gamma_{3d.3}$: This differs from Game $\Gamma_{3d.2}$ in that the challenger initially outputs π_j^t, the session that generated prek_j which formed part of the input to $\mathsf{BatchReceive}$ in the test session π_i^s. By a standard argument we have:

$$\mathsf{Adv}^{\mathrm{g3d.2}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq q\cdot \mathsf{Adv}^{\mathrm{g3d.3}}_{\mathsf{DAKE},n}(\mathscr{A}) .$$

Game $\Gamma_{3d.4}$: This differs from Game $\Gamma_{3d.3}$ in that the challenger aborts if the $\mathsf{Send}(\mathsf{sk}_j,\mathsf{pk}_i,\mathsf{st}_j,\mathsf{prek}_i)$ call in session π_j^t (if it exists) and the relevant component in the $\mathsf{BatchReceive}$ call in the test session π_i^s were not both with respect to honestly generated split-KEM keying material (namely, an honestly generated split-KEM public key from prek_i and prek_j from the previous hop) and the same split-KEM ciphertext. By freshness, neither of the two ephemeral states π_i^s.st and π_j^t.st are revealed. Consequently, we reduce to the UNF-1KCA security of split-KEM sKEM.

UNF-1KCA adversary $\mathscr{A}'$ simulates for Game $\Gamma_{3d.3}$/Game $\Gamma_{3d.4}$ adversary $\mathscr{A}$ as follows. Let $(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B})$ be the two challenge public keys given to $\mathscr{A}'$. In the $\mathsf{Init}(\mathsf{sk}_i,\mathsf{receiver})$ call in session π_i^s, $\mathscr{A}'$ simulates except replaces the call to $\mathsf{KeyGenA}_{\mathsf{sKEM}}$ by pk_A. Similarly, in the $\mathsf{Init}(\mathsf{sk}_j,\mathsf{sender})$ call in session π_j^t, $\mathscr{A}'$ replaces $\mathsf{KeyGenB}_{\mathsf{sKEM}}$ by pk_B. In the $\mathsf{Send}(\ldots,\mathsf{prek})$ call in session π_j^t where $\mathsf{prek}=(\mathsf{espk},\ldots)$, $\mathscr{A}'$ outputs espk to its UNF-1KCA challenger, receives $(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B},\mathsf{ct},K_B)$ from its challenger, and replaces the call to $\mathsf{Encaps}_{\mathsf{sKEM}}$ with tuple (ct,K_B). Finally, when the $\mathsf{BatchReceive}(\mathsf{sk}_i,\cdot,\{\cdot,\mathsf{prek}_{j'},m=(\cdot,\cdot,\mathsf{ct}_s)\}_{j'})$ call in test session π_i^s is made, $\mathscr{A}'$ outputs ct_s corresponding to $j'=j$ to its challenger. As the simulation is perfect and the probability that $\mathscr{A}'$ wins is exactly the probability that 1) $(\mathsf{ct},\mathsf{pk}_\mathsf{A})\neq(\mathsf{ct}_s,\mathsf{pk})$ and 2) relevant $\mathsf{Decaps}_{\mathsf{sKEM}}$ call in $\mathsf{BatchReceive}$ outputs $k\neq\perp$, it follows by a standard failure event argument that:

$$\mathsf{Adv}^{\mathrm{g3d.3}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3d.4}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{unf-1kca}}_{\mathsf{sKEM}}(\mathscr{A}') .$$

Game $\Gamma_{3d.5}$: This differs from Game $\Gamma_{3d.4}$ in that the output k of the relevant test session split-KEM decapsulation and the corresponding encapsulation (if it exists) are both replaced by a uniformly random key. Note that by definition of Game $\Gamma_{3d.4}$, $\mathscr{A}$ can only input an honestly generated split-KEM ciphertext to the $\mathsf{BatchReceive}$ call in the test session from P_j

and that the split-KEM public key in P_j's corresponding Send call (if it exists) must be honestly generated. We therefore reduce to the IND-1BatchCCA security of sKEM. We embed the IND-1BatchCCA keys pk_A, pk_B in the simulation as in the previous hop. When $\mathscr{A}$ queries $\mathsf{EXEC}(i, s, S = \{s_{j'}, \mathsf{prek}_{j'}, m_{j'}\}_{j'})$, $\mathscr{A}'$ replaces all $\mathsf{Decaps}_{\mathsf{sKEM}}$ calls involving sk_A *except* the call corresponding to the test session by the output of its query to oracle BatchDEC, replaces this final $\mathsf{Decaps}_{\mathsf{sKEM}}$ call with the IND-1BatchCCA challenge key and otherwise simulates locally. It follows that:

$$\mathsf{Adv}^{\mathrm{g3d.4}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3d.5}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{ind-1batchcca}}_{\mathsf{sKEM}}(\mathscr{A}') \,.$$

Game $\Gamma_{3d.6}$: This game replaces the relevant invocation of KDF in the test session's BatchReceive call by a uniformly random value. Note as usual that $\mathsf{Adv}^{\mathrm{g3d.6}}_{\mathsf{DAKE},n}(\mathscr{A}) = 0$. By keying KDF in its third argument as a PRF and a standard argument it follows that:

$$\mathsf{Adv}^{\mathrm{g3d.5}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3d.6}}_{\mathsf{DAKE},n}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{3prf}}_{\mathsf{KDF}}(\mathscr{A}') \,.$$

Finally note that by the triangle inequality, we have, among other inequalities:

$$\mathsf{Adv}^{\mathrm{g3}}_{\mathsf{DAKE},n}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g3}}_{\mathsf{DAKE},n}(\mathscr{A}, E_p) + \mathsf{Adv}^{\mathrm{g3}}_{\mathsf{DAKE},n}(\mathscr{A}, \neg E_p) \,.$$

The result follows using this observation and by combining the sequences of hybrids together in a standard way. □

We now prove the deniability of K-Waay.

Theorem 2. Consider deniable split-KEM sKEM with simulator $\mathsf{Sim}_{\mathsf{sKEM}}$ used to build K-Waay (Figure 3.7). Then, we have that for every efficient adversary $\mathscr{A}$ that makes at most q oracle queries, there exists an efficient Sim s.t. one can build an adversary $\mathscr{B}$ such that for $\mathsf{exp} \in \{\mathsf{true}, \mathsf{false}\}$ we have:

$$\mathsf{Adv}^{\mathrm{deny}}_{\mathsf{K\text{-}Waay},\mathsf{Sim},\mathsf{exp}}(\mathscr{A}) \leq q \cdot \mathsf{Adv}^{\mathrm{deny}}_{\mathsf{sKEM},\mathsf{Sim}_{\mathsf{sKEM}}}(\mathscr{B}) \,.$$

Proof. We construct a sequence of hybrids and reduce to the deniability of sKEM (i.e., $\mathsf{DENY}_{\mathsf{sKEM},\mathsf{Sim}_{\mathsf{sKEM}}}$ security) in each step. Before this, we define the simulator Sim that we use in the proof, which uses the simulator $\mathsf{Sim}_{\mathsf{sKEM}}$ (which must exist if sKEM is deniable) as a subroutine.

Observe in K-Waay that, given an honestly generated prek_j, any party with knowledge only of public keying material can simulate all steps in Send except for the $\mathsf{Encaps}_{\mathsf{sKEM}}$ call which requires sender P_i's secret key. Thus, our simulator Sim (Figure 3.8) simulates these steps and since it takes the receiver's key sk_j as input it can also invoke the deniability simulator $\mathsf{Sim}_{\mathsf{sKEM}}$ to complete the call.

$\mathsf{Sim}(\mathsf{sk}_j, \mathsf{pk}_i, \mathsf{st}_j, \mathsf{prek}_i, \mathsf{prek}_j)$

1: $(\mathsf{essk}_j, \mathsf{eksk}_j, \mathsf{prek}_j) \leftarrow \mathsf{st}_j$
2: $(\mathsf{espk}_i, \mathsf{ekpk}_i, \sigma_i) \leftarrow \mathsf{prek}_i$
3: $(\mathsf{espk}_j, \mathsf{ekpk}_j, \sigma_j) \leftarrow \mathsf{prek}_j$
4: $\mathsf{spk} \leftarrow \mathsf{GetPK}(\mathsf{sk}_j.\mathsf{ssk})$
5: **require** $\mathsf{Vrfy}_{\mathsf{Sig}}(\mathsf{spk}, (\mathsf{espk}_j, \mathsf{ekpk}_j), \sigma_j) = 1$
6: $(K_\ell, \mathsf{ct}_\ell) \xleftarrow{\$} \mathsf{Encaps}_{\mathsf{LKEM}}(\mathsf{pk}_j.\mathsf{kpk})$
7: $(K_k, \mathsf{ct}_k) \xleftarrow{\$} \mathsf{Encaps}_{\mathsf{EKEM}}(\mathsf{ekpk}_j)$
8: $(K_s, \mathsf{ct}_s) \xleftarrow{\$} \mathsf{Sim}_{\mathsf{sKEM}}(\mathsf{espk}_i, \mathsf{essk}_j)$
9: $m \leftarrow (\mathsf{ct}_\ell, \mathsf{ct}_k, \mathsf{ct}_s)$
10: $\mathsf{sid} \leftarrow P_i || P_j || \mathsf{pk}_i || \mathsf{pk}_j || \mathsf{prek}_i || \mathsf{prek}_j || m$
11: $\mathsf{k} \leftarrow \mathsf{KDF}(K_\ell, K_k, K_s, \mathsf{sid})$
12: **return** (k, m)

Figure 3.8: Simulator Sim for the deniability game where we assume function GetPK(sk) that takes a signature secret key as input and outputs the corresponding public key.

Let Γ_0 be the DAKE DENY game instantiated with K-Waay. For $i \in [q]$, let Γ_i be the same as Γ_{i-1} except that in the i-th CHAL call, the call to Send is replaced with a call to Sim. Note that the steps executed in Send only differ in that it calls $\mathsf{Encaps}_{\mathsf{sKEM}}$ rather than $\mathsf{Sim}_{\mathsf{sKEM}}$.

For $i \in [q]$, let $\mathscr{B}$ be a split-KEM DENY adversary with input $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{B}, K, \mathsf{ct})$ from its challenger playing $\mathsf{DENY}^{\mathsf{REAL}}$ given $\mathscr{A}$ is playing Γ_{i-1} and $\mathsf{DENY}^{\mathsf{SIM}}$ if it is playing Γ_i. $\mathscr{B}'$ locally simulates long-term public key generation and the first $i-1$ calls to CHAL. When $\mathscr{A}$ makes their i-th call to CHAL, $\mathscr{B}$ simulates CHAL until it reaches the **if** statement except that it replaces the output of calls KeyGenA/KeyGenB calls in Init calls with $\mathsf{pk}_\mathsf{A}/\mathsf{pk}_\mathsf{B}$. Then, instead of executing the **if**/**else** block in CHAL, $\mathscr{B}$ simulates Sim except that it replaces the output of the call to $\mathsf{Sim}_{\mathsf{sKEM}}$ with (K, ct). $\mathscr{B}$ then simulates locally, and returns (k, T, st_r) (where st_r contains sk_B) if $exp = \mathsf{true}$ and returns (k, T) otherwise. $\mathscr{B}$ continues simulating locally and finally outputs the same bit as $\mathscr{A}$. Noting that DAKE deniability game $\mathsf{DENY}_{\mathsf{K\text{-}Waay},\mathsf{Sim}}$ considers only honest executions of K-Waay, it follows that the simulation is perfect, and so by $\mathsf{DENY}_{\mathsf{sKEM}}$ security we have

$$\left|\mathsf{Adv}^{\Gamma_{i-1}}_{\mathsf{DAKE}}(\mathscr{A}) - \mathsf{Adv}^{\Gamma_i}_{\mathsf{DAKE}}(\mathscr{A})\right| \leq \mathsf{Adv}^{\mathrm{deny}}_{\mathsf{sKEM},\mathsf{Sim}_{\mathsf{sKEM}}}(\mathscr{B}) .$$

By application of the triangle inequality and telescoping sums:

$$\left|\mathsf{Adv}^{\Gamma_0}_{\mathsf{DAKE}}(\mathscr{A}) - \mathsf{Adv}^{\Gamma_q}_{\mathsf{DAKE}}(\mathscr{A})\right| \leq q \cdot \mathsf{Adv}^{\mathrm{deny}}_{\mathsf{sKEM},\mathsf{Sim}_{\mathsf{sKEM}}}(\mathscr{B}) .$$

To complete the proof, observe that $\mathsf{Adv}^{\Gamma_q}_{\mathsf{DAKE},n}(\mathscr{A}) = 0$ since CHAL behaves identically independent of challenge bit b. □

3.5 Deniable Split-KEM from Lattices

In this section we build an efficient deniable split-KEM under the hardness of LWE. We start by introducing briefly several concepts of lattice-based cryptography that we use to design the scheme. We then present our split-KEM and prove that it is correct, OW-CPA, deniable and decaps-CPA. After this, we provide a generic transformation in the (Q)ROM to build UNF-1KCA and IND-1BatchCCA split-KEM assuming decaps-CPA and OW-CPA security respectively. Finally, we provide concrete parameters for our split-KEM that provide at least 128 bits of classical and quantum security.

3.5.1 Lattice Toolbox

L_∞ and L_α norms. We start by recalling what the L_∞ and L_α norms over $\mathbb{Z}_q$ are. For an element w in $\mathbb{Z}_q$, we write $\|w\|_\infty$ to mean $|\langle w \rangle_q|$ (i.e., $|w \pmod q|$). Then, we define the L_∞ and L_α norms for $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ over $\mathbb{Z}_q$ as follows:

$$\|\mathbf{w}\|_\infty = \max_{j \in [n]} \|w_j\|_\infty, \quad \|\mathbf{w}\|_\alpha = \sqrt[\alpha]{\|w_1\|_\infty^\alpha + \ldots + \|w_n\|_\infty^\alpha}.$$

By default, $\|\mathbf{w}\| := \|\mathbf{w}\|_2$.

Probability Distributions. For a finite set S, we define $\mathscr{U}(S)$ to be the uniform distribution on S. We will also use the binomial distribution Bin_1 which is defined as: $\mathsf{Bin}_1(-1) = \mathsf{Bin}_1(1) = 1/4$ and $\mathsf{Bin}_1(0) = 1/2$.

Rounding Functions. Given two parameters q and $B < \log q - 1$, we define the rounding function $\lfloor \cdot \rceil_{q,2}$ and the cross-rounding function $\langle \cdot \rangle_{q,B}$ as follows:

$$\lfloor v \rceil_{q,B} := \left\lfloor \frac{2^B}{q} \cdot v \right\rceil \bmod 2^B, \quad \langle v \rangle_{q,B} := \left\lfloor \frac{2^{B+1}}{q} \cdot v \right\rfloor \bmod 2\,,$$

for any $v \in \mathbb{Z}_q$.

Reconciliation Function. We recall the (generalised) reconciliation mechanism from Bos et al. and Peikert [BCD+16, Pei14], which for every approximate agreement in $\mathbb{Z}_q$ allows extracting shared bits. We refer the reader to the aforementioned works for more details. Let q be a positive integer. Let B be the number of bits we want to extract from one coefficient in $\mathbb{Z}_q$ so that $B < \log q - 1$. Now, for any $v \in \mathbb{Z}_q$, which is represented as an integer in $[0, q)$, we define the following functions.

Definition 28 (Randomised doubling function (dbl))**.** For any $v \in \mathbb{Z}_q$, we define $\mathsf{dbl}(\cdot)$ as follows:

$$\mathsf{dbl}(v) : v \mapsto 2v - e, \quad e \xleftarrow{\$} \mathsf{Bin}_1$$

Then, we have the following property which comes from [BCD+16, Claim 3.1].

Lemma 1. Let q be odd. If $v \in \mathbb{Z}_q$ is uniformly random and $\bar{v} \xleftarrow{\$} \mathsf{dbl}(v) \in \mathbb{Z}_{2q}$, then $\lfloor \bar{v} \rceil_{2q,B}$ is uniformly random given $\langle \bar{v} \rangle_{2q,B}$.

Now, we are ready to define the reconciliation function $\mathsf{rec} : \mathbb{Z}_{2q} \times \mathbb{Z}_2 \to \mathbb{Z}_{2^B}$.

Definition 29 (Reconciliation function (rec))**.** For any $w \in \mathbb{Z}_{2q}$ and bit $b \in \{0,1\}$, let v be the closest element to $w \in \mathbb{Z}_{2q}$ s.t. $\langle v \rangle_{2q,B} = b$. Then, we define rec as

$$\mathsf{rec}(w, b) := \lfloor v \rceil_{2q,B} .$$

The next result gives an important property of the reconciliation function rec, as described by Peikert [Pei14, Section 3.2].

Lemma 2. Let q be odd and $\bar{v} \xleftarrow{\$} \mathsf{dbl}(v)$. If $|v - w| \leq \lfloor \frac{q}{2^{B+2}} \rfloor$ then

$$\mathsf{rec}(2w, \langle \bar{v} \rangle_{2q,B}) = \lfloor \bar{v} \rceil_{2q,B}.$$

Finally, we define the $\mathsf{HelpRec} : \mathbb{Z}_q \mapsto \{0,1\}$ function as follows:

Definition 30 (HelpRec function)**.** On any input $v \in \mathbb{Z}_q$,

$$\mathsf{HelpRec}(v) := \langle \bar{v} \rangle_{2q,B}, \quad \text{where } \bar{v} \leftarrow \mathsf{dbl}(v) .$$

All the functions above can be naturally generalised to take as input vectors and matrices over $\mathbb{Z}_q$ by applying the function to each of the coefficients.

Learning-with-Errors. The security of our lattice constructions relies on the learning-with-errors (LWE) problem introduced by Regev [Reg05]. In this chapter we will consider the case where both the secret and error coefficients come from a probability distribution over $\mathbb{Z}$.

Definition 31 ($\mathsf{LWE}_{n,m,\chi,q}$)**.** Let $n, m \in \mathbb{N}$ and χ be a probability distribution over $\mathbb{Z}$. The LWE problem asks the adversary $\mathscr{A}$ to distinguish between the following two cases:

1. $(\mathbf{A}, \mathbf{As} + \mathbf{e} \bmod q)$ for $\mathbf{A} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n \times m})$, a secret $\mathbf{s} \xleftarrow{\$} \chi^m$ and error $\mathbf{e} \xleftarrow{\$} \chi^n$,
2. $(\mathbf{A}, \mathbf{t}) \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n \times m}) \times \mathscr{U}(\mathbb{Z}_q^n)$.

We say that the $\mathsf{LWE}_{n,m,\chi,q}$ assumption holds if any efficient adversary cannot distinguish between the two distributions except with negligible probability.

3.5.2 Extended-LWE

Our proof of deniability for the split-KEM will involve a new security assumption, which we call the Extended-LWE problem (ELWE). Intuitively, it is similar to the plain LWE problem, but the adversary is now also given random linear combinations of the secrets and errors.

Definition 32 ($\mathsf{ELWE}_{n,m,\bar{n},\chi,q}$)**.** Let $n, m \in \mathbb{N}$ and χ be a probability distribution over $\mathbb{Z}$. The ELWE problem asks the adversary $\mathscr{A}$ to distinguish between the following two cases:

1. $\left(\mathbf{A}, \mathbf{As}+\mathbf{e} \bmod q, \mathbf{Z}, \mathbf{W}, \mathbf{Zs}+\mathbf{We} \bmod q\right)$ for $\mathbf{A} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n\times m})$, secret $\mathbf{s} \xleftarrow{\$} \chi^m$, error $\mathbf{e} \xleftarrow{\$} \chi^n$, and $(\mathbf{Z}, \mathbf{W}) \xleftarrow{\$} \chi^{\bar{n}\times m} \times \chi^{\bar{n}\times n}$,
2. $\left(\mathbf{A}, \mathbf{t}, \mathbf{Z}, \mathbf{W}, \mathbf{Zs}+\mathbf{We} \bmod q\right)$ for $\mathbf{A} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n\times m})$, $\mathbf{t} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n})$, secret $\mathbf{s} \xleftarrow{\$} \chi^m$, error $\mathbf{e} \xleftarrow{\$} \chi^n$, and $(\mathbf{Z}, \mathbf{W}) \xleftarrow{\$} \chi^{\bar{n}\times m} \times \chi^{\bar{n}\times n}$.

We say that the $\mathsf{ELWE}_{n,m,\bar{n},k,\chi,q}$ assumption holds if any efficient adversary cannot distinguish between the two distributions except with negligible probability.

This problem is a natural generalisation of the Extended-LWE problem by Alperin-Sheriff and Peikert [AP12], where now $(\mathbf{Z}, \mathbf{W})$ are matrices and not just vectors. Here, we also simplify the definition and assume that the coefficients of $\mathbf{Z}$ and $\mathbf{W}$ come from the same distribution χ as the secrets and errors.

We show in the following theorem that the hardness of this newly introduced ELWE problem reduces to the hardness of LWE.

Theorem 3. Let q be an odd prime and χ be symmetric around 0. If there is an efficient adversary $\mathscr{A}$ which wins $\mathsf{ELWE}_{n,m,\bar{n},\chi,q}$ with probability ε, then there also exists an efficient adversary $\mathscr{B}$ which wins $\mathsf{LWE}_{n+m,m,\chi,q}$ with probability at least $\delta_{\mathsf{elwe}} \cdot \varepsilon - \mathsf{negl}(n)$ where

$$\delta_{\mathsf{elwe}} := \Pr\left[\mathbf{Z}(\mathbf{e}-\mathbf{d}) = \mathbf{0} \pmod q : \mathbf{Z} \xleftarrow{\$} \chi^{\bar{n}\times(n+m)}, \mathbf{e}, \mathbf{d} \xleftarrow{\$} \chi^{n+m}\right]. \tag{3.1}$$

Proof. We prove the statement by introducing a sequence of LWE-type games Γ_i. We start with $\Gamma^1 := \mathsf{ELWE}_{n,m,\bar{n},\chi,q}$ and give an efficient reduction from Γ_i to Γ_{i+1}. In the end, we finish with $\mathsf{LWE}_{n+m,m,\chi,q}$. We denote $\mathsf{Adv}_i(\mathscr{A})$ to be the probability that $\mathscr{A}$ wins Γ_i. Then, the proof follows by the composition of the constant number of efficient reductions.

Game Γ_1: This is the standard $\mathsf{ELWE}_{n,m,\bar{n},\chi,q}$ game. The adversary $\mathscr{A}$ wins this game with probability ε.

Game Γ_2: Here, we consider the ELWE-type game where the secret vector is uniformly random. Namely, the challenger samples the public $\mathbf{A} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{(n+m)\times m})$, secret $\mathbf{s} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{m})$, error $\mathbf{e} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n+m})$ as well as the hint matrix $\mathbf{Z} \xleftarrow{\$} \chi^{\bar{n}\times(n+m)}$. Then it flips a bit $b \xleftarrow{\$} \mathscr{U}(\{0,1\})$. If $b = 0$ then the challenger computes

$$\mathbf{t} := \mathbf{As}+\mathbf{e}$$

and otherwise it samples $\mathbf{t} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n+m})$. The challenger outputs $(\mathbf{A}, \mathbf{t}, \mathbf{Z}, \mathbf{Ze})$.

Lemma 3. For every efficient adversary $\mathscr{A}$, there is an efficient adversary $\mathscr{B}$ such that $\mathsf{Adv}_2(\mathscr{B}) \geq \mathsf{Adv}_1(\mathscr{A}) - \mathsf{negl}(n)$.

Proof. The reduction follows similarly as in the one by Applebaum et al. [ACPS09]. Suppose the algorithm $\mathscr{B}$ is given a tuple $(\mathbf{A}, \mathbf{t}, \mathbf{Z}, \mathbf{h})$ from Γ_2. With probability at most $1/q^{(n+m)-m-1} \leq 1/q^{n-1}$, matrix $\mathbf{A}$ is not full-rank. Let us exclude that case and assume without loss of generality that we can write

$$\mathbf{A} := \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \end{bmatrix}, \quad \mathbf{Z} := \begin{bmatrix} \mathbf{Z}_0 & \mathbf{Z}_1 \end{bmatrix}, \quad \text{and} \quad \mathbf{t} := \begin{bmatrix} \mathbf{t}_0 \\ \mathbf{t}_1 \end{bmatrix}$$

where $\mathbf{A}_1 \in \mathbb{Z}_q^{n \times m}$ and the matrix $\mathbf{A}_0 \in \mathbb{Z}_q^{m \times m}$, which contains the first m rows of $\mathbf{A}$, is invertible. Thus, define $\mathbf{A}' := \mathbf{A}_1 \mathbf{A}_0^{-1} \in \mathbb{Z}_q^{n \times m}$, and $\mathbf{t}' := \mathbf{A}'\mathbf{t}_0 - \mathbf{t}_1 \in \mathbb{Z}_q^n$. Then, it runs $\mathscr{A}$ on input

$$\left(\mathbf{A}', \mathbf{t}', \mathbf{Z}_0, -\mathbf{Z}_1, \mathbf{h}\right)$$

and returns what $\mathscr{A}$ outputs.

Suppose that $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$ where $\mathbf{s} \in \mathbb{Z}_q^m$ and $\mathbf{e} := (\mathbf{e}_0, \mathbf{e}_1) \in \mathbb{Z}_q^m \times \mathbb{Z}_q^n$. Then

$$\mathbf{t}' = \mathbf{A}'\mathbf{t}_0 - \mathbf{t}_1 = \mathbf{A}_1\mathbf{A}_0^{-1}(\mathbf{A}_0\mathbf{s} + \mathbf{e}_0) - (\mathbf{A}_1\mathbf{s} + \mathbf{e}_1) = \mathbf{A}'\mathbf{e}_0 - \mathbf{e}_1$$

which is a valid LWE instance since χ is symmetric around 0. Also, if $\mathbf{A}$ is uniformly random among all nonsingular matrices, then $\mathbf{A}'$ and $\mathbf{B}'$ are statistically close to uniformly random matrices over $\mathbb{Z}_q$. As for the hints, note that

$$\mathbf{h} = \mathbf{Z}_0\mathbf{e}_0 + \mathbf{Z}_1\mathbf{e}_1 = \mathbf{Z}_0\mathbf{e}_0 + (-\mathbf{Z}_1)(-\mathbf{e}_1),$$

so $\mathbf{h}$ is a well-formed hint for Γ_1.

On the other hand, if $\mathbf{t}$ is uniformly random, then so is $\mathbf{t}'$. It can be argued similarly as before that all the other components follow the distribution for $b = 1$. □

Game Γ_3: We consider the knapsack version of ELWE. Here, the challenger samples the public $\mathbf{G} :\xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n \times (n+m)})$, secret $\mathbf{e} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n+m})$ and the hint matrix $\mathbf{Z} \xleftarrow{\$} \chi^{\bar{n} \times (n+m)}$. Then it flips a bit $b \xleftarrow{\$} \mathscr{U}(\{0,1\})$. If $b = 0$ then the challenger computes $\mathbf{t} := \mathbf{G}\mathbf{e}$, and otherwise it samples $\mathbf{t} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^n)$. Finally, the challenger outputs $(\mathbf{G}, \mathbf{t}, \mathbf{Z}, \mathbf{Z}\mathbf{e})$.

Lemma 4. For every efficient adversary $\mathscr{A}$, there is an efficient adversary $\mathscr{B}$ such that $\mathsf{Adv}_3(\mathscr{B}) \geq \mathsf{Adv}_2(\mathscr{A}) - \mathsf{negl}(n)$.

Proof. The reduction is similar to the proof of Micciancio and Mol [MM11, Lemma 4.9]. Suppose the algorithm $\mathscr{B}$ is given a tuple $(\mathbf{G}, \mathbf{t}, \mathbf{Z}, \mathbf{h})$ from Γ_3. Then, $\mathscr{B}$ can construct a randomised matrix $\mathbf{A} \in \mathbb{Z}_q^{(n+m) \times m}$ whose columns generate the kernel of $\mathbf{G}$. In particular, if $\mathbf{G}$ is uniformly random, then so are $(\mathbf{A}, \mathbf{B})$, up to the constraint that they are nonsingular. Then, $\mathscr{B}$ computes any solution $\mathbf{r}$ such that $\mathbf{G}\mathbf{r} = \mathbf{t}$. Finally, it samples a uniformly random $\mathbf{s} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^m)$ and runs $\mathscr{A}$ on input

$$(\mathbf{A}, \mathbf{A}\mathbf{s} + \mathbf{r}, \mathbf{Z}, \mathbf{h})$$

and returns what $\mathscr{A}$ outputs.

Suppose that $\mathbf{Ge} = \mathbf{t} = \mathbf{Gr}$. By definition of the matrix $\mathbf{A}$, $\mathbf{G}(\mathbf{r}-\mathbf{e}) = \mathbf{0}$ implies that there exists some vector $\mathbf{x} \in \mathbb{Z}_q^m$ such that $\mathbf{r}-\mathbf{e} = \mathbf{Ax}$. Thus,

$$\mathbf{As}+\mathbf{r} = \mathbf{A}(\mathbf{s}+\mathbf{x})+\mathbf{e}$$

which is a valid LWE instance since $\mathbf{s}+\mathbf{x}$ is still uniformly random over $\mathbb{Z}_q^m$. As for the hints, we still have $\mathbf{h} = \mathbf{Ze}$ and thus $\mathscr{B}$ correctly simulates Γ_2 for $b = 0$. The case $b = 1$ follows by arguing that $\mathbf{t}$ is uniformly random and if $\mathbf{G}$ is nonsingular then $\mathbf{r}$ must be uniformly random. □

Game Γ_4: This game is a plain knapsack LWE problem. The challenger samples the public $\mathbf{G} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n\times(n+m)})$ and a secret $\mathbf{e} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n+m})$. Then it flips a bit $b \xleftarrow{\$} \mathscr{U}(\{0,1\})$. If $b = 0$ then the challenger computes $\mathbf{t} := \mathbf{Ge}$, and otherwise it samples $\mathbf{t} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^n)$. Finally, the challenger outputs $(\mathbf{G}, \mathbf{t})$.

Lemma 5. For every efficient adversary $\mathscr{A}$, there is an efficient adversary $\mathscr{B}$ such that $\mathsf{Adv}_4(\mathscr{B}) \geq \delta_{\mathsf{elwe}} \cdot \mathsf{Adv}_3(\mathscr{A})$.

Proof. We follow the proof strategy from [AP12, Theorem 1]. Suppose the algorithm $\mathscr{B}$ is given a tuple $(\mathbf{G}_0, \mathbf{G}_1, \mathbf{t})$ from Γ_4. Then, it samples $\mathbf{Z} \xleftarrow{\$} \chi^{\bar{n}\times(n+m)}$, $\mathbf{d} \xleftarrow{\$} \chi^{n+m}$ and a matrix $\mathbf{V} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n\times\bar{n}})$. Further, it sets

$$\mathbf{G}' := \mathbf{G}-\mathbf{VZ} \quad \text{and} \quad \mathbf{t}' := \mathbf{t}-\mathbf{VZd}.$$

Finally, it runs $\mathscr{A}$ on input

$$(\mathbf{G}', \mathbf{t}', \mathbf{Z}, \mathbf{Zd})$$

and returns what $\mathscr{A}$ outputs.

Clearly, if $\mathbf{G}$ (resp. $\mathbf{t}$) is uniformly random then so is $\mathbf{G}'$ (resp. $\mathbf{t}'$). Hence, the case $b = 1$ follows directly. Suppose $b = 0$ and thus $\mathbf{t} = \mathbf{Ge}$. Then, we have

$$\mathbf{t}' = \mathbf{t}-\mathbf{VZd} = \mathbf{Ge}-\mathbf{VZd} = \mathbf{G}'\mathbf{e}+\mathbf{V}(\mathbf{Ze}-\mathbf{Zd}).$$

Hence, if $\mathbf{Ze} = \mathbf{Zd}$ then $([\mathbf{G}_0' \ \mathbf{G}_1], \mathbf{t}', \mathbf{Z}, \mathbf{Zd})$ is indeed a valid knapsack ELWE tuple. This happens exactly with probability at most δ_{elwe} by definition. Otherwise, $\mathbf{V}(\mathbf{Ze}-\mathbf{Zd})$ is a uniformly random vector over $\mathbb{Z}_q$, and so is $\mathbf{t}'$. Thus, the tuple output by $\mathscr{B}$ follows the case $b = 1$ for Γ_3. The statement now follows by simple calculation. □

Game Γ_5: Here, we consider the plain LWE game. Recall that the challenger samples the public $\mathbf{A} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{(n+m)\times m})$, secret $\mathbf{s} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^m)$, error $\mathbf{e} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n+m})$. Then it flips a bit $b \xleftarrow{\$} \mathscr{U}(\{0,1\})$. If $b = 0$ then the challenger computes $\mathbf{t} := \mathbf{As}+\mathbf{e}$, and otherwise it samples $\mathbf{t} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n+m})$. At the end, the challenger outputs $(\mathbf{A}, \mathbf{t})$.

Lemma 6. For every efficient adversary $\mathscr{A}$, there is an efficient adversary $\mathscr{B}$ such that $\mathsf{Adv}_5(\mathscr{B}) \geq \mathsf{Adv}_4(\mathscr{A}) - \mathsf{negl}(n)$.

Proof. The reduction is identical to the one of Micciancio and Mol [MM11, Lemma 4.8] which we recall for completeness. Suppose the algorithm $\mathscr{B}$ is given a tuple $(\mathbf{A}, \mathbf{t})$ from Γ_5. If $\mathbf{A}$ is full-rank, then $\mathscr{B}$ can construct a (randomised) matrix $\mathbf{G} \in \mathbb{Z}_q^{n \times (n+m)}$ whose rows generate all the vectors $\mathbf{x}$ such that $\mathbf{x}^T\mathbf{A} = \mathbf{0}$. Also, if $\mathbf{A}$ is chosen at random among all full-rank matrices, then $\mathbf{G}$ is also distributed statistically close to a uniformly random. Then, $\mathscr{B}$ outputs $(\mathbf{G}, \mathbf{Gt})$ to $\mathscr{A}$ and returns what $\mathscr{A}$ outputs.

Suppose $b = 0$ and $\mathbf{t} = \mathbf{As} + \mathbf{e}$. Then $\mathbf{Gt} = \mathbf{GAs} + \mathbf{e} = \mathbf{Ge}$, which is the correct instance of Γ_4 for $b = 0$. On the other hand, if $\mathbf{t}$ is uniformly random, then so is $\mathbf{Gt}$. □

The statement of the theorem now follows by combining all the previous lemmas using reduction composition. □

3.5.3 Construction

We can now present our Frodo-inspired [BCD+16] split-KEM, which we call FrodoKEX+. The scheme is given in Figure 3.9. The key generation works as follows. The public key pk_A for party A is a pair $(\mathbf{A}, \mathbf{B}_\mathsf{A})$, where $\mathbf{A}$ is a uniformly random matrix over $\mathbb{Z}_q$ given as a public parameter, and $\mathbf{B}_\mathsf{A} := \mathbf{AS}_\mathsf{A} + \mathbf{D}_\mathsf{A}$ where $\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A} \xleftarrow{\$} \chi^{n \times \bar{n}}$. The secret key becomes a pair $\mathsf{sk}_\mathsf{A} = (\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A})$. Similarly, the public key pk_B for party B is a pair $(\mathbf{A}, \mathbf{B}_\mathsf{B})$, where $\mathbf{B}_\mathsf{B} := \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$, while the secret key is $\mathsf{sk}_\mathsf{B} = (\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B})$, where $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n} \times n}$.

Then, B samples a matrix $\mathbf{E}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n} \times \bar{n}}$ and computes the matrix $\mathbf{V} := \mathbf{S}_\mathsf{B}\mathbf{B}_\mathsf{A} + \mathbf{E}_\mathsf{B}$. Next, it computes $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$ and $\mathbf{K} \leftarrow \mathsf{rec}(\mathbf{V}, \mathsf{ct})$. Then, B outputs ct. Then, party A decapsulates as follows: given $(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{A}, \mathsf{ct})$, it computes $\mathbf{V}' = \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{F}_\mathsf{A}$ and $\mathbf{K}' = \mathsf{rec}(2\mathbf{V}', \mathsf{ct})$. Finally, A returns the key $\mathbf{K}'$.

We note that the construction can easily be made symmetric, in the sense that A could encapsulate using B's public key by changing the order of matrices when multiplying in Encaps such that the dimensions match. Then, Decaps can be modified similarly such that B can decapsulate the resulting ciphertext.

3.5.4 Security Analysis

Lemma 7 (Correctness)**.** Let χ be a symmetric distribution around 0 and δ_{corr} be the following probability:

$$\Pr\left[|\langle \mathbf{s}, \mathbf{d} \rangle + e + f| > \frac{q}{2^{B+2}} : \mathbf{s}, \mathbf{d} \xleftarrow{\$} \chi^{2n}, e, f \xleftarrow{\$} \chi\right]. \tag{3.2}$$

Then, sKEM defined in Figure 3.9 is $(\bar{n}^2\delta_{\mathsf{corr}})$-correct.

$\mathsf{KeyGenA}(1^\lambda)$

1: $\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A} \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\bar{n}})$
2: $\mathbf{F}_\mathsf{A} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
3: $\mathbf{B}_\mathsf{A} \leftarrow \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A}$
4: $\mathsf{pk}_\mathsf{A} \leftarrow (\mathbf{A}, \mathbf{B}_\mathsf{A})$
5: $\mathsf{sk}_\mathsf{A} \leftarrow (\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A}, \mathbf{F}_\mathsf{A})$
6: **return** $(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{A})$

$\mathsf{KeyGenB}(1^\lambda)$

1: $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times n}$
2: $\mathbf{F}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
3: $\mathbf{B}_\mathsf{B} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$
4: $\mathsf{pk}_\mathsf{B} \leftarrow (\mathbf{A}, \mathbf{B}_\mathsf{B})$
5: $\mathsf{sk}_\mathsf{B} \leftarrow (\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B}, \mathbf{F}_\mathsf{B})$
6: **return** $(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{B})$

$\mathsf{Encaps}(\mathsf{pk}_\mathsf{A} = (\mathbf{A}, \mathbf{B}_\mathsf{A}), \mathsf{sk}_\mathsf{B} = (\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B}, \mathbf{F}_\mathsf{B}))$

1: // We assume B encapsulates
2: $\mathbf{E}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
3: $\mathbf{V} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{B}_\mathsf{A} + \mathbf{E}_\mathsf{B}$
4: $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$
5: $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$
6: **return** $(\mathbf{K}, \mathsf{ct})$

$\mathsf{Decaps}(\mathsf{pk}_\mathsf{B} = (\mathbf{A}, \mathbf{B}_\mathsf{B}), \mathsf{sk}_\mathsf{A} = (\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A}, \mathbf{F}_\mathsf{A}), \mathsf{ct})$

1: $\mathbf{V}' \leftarrow \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{F}_\mathsf{A}$
2: $\mathbf{K}' \leftarrow \mathsf{rec}(2\mathbf{V}', \mathsf{ct})$
3: **return** $\mathbf{K}'$

Figure 3.9: Our variant of FrodoKEX [BCD+16] expressed as a split-KEM. The matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$ is assumed to be a public parameter and sampled uniformly at random.

Proof. Suppose $(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{A}) \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$ and $(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{B}) \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$, and let

$$(\mathbf{K}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{B}) \quad \text{and} \quad \mathbf{K}' \xleftarrow{\$} \mathsf{Decaps}(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{A}, \mathsf{ct}).$$

We want to prove that $\mathbf{K} = \mathbf{K}'$. By definition of encapsulation, we know that $\mathbf{K} = \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$ where $\mathsf{ct} = \mathsf{HelpRec}(\mathbf{V})$ and

$$\mathbf{V} = \mathbf{S}_\mathsf{B}\mathbf{B}_\mathsf{A} + \mathbf{E}_\mathsf{B} = \mathbf{S}_\mathsf{B}\mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{S}_\mathsf{B}\mathbf{D}_\mathsf{A} + \mathbf{E}_\mathsf{B}.$$

Thus, by Lemma 2, $\mathbf{K} = \lfloor \mathbf{V} \rceil_{2q,2^B}$. On the other hand,

$$\mathbf{V}' = \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{F}_\mathsf{A} = \mathbf{S}_\mathsf{B}\mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{F}_\mathsf{A}$$

which implies that $\mathbf{V} - \mathbf{V}' = \mathbf{S}_\mathsf{B}\mathbf{D}_\mathsf{A} + \mathbf{E}_\mathsf{B} - \mathbf{D}_\mathsf{B}\mathbf{S}_\mathsf{A} - \mathbf{F}_\mathsf{A}$. If $\|\mathbf{V} - \mathbf{V}'\|_\infty < \frac{q}{2^{B+2}}$ then by Lemma 2 we must have

$$\mathbf{K}' = \mathsf{rec}(2\mathbf{V}', \mathsf{HelpRec}(\mathbf{V})) = \lfloor \mathbf{V} \rceil_{2q,2^B} = \mathbf{K}$$

so correctness holds. Now, using the fact that χ is symmetric around 0, the probability $\|\mathbf{V} - \mathbf{V}'\|_\infty > \frac{q}{2^{B+2}}$ can be upper-bounded using the union bound as follows:

$$\Pr\left[\|\mathbf{S}_\mathsf{B}\mathbf{D}_\mathsf{A} + \mathbf{E}_\mathsf{B} - \mathbf{D}_\mathsf{B}\mathbf{S}_\mathsf{A} - \mathbf{F}_\mathsf{A}\|_\infty > \frac{q}{2^{B+2}}\right] \leq \bar{n}^2 \cdot \Pr\left[|\mathbf{s}_0^T\mathbf{d}_0 + \mathbf{s}_1^T\mathbf{d}_1 + e + f| > \frac{q}{2^{B+2}}\right]$$

where $\mathbf{s}_0, \mathbf{s}_1, \mathbf{d}_0, \mathbf{d}_1 \xleftarrow{\$} \chi^n$ and $e, f \xleftarrow{\$} \chi$. This concludes the proof. □

OW-CPA **Security.** Next, we focus on proving OW-CPA security.

Lemma 8 (OW-CPA Security)**.** Let $\bar{n} = O(\lambda)$ and χ be a symmetric distribution over $[-\gamma, \gamma]$ for any $\gamma > 0$. Then, under the $\mathsf{LWE}_{n,n,\chi,q}$ and $\mathsf{LWE}_{n+\bar{n},n,\chi,q}$ assumptions, for every efficient adversary $\mathscr{A}$, the probability of $\mathscr{A}$ winning the OW-CPA game is at most $2^{-B\bar{n}^2} + \mathsf{negl}(\lambda)$.

Proof. Let $\mathscr{A}$ be an efficient adversary against the OW-CPA game. We prove the statement using the hybrid games described explicitly in Figure 3.10. In each game Γ_i, we define ε_i to be the probability that the efficient adversary $\mathscr{A}$ wins the security game.

Game Γ_1: This is the standard OW-CPA game.

Game Γ_2: Instead of computing $\mathbf{B}_\mathsf{A} \leftarrow \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A}$, the experiment samples $\mathbf{B}_\mathsf{A} \leftarrow \mathscr{U}(\mathbb{Z}_q^{n\times\bar{n}})$. One can naturally build an efficient adversary, which can solve the $\mathsf{LWE}_{n,n,\chi,q}$ problem with probability at least $\frac{1}{\bar{n}}|\varepsilon_2 - \varepsilon_1|$. Hence, we deduce that this probability is negligible.

Game Γ_3: Here, the experiment computes the values $\mathbf{B}_\mathsf{B}$ and $\mathbf{V}$ differently. Namely, instead of computing:

$$\begin{bmatrix}\mathbf{B}_\mathsf{B} & \mathbf{V}\end{bmatrix} := \mathbf{S}_\mathsf{B}\begin{bmatrix}\mathbf{A} & \mathbf{B}_\mathsf{A}\end{bmatrix} + \begin{bmatrix}\mathbf{D}_\mathsf{B} & \mathbf{E}_\mathsf{B}\end{bmatrix},$$

it samples

$$\begin{bmatrix}\mathbf{B}_\mathsf{B} & \mathbf{V}\end{bmatrix} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{\bar{n}\times(n+\bar{n})}).$$

Thus, one can naturally construct an efficient reduction which solves $\mathsf{LWE}_{n+\bar{n},n,\chi,q}$ with probability at least $\frac{1}{\bar{n}}|\varepsilon_6 - \varepsilon_5|$.

Finally, it is easy to see that in Γ_3 the matrix $\mathbf{V}$ is actually uniformly random over $\mathbb{Z}_q$. Hence by Lemma 1, for the adversary $\mathscr{A}$, which is given ct, the key $\mathbf{K}$ looks uniformly random. Therefore, the probability of guessing the key is bounded by $2^{-\bar{n}^2 B}$. □

Deniability. We will use the (transposed) matrix version of ELWE where the secrets and errors are now matrices. In particular, we will be interested in the problem of distinguishing between

$$(\mathbf{A}, \mathbf{S}\mathbf{A} + \mathbf{E} \bmod q, \mathbf{Z}, \mathbf{W}, \mathbf{S}\mathbf{Z} + \mathbf{E}\mathbf{W} \bmod q)$$

and

$$(\mathbf{A}, \mathbf{T}, \mathbf{Z}, \mathbf{W}, \mathbf{S}\mathbf{Z} + \mathbf{E}\mathbf{W} \bmod q)$$

where $\mathbf{S} \xleftarrow{\$} \chi^{\bar{n}\times m}$, $\mathbf{E} \xleftarrow{\$} \chi^{\bar{n}\times n}$ and $\mathbf{T} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{\bar{n}\times n})$. This problem can be reduced to ELWE with reduction loss $\bar{n}$ via a standard hybrid argument.

We are ready to prove deniability of the split-KEM based on Extended-LWE. Intuitively, matrices $(\mathbf{S}, \mathbf{E}) := (\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B})$ will be the secret and error constructed by party B, which are hidden from the adversary, while $(\mathbf{Z}, \mathbf{W}) := (\mathbf{D}_\mathsf{A}, \mathbf{S}_\mathsf{A})$ will be the error and the secret generated by $\mathbf{A}$ which are

$\Gamma_1(\mathscr{A})$	$\Gamma_2(\mathscr{A})$	$\Gamma_3(\mathscr{A})$
$\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A} \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\overline{n}})$	$\mathbf{B}_\mathsf{A} \leftarrow \mathscr{U}(\mathbb{Z}_q^{n\times\bar{n}})$	$\mathbf{B}_\mathsf{A} \leftarrow \mathscr{U}(\mathbb{Z}_q^{n\times\bar{n}})$
$\mathbf{F}_\mathsf{A} \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$	$\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\overline{n}\times n}$	$\mathbf{B}_\mathsf{B} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{n\times\bar{n}})$
$\mathbf{B}_\mathsf{A} \leftarrow \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A}$	$\mathbf{F}_\mathsf{B} \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$	$\mathbf{V} \xleftarrow{\$} \mathscr{U}(\mathbb{Z}_q^{\bar{n}\times\bar{n}})$
$\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\overline{n}\times n}$	$\mathbf{B}_\mathsf{B} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$	$\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$
$\mathbf{F}_\mathsf{B} \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$	$\mathbf{E}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$	$\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$
$\mathbf{B}_\mathsf{B} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$	$\mathbf{V} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{B}_\mathsf{A} + \mathbf{E}_\mathsf{B}$	$\mathbf{K}' \xleftarrow{\$} \mathscr{A}(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B}, \mathsf{ct})$
$\mathbf{E}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$	$\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$	**return** $\mathbb{1}[\mathbf{K} = \mathbf{K}']$
$\mathbf{V} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{B}_\mathsf{A} + \mathbf{E}_\mathsf{B}$	$\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$	
$\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$	$\mathbf{K}' \xleftarrow{\$} \mathscr{A}(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B}, \mathsf{ct})$	
$\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$	**return** $\mathbb{1}[\mathbf{K} = \mathbf{K}']$	
$\mathbf{K}' \xleftarrow{\$} \mathscr{A}(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B}, \mathsf{ct})$		
return $\mathbb{1}[\mathbf{K} = \mathbf{K}']$		

Figure 3.10: Security games for the proof of Lemma 8. The lines in blue highlight the main differences from the previous game.

given as input to the simulator. The key observation is that the additional hint provided as $\mathbf{SZ} + \mathbf{EW} \bmod q$ will be used to simulate the "shared key" $\mathbf{V}$ (before applying the reconciliation function).

Theorem 4 (Deniability)**.** Let $\bar{n} = \mathsf{poly}(\lambda)$. Then, the sKEM defined in Figure 3.9 is deniable under the $\mathsf{ELWE}_{n,n,\bar{n},\chi,q}$ and $\mathsf{LWE}_{n,n,\chi,q}$ assumptions.

Proof. Let $\mathscr{A}$ be an efficient adversary against the deniability game. We prove the statement using the hybrid games defined in Figure 3.12. In each game Γ_i, we define ε_i to be the probability that the efficient adversary $\mathscr{A}$ outputs $b = 1$.

Game Γ_1: This is the standard (real) deniability experiment, which we recall here. First, both $\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A} \xleftarrow{\$} \chi^{n\times\overline{n}}$ and $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\overline{n}\times n}$ and $\mathbf{F}_\mathsf{A}, \mathbf{F}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$ are sampled. Then, the public keys $\mathbf{B}_\mathsf{A} = \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A}$ and $\mathbf{B}_\mathsf{B} = \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$ are computed. The encapsulation algorithm samples $\mathbf{E}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$ and sets $\mathbf{V} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{B}_\mathsf{A} + \mathbf{E}_\mathsf{B}$. Finally, the experiment runs $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$ and $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$ and eventually outputs

$$(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B}, \mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A}, \mathbf{F}_\mathsf{A}, \mathbf{K}, \mathsf{ct})$$

to the adversary $\mathscr{A}$.

Game Γ_2: The experiment is identical to the previous one, apart from the fact that

now $\mathbf{V}$ is explicitly computed as $\mathbf{V} = \mathbf{S}_\mathsf{B}\mathbf{D}_\mathsf{A} - \mathbf{D}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{E}_\mathsf{B}$. Clearly, $\varepsilon_1 = \varepsilon_2$ since

$$\begin{aligned}\mathbf{V} &= \mathbf{S}_\mathsf{B}\mathbf{D}_\mathsf{A} - \mathbf{D}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{E}_\mathsf{B} \\ &= \mathbf{S}_\mathsf{B}\mathbf{D}_\mathsf{A} - \mathbf{D}_\mathsf{B}\mathbf{S}_\mathsf{A} + (\mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B})\mathbf{S}_\mathsf{A} + \mathbf{E}_\mathsf{B} \\ &= \mathbf{S}_\mathsf{B}\mathbf{B}_\mathsf{A} + \mathbf{E}_\mathsf{B}.\end{aligned}$$

Game Γ_3: Here, the experiment follows Γ_2 with the only difference being that the experiment samples $\mathbf{B}_\mathsf{B}$ uniformly at random from $\mathbb{Z}_q^{\bar{n}\times n}$ instead of computing $\mathbf{B}_\mathsf{B} = \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$.

Lemma 9. There exists an efficient algorithm $\mathscr{B}$ that solves the $\mathsf{ELWE}_{n,n,\bar{n},\chi,q}$ problem with probability at least $\frac{1}{\bar{n}}|\varepsilon_3 - \varepsilon_2|$.

Proof. We provide a reduction $\mathscr{B}$ to the (transposed) matrix version of the Extended-LWE problem as described above. Namely, the reduction is given a tuple of matrices $(\mathbf{A}, \mathbf{B}, \mathbf{Z}, \mathbf{W}, \mathbf{H})$. Then, it sets $\mathbf{S}_\mathsf{A} := -\mathbf{W}$, $\mathbf{D}_\mathsf{A} := \mathbf{Z}$ and $\mathbf{B}_\mathsf{B} := \mathbf{B}$. Further, the reduction samples $\mathbf{F}_\mathsf{A} \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$ and computes

$$\mathbf{B}_\mathsf{A} := \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A} \quad \text{and} \quad \mathbf{V} := \mathbf{H} + \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{E}_\mathsf{B}$$

where $\mathbf{E}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$. Finally, the reduction runs $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$ and $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$ and outputs $(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B}, \mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A}, \mathbf{F}_\mathsf{A}, \mathbf{K}, \mathsf{ct})$ to the adversary.

Suppose the input tuple received by $\mathscr{B}$ is a true Extended-LWE instance, i.e. $\mathbf{B}_\mathsf{B} = \mathbf{B} = \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$ for $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\overline{n}\times n}$. This implies that $\mathbf{H} = \mathbf{S}_\mathsf{B}\mathbf{Z} + \mathbf{D}_\mathsf{B}\mathbf{W} = \mathbf{S}_\mathsf{B}\mathbf{D}_\mathsf{A} - \mathbf{D}_\mathsf{B}\mathbf{S}_\mathsf{A}$ and hence

$$\mathbf{V} = \mathbf{H} + \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{E}_\mathsf{B} = \mathbf{S}_\mathsf{B}\mathbf{D}_\mathsf{A} - \mathbf{D}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{E}_\mathsf{B}.$$

This implies that when the input tuple is the Extended-LWE instance then $\mathscr{B}$ perfectly simulates the output of Γ_2.[5] On the other hand, if $\mathbf{B}_\mathsf{B}$ is uniformly random then $\mathscr{B}$ perfectly simulates the output of Γ_3. Finally the statement follows by further reducing the matrix version of ELWE to the standard one. □

Game Γ_4: First, we rename the variables $(\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B}, \mathbf{E}_\mathsf{B}) := (\mathbf{S}_\mathsf{sim}, \mathbf{D}_\mathsf{sim}, \mathbf{E}_\mathsf{sim})$. Further, instead of picking $\mathbf{B}_\mathsf{B}$ uniformly at random, the experiment now samples alternative secrets/errors $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\overline{n}\times n}$ for B and sets $\mathbf{B}_\mathsf{B} := \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$. The rest is identical as in Γ_3.

Lemma 10. There exists an efficient algorithm $\mathscr{B}'$ solves the $\mathsf{LWE}_{n,n,\chi,q}$ problem with probability at least $\frac{1}{\bar{n}}|\varepsilon_3 - \varepsilon_2|$.

Proof. We describe a reduction $\mathscr{B}$ which solves the matrix version of LWE. Then, the reduction to plain LWE follows by a hybrid argument. First, $\mathscr{B}$ is given a tuple $(\mathbf{A}, \mathbf{B})$ where either $\mathbf{B} = \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$ for short $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B}$ or $\mathbf{B}$ is uniformly random. In either case, only given $\mathbf{A}$ and $\mathbf{B}$, the reduction $\mathscr{B}$ can simulate the rest of Γ_3 (and Γ_4). If $\mathbf{B} = \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$ then this becomes Γ_4, and when $\mathbf{B}_\mathsf{B}$ is uniformly random then $\mathscr{B}$ simulates Γ_3. □

[5] We used the fact that χ is symmetric around 0 to argue that $\mathbf{S}_\mathsf{A} := -\mathbf{W}$ is correctly distributed.

$\mathsf{Sim}(\mathbf{A}, \mathbf{B}_\mathsf{B}, \mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A}, \mathbf{F}_\mathsf{A})$

1: $\mathbf{S}_\mathsf{sim}, \mathbf{D}_\mathsf{sim} \xleftarrow{\$} \chi^{\bar{n}\times n}$
2: $\mathbf{E}_\mathsf{sim} \xleftarrow{\$} \chi^{\bar{n}\times \bar{n}}$
3: $\mathbf{V}_\mathsf{sim} \leftarrow \mathbf{S}_\mathsf{sim}\mathbf{D}_\mathsf{A} - \mathbf{D}_\mathsf{sim}\mathbf{S}_\mathsf{A} + \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{E}_\mathsf{sim}$
4: $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V}_\mathsf{sim})$
5: $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}_\mathsf{sim}, \mathsf{ct})$
6: **return** $(\mathbf{K}, \mathsf{ct})$

Figure 3.11: Simulator for the deniability game.

Finally, we present the simulator in Figure 3.11. Γ_4 can now be alternatively described in the following way. The experiment first samples $\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A} \xleftarrow{\$} \chi^{n\times\overline{n}}$ and $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \mathbb{Z}_q^{\overline{n}\times n}$ and $\mathbf{F}_\mathsf{A} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$. Further, the public keys are defined as $\mathbf{B}_\mathsf{A} = \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A}$ and $\mathbf{B}_\mathsf{B} = \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$. Finally, it runs $(\mathbf{K}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Sim}(\mathbf{A}, \mathbf{B}_\mathsf{B}, \mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A}, \mathbf{F}_\mathsf{A})$ and outputs $(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B}, \mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A}, \mathbf{F}_\mathsf{A}, \mathbf{K}, \mathsf{ct})$. Thus, deniability follows by simply combining the previous lemmas. □

decaps-CPA **Security.** Finally, we show that our split-KEM satisfies the decaps-CPA security notion (see Definition 22).

Lemma 11 (decaps-CPA Security)**.** Let $\bar{n} = O(\lambda)$, m be such that the ciphertext space of sKEM is $\{0,1\}^m$, and χ be a probability distribution over $[-\gamma, \gamma]$ symmetric around 0 for any $\gamma > 0$. Suppose $\mathsf{LWE}_{n+\bar{n},n,\chi,q}$ is hard. Then, for every efficient algorithm $\mathscr{A}$, the probability of winning the decaps-CPA game is at most $2^m \cdot (\delta_{\mathsf{cpa}}^{\bar{n}^2} + \mathsf{negl}(\lambda))$ where

$$\delta_{\mathsf{cpa}} := \max_{\substack{\mathsf{ct}\in\{0,1\} \\ u\in\mathbb{Z}_{2B}}} \Pr_{w\xleftarrow{\$}\mathbb{Z}_q} [\mathsf{rec}(2w, \mathsf{ct}) = u] \ . \tag{3.3}$$

Proof. Let $\mathscr{A}$ be an efficient adversary against the decaps-CPA game. We prove the statement using the hybrid games described explicitly in Figure 3.13. In each game Γ_i, we define ε_i to be the probability that the efficient adversary $\mathscr{A}$ wins the security game.

Game Γ_1: This is the standard decaps-CPA game corresponding to the sKEM in Figure 3.9.

Game Γ_2: In this game, the ciphertext ct is not given to the adversary anymore. Note that the first phase adversary outputting $\mathbf{B}$ is now useless and it can be removed[6], along with the operations needed to compute ct. Given the ciphertext space is $\{0,1\}^m$ for some $m \in \mathbb{Z}$, we have $\epsilon_2 \geq \frac{1}{2^m}\epsilon_1$ as any adversary in Γ_2 can simulate the view of an adversary in Γ_1 by guessing ct.

Game Γ_3: In this game, the only change is that instead of computing $\mathbf{B}_\mathsf{B} = \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$, it is picked uniformly at random from $\mathbb{Z}_q^{\bar{n}\times n}$. The indistinguishability between Γ_3 and Γ_2

[6] Formally, we would run it anyway to propagate the adversary's state st, but we omit this for simplicity.

$\Gamma_1(\mathscr{A})$

1: $\mathbf{S}_A, \mathbf{D}_A \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\overline{n}})$
2: $\mathbf{F}_A \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$
3: $\mathbf{B}_A \leftarrow \mathbf{A}\mathbf{S}_A + \mathbf{D}_A$
4: $\mathsf{pk}_A = (\mathbf{A}, \mathbf{B}_A)$
5: $\mathsf{sk}_A = (\mathbf{S}_A, \mathbf{D}_A, \mathbf{F}_A)$
6: $\mathbf{S}_B, \mathbf{D}_B \xleftarrow{\$} \chi^{\overline{n}\times n}$
7: $\mathbf{F}_B \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$
8: $\mathbf{B}_B \leftarrow \mathbf{S}_B\mathbf{A} + \mathbf{D}_B$
9: $\mathsf{pk}_B = (\mathbf{A}, \mathbf{B}_B)$
10: $\mathbf{E}_B \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
11: $\mathbf{V} \leftarrow \mathbf{S}_B\mathbf{B}_A + \mathbf{E}_B$
12: $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$
13: $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$
14: $b \xleftarrow{\$} \mathscr{A}(\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{sk}_A, \mathbf{K}, \mathsf{ct})$
15: **return** b

$\Gamma_2(\mathscr{A})$

1: $\mathbf{S}_A, \mathbf{D}_A \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\overline{n}})$
2: $\mathbf{F}_A \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$
3: $\mathbf{B}_A \leftarrow \mathbf{A}\mathbf{S}_A + \mathbf{D}_A$
4: $\mathsf{pk}_A = (\mathbf{A}, \mathbf{B}_A)$
5: $\mathsf{sk}_A = (\mathbf{S}_A, \mathbf{D}_A, \mathbf{F}_A)$
6: $\mathbf{S}_B, \mathbf{D}_B \xleftarrow{\$} \chi^{\overline{n}\times n}$
7: $\mathbf{F}_B \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$
8: $\mathbf{B}_B \leftarrow \mathbf{S}_B\mathbf{A} + \mathbf{D}_B$
9: $\mathsf{pk}_B = (\mathbf{A}, \mathbf{B}_B)$
10: $\mathbf{E}_B \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
11: $\mathbf{V} \leftarrow \mathbf{S}_B\mathbf{D}_A - \mathbf{D}_B\mathbf{S}_A + \mathbf{B}_B\mathbf{S}_A + \mathbf{E}_B$
12: $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$
13: $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$
14: $b \xleftarrow{\$} \mathscr{A}(\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{sk}_A, \mathbf{K}, \mathsf{ct})$
15: **return** b

$\Gamma_3(\mathscr{A})$

1: $\mathbf{S}_A, \mathbf{D}_A \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\overline{n}})$
2: $\mathbf{F}_A \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$
3: $\mathbf{B}_A \leftarrow \mathbf{A}\mathbf{S}_A + \mathbf{D}_A$
4: $\mathsf{pk}_A = (\mathbf{A}, \mathbf{B}_A)$
5: $\mathsf{sk}_A = (\mathbf{S}_A, \mathbf{D}_A, \mathbf{F}_A)$
6: $\mathbf{S}_B, \mathbf{D}_B \xleftarrow{\$} \chi^{\overline{n}\times n}$
7: $\mathbf{F}_B \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$
8: $\mathbf{B}_B \leftarrow \mathscr{U}(\mathbb{Z}_q^{\bar{n}\times n})$
9: $\mathsf{pk}_B = (\mathbf{A}, \mathbf{B}_B)$
10: $\mathbf{E}_B \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
11: $\mathbf{V} \leftarrow \mathbf{S}_B\mathbf{D}_A - \mathbf{D}_B\mathbf{S}_A + \mathbf{B}_B\mathbf{S}_A + \mathbf{E}_B$
12: $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$
13: $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$
14: $b \xleftarrow{\$} \mathscr{A}(\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{sk}_A, \mathbf{K}, \mathsf{ct})$
15: **return** b

$\Gamma_4(\mathscr{A})$

1: $\mathbf{S}_A, \mathbf{D}_A \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\overline{n}})$
2: $\mathbf{F}_A \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$
3: $\mathbf{B}_A \leftarrow \mathbf{A}\mathbf{S}_A + \mathbf{D}_A$
4: $\mathsf{pk}_A = (\mathbf{A}, \mathbf{B}_A)$
5: $\mathsf{sk}_A = (\mathbf{S}_A, \mathbf{D}_A, \mathbf{F}_A)$
6: $\mathbf{S}_B, \mathbf{D}_B \xleftarrow{\$} \chi^{\overline{n}\times n}$
7: $\mathbf{F}_B \xleftarrow{\$} \chi^{\overline{n}\times\overline{n}}$
8: $\mathbf{B}_B \leftarrow \mathbf{S}_B\mathbf{A} + \mathbf{D}_B$
9: $\mathsf{pk}_B = (\mathbf{A}, \mathbf{B}_B)$
10: $\mathbf{E}_{\mathsf{sim}} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
11: $\mathbf{S}_{\mathsf{sim}}, \mathbf{D}_{\mathsf{sim}} \xleftarrow{\$} \chi^{\overline{n}\times n}$
12: $\mathbf{V} \leftarrow \mathbf{S}_{\mathsf{sim}}\mathbf{D}_A - \mathbf{D}_{\mathsf{sim}}\mathbf{S}_A + \mathbf{B}_B\mathbf{S}_A + \mathbf{E}_{\mathsf{sim}}$
13: $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$
14: $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$
15: $b \xleftarrow{\$} \mathscr{A}(\mathsf{pk}_A, \mathsf{pk}_B, \mathsf{sk}_A, \mathbf{K}, \mathsf{ct})$
16: **return** b

Figure 3.12: Security games for the proof of Theorem 4. The lines in blue highlight the main differences from the previous game. The lines in gray correspond to the simulator defined in Figure 3.11.

follows directly from $\mathsf{LWE}_{n,n,\chi,q}$.

Game Γ_4: Now, instead of computing $\mathbf{B}_\mathsf{A}$ and $\mathbf{V}'$ as:

$$\begin{bmatrix}\mathbf{B}_\mathsf{A}\\ \mathbf{V}'\end{bmatrix} = \begin{bmatrix}\mathbf{A}\\ \mathbf{B}_\mathsf{B}\end{bmatrix}\mathbf{S}_\mathsf{A} + \begin{bmatrix}\mathbf{D}_\mathsf{A}\\ \mathbf{F}_\mathsf{A}\end{bmatrix},$$

the experiment samples $\mathbf{B}_\mathsf{A} \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^{n\times\bar{n}})$ and $\mathbf{V}' \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^{\bar{n}\times\bar{n}})$ uniformly at random. Then, the reduction executes Lines 4 to 6 of Γ_4. Clearly there is an efficient adversary which solves $\mathsf{LWE}_{n+\bar{n},n,\chi,q}$ with probability at least $\frac{1}{\bar{n}}|\varepsilon_4 - \varepsilon_3|$.

Finally, since $\mathbf{V}'$ is uniformly random, the probability that any adversary wins Γ_4, i.e. $\mathbf{K}_\mathsf{A} = \mathbf{K}'_\mathsf{A}$, can be upper-bounded by $\delta_{\mathsf{cpa}}^{\bar{n}^2}$ by definition of δ_{cpa}. The statement now follows by combining the previous hybrid games. □

$\Gamma_1(\mathscr{A})$

1: $\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A} \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\bar{n}})$
2: $\mathbf{F}_\mathsf{A} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
3: $\mathbf{B}_\mathsf{A} \leftarrow \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A}$
4: $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times n}$
5: $\mathbf{F}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
6: $\mathbf{B}_\mathsf{B} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$
7: $\mathbf{B}, \mathsf{st} \xleftarrow{\$} \mathscr{A}(\mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B})$
8: $\mathbf{E}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
9: $\mathbf{V} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{B} + \mathbf{E}_\mathsf{B}$
10: $\mathsf{ct} \leftarrow \mathsf{HelpRec}(\mathbf{V})$
11: $\mathbf{K} \leftarrow \mathsf{rec}(2\mathbf{V}, \mathsf{ct})$
12: $\mathbf{K}'_\mathsf{A}, \mathsf{ct}' \xleftarrow{\$} \mathscr{A}(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B}, \mathsf{ct}, \mathsf{st})$
13: $\mathbf{V}' \leftarrow \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{F}_\mathsf{A}$
14: $\mathbf{K}_\mathsf{A} \leftarrow \mathsf{rec}(2\mathbf{V}', \mathsf{ct}')$
15: **return** $\mathbb{1}[\![\mathbf{K}_\mathsf{A} = \mathbf{K}'_\mathsf{A}]\!]$

$\Gamma_2(\mathscr{A})$

1: $\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A} \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\bar{n}})$
2: $\mathbf{F}_\mathsf{A} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
3: $\mathbf{B}_\mathsf{A} \leftarrow \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A}$
4: $\mathbf{S}_\mathsf{B}, \mathbf{D}_\mathsf{B} \xleftarrow{\$} \chi^{\bar{n}\times n}$
5: $\mathbf{B}_\mathsf{B} \leftarrow \mathbf{S}_\mathsf{B}\mathbf{A} + \mathbf{D}_\mathsf{B}$
6: $\mathbf{K}'_\mathsf{A}, \mathsf{ct}' \xleftarrow{\$} \mathscr{A}(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B})$
7: $\mathbf{V}' \leftarrow \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{F}_\mathsf{A}$
8: $\mathbf{K}_\mathsf{A} \leftarrow \mathsf{rec}(2\mathbf{V}', \mathsf{ct}')$
9: **return** $\mathbb{1}[\![\mathbf{K}_\mathsf{A} = \mathbf{K}'_\mathsf{A}]\!]$

$\Gamma_3(\mathscr{A})$

1: $\mathbf{S}_\mathsf{A}, \mathbf{D}_\mathsf{A} \xleftarrow{\$} \chi(\mathbb{Z}_q^{n\times\bar{n}})$
2: $\mathbf{F}_\mathsf{A} \xleftarrow{\$} \chi^{\bar{n}\times\bar{n}}$
3: $\mathbf{B}_\mathsf{A} \leftarrow \mathbf{A}\mathbf{S}_\mathsf{A} + \mathbf{D}_\mathsf{A}$
4: $\mathbf{B}_\mathsf{B} \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^{\bar{n}\times n})$
5: $\mathbf{K}'_\mathsf{A}, \mathsf{ct}' \xleftarrow{\$} \mathscr{A}(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B})$
6: $\mathbf{V}' \leftarrow \mathbf{B}_\mathsf{B}\mathbf{S}_\mathsf{A} + \mathbf{F}_\mathsf{A}$
7: $\mathbf{K}_\mathsf{A} \leftarrow \mathsf{rec}(2\mathbf{V}', \mathsf{ct}')$
8: **return** $\mathbb{1}[\![\mathbf{K}_\mathsf{A} = \mathbf{K}'_\mathsf{A}]\!]$

$\Gamma_4(\mathscr{A})$

1: $\mathbf{B}_\mathsf{A} \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^{n\times\bar{n}})$
2: $\mathbf{B}_\mathsf{B} \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^{\bar{n}\times n})$
3: $\mathbf{V}' \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^{\bar{n}\times\bar{n}})$
4: $\mathbf{K}'_\mathsf{A}, \mathsf{ct}' \xleftarrow{\$} \mathscr{A}(\mathbf{A}, \mathbf{B}_\mathsf{A}, \mathbf{B}_\mathsf{B})$
5: $\mathbf{K}_\mathsf{A} \leftarrow \mathsf{rec}(2\mathbf{V}', \mathsf{ct}')$
6: **return** $\mathbb{1}[\![\mathbf{K}_\mathsf{A} = \mathbf{K}'_\mathsf{A}]\!]$

Figure 3.13: Security games for the proof of Lemma 11. The lines in blue highlight the main differences from the previous game.

Remark. We highlight that the construction does not require a super-polynomial modulus. To see this, let us first focus on δ_{cpa} in (3.3). In the instantiation we set $B = O(1), \bar{n} = O(\sqrt{\lambda})$ and $\gamma = O(1)$. Further, δ_{cpa} can be bounded by $2^{-B} + 1/q$, and if $q = \mathsf{poly}(\lambda)$ then $\delta_{\mathsf{cpa}} \leq (2^{-B} + 1/q)^{\bar{n}^2} = \mathsf{negl}(\lambda)$. Hence, the winning probability in Lemma 11 is negligible for $q = \mathsf{poly}(\lambda)$.

Now, let us move on to Equation 3.2. Recall that n is a dimension responsible for hardness of LWE, say $n = \mathsf{poly}(\lambda)$. Suppose as in Lemma 11 that the distribution χ outputs integers between $-\gamma$ and γ. Then, to obtain $\delta_{\mathsf{corr}} = 0$, we need to pick $q > 2^{B+2}(2n\gamma^2 + 2\gamma) = \mathsf{poly}(\lambda)$, hence asymptotically the modulus can still be polynomial. In practice, we would allow negligible δ_{corr} (as we do for K-Waay), and thus we further decrease the modulus.

3.5.5 Building a UNF-1KCA and IND-1BatchCCA Split-KEM

$\mathsf{KeyGen}_{\mathsf{sKEM}}(1^\lambda)$

1: $(\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}_{\mathsf{sKEM}_0}(1^\lambda)$
2: **return** $(\mathsf{pk},\mathsf{sk})$

$\mathsf{Encaps}_{\mathsf{sKEM}}(\mathsf{pk}_\mathsf{A},\mathsf{sk}_\mathsf{B})$

1: $K_0,\mathsf{ct} \xleftarrow{\$} \mathsf{Encaps}_{\mathsf{sKEM}_0}(\mathsf{pk}_\mathsf{A},\mathsf{sk}_\mathsf{B})$
2: $t \leftarrow H'(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B},\mathsf{ct},K_0)$
3: $K \leftarrow H(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B},\mathsf{ct},K_0)$
4: **return** $K,(\mathsf{ct},t)$

$\mathsf{Decaps}_{\mathsf{sKEM}}(\mathsf{pk}_\mathsf{B},\mathsf{sk}_\mathsf{A},(\mathsf{ct},t))$

1: $K_0' \leftarrow \mathsf{Decaps}_{\mathsf{sKEM}_0}(\mathsf{pk}_\mathsf{B},\mathsf{sk}_\mathsf{A},(\mathsf{ct},t))$
2: **if** $H'(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B},\mathsf{ct},K_0') \neq t$:
3: **return** $\perp$
4: **return** $H(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B},\mathsf{ct},K_0')$

Figure 3.14: T_{CH} transform for split-KEMs. We assume that pk_B can be derived from sk_B or is contained in it.

We have proven so far that the modified version of FrodoKEX given above is decaps-CPA and OW-CPA. We show now that any scheme satisfying both these properties can easily be transformed into a UNF-1KCA and IND-1BatchCCA split-KEM in the ROM and QROM. The construction is similar to the T_{CH} transform introduced by Huguenin-Dumittan and Vaudenay [HV22] translated to the split-KEM setting. We present it in Figure 3.14. Then, the following theorem states the security guarantees of the resulting split-KEM.

Theorem 5. Let sKEM_0 be any split-KEM and $\mathsf{sKEM} := \mathsf{T}_{\mathsf{CH}}(\mathsf{sKEM}_0)$ be the split-KEM obtained from applying the T_{CH} transform (Figure 3.14) to sKEM_0. Then, in the ROM, we have that for any efficient UNF-1KCA adversary $\mathscr{A}$, one can build efficient $\mathscr{B}$ and $\mathscr{C}$ adversaries s.t.

$$\mathsf{Adv}^{\mathrm{unf-1kca}}_{\mathsf{sKEM}}(\mathscr{A}) \leq \frac{q_{H'}^2+1}{2^s} + (q_H + q_{H'} + 1)\cdot \mathsf{Adv}^{\mathrm{decaps-cpa}}_{\mathsf{sKEM}_0}(\mathscr{C}) ,$$

where q_H and $q_{H'}$ are the number of queries made by $\mathscr{A}$ to the random oracles H and H', respectively, and s is the output size of both random oracles. In the QROM, the bound becomes

$$\mathsf{Adv}^{\mathrm{unf-1kca}}_{\mathsf{sKEM}}(\mathscr{A}) \leq \frac{8(q_H + q_{H'})^2}{2^{2s}} + \epsilon + 2(2(q_H + q_{H'}) + 1)^2 \cdot \mathsf{Adv}^{\mathrm{decaps-cpa}}_{\mathsf{sKEM}_0}(\mathscr{B}) ,$$

where $\epsilon := \frac{2}{2^s} + 8\sqrt{2/2^s} + \frac{40e^2(q_{H'}+2)^3+2}{2^s}$.

n	$\bar{n}$	q	B	χ	$\lvert t\rvert$	$\lvert\mathsf{pk}\rvert$	$\lvert\mathsf{ct}\rvert$
1452	8	31751	4	$\mathcal{U}(\{-1,1\})$	64B	21.3KB	72B

Table 3.2: Concrete parameters for our lattice-based split-KEM. We note that in practice, we do not need to include the whole matrix $\mathbf{A}$ in the public key pk, but rather the seed for the pseudorandom function to generate it (as is the case in this table). The ciphertext ct comprises the original split-KEM ciphertext (8B) and the tag t (64B).

$\bar{n}^2\delta_{\mathsf{corr}}$ (3.2)	δ_{elwe} (3.1)	δ_{cpa} (3.3)
2^{-48}	2^{-46}	$2^{-3.9996}$

Table 3.3: Correctness and security terms.

Proof. We defer the proof to Appendix A.2. □

Similarly, we have that the T_{CH} transform makes an IND-1BatchCCA scheme out of an OW-CPA one, which is stated in the following theorem.

Theorem 6. Let sKEM_0 be any split-KEM and $\mathsf{sKEM} := \mathsf{T}_{\mathsf{CH}}(\mathsf{sKEM}_0)$ be the split-KEM obtained from applying the T_{CH} transform (Figure 3.14) to sKEM_0. Then, in the ROM, we have that for any efficient IND-1BatchCCA adversary $\mathscr{A}$, one can build efficient $\mathscr{B}$ s.t.

$$\mathsf{Adv}^{\mathrm{ind-1batchcca}}_{\mathsf{sKEM}}(\mathscr{A}) \leq \frac{q_{H'}^2 + d}{2^s} + 2(q_H + q_{H'} + d) \cdot \mathsf{Adv}^{\mathrm{ow-cpa}}_{\mathsf{sKEM}_0}(\mathscr{B})$$

where q_H and $q_{H'}$ are the number of queries made by $\mathscr{A}$ to the random oracles H and H', respectively, s is the output size of both random oracles, and d is the number of tuples submitted to the IND-1BatchCCA oracle BatchDEC. In the QROM, the previous bound becomes

$$\mathsf{Adv}^{\mathrm{ind-1batchcca}}_{\mathsf{sKEM}}(\mathscr{A}) \leq \delta + \epsilon_1 + \epsilon_2 + \epsilon_3 + 2(q_H + d + q_{H'})\sqrt{2\mathsf{Adv}^{\mathrm{ow-cpa}}_{\mathsf{sKEM}_0}(\mathscr{B})}\,,$$

where δ is the correctness error, $\epsilon_1 = \frac{40e^2(q_{H'}+d+1)^3+2}{2^s}$, $\epsilon_2 = 8d(d+2q_{H'}+1)\sqrt{2/2^s}$ and $\epsilon_3 = \frac{4d}{2^s}$.

Proof. As the proof is nearly identical to the proof of IND-qCCA security of the T_{CH} transform for PKE/KEM [HV22], we defer it to the Appendix A.3. □

3.5.6 Concrete Instantiation

In Table 3.2 we propose a parameter set for FrodoKEX+ where we aim for 256-bit security before applying the transform and 128-bit (resp. 64-bit) security after the transform assuming 2^{64} random oracle (resp. quantum random oracle) queries. In addition, we give the security terms in Table 3.3. We show in the following how these parameters were computed, where we set $(B, \bar{n}) = (4, 8)$.

Correctness Error and Security Loss. One of the main challenges in instantiating our FrodoKEX variant is computing δ_{corr} and δ_{elwe} from Equations 3.2 and 3.1. They are related to the correctness error and the security of loss of ELWE. To this end, we introduce a simple distribution χ which will allow us to efficiently compute these values. Namely, we set χ to be a uniform distribution over the set $\{-1,1\}$. Clearly, it is symmetric around 0 and has standard deviation equal to 1.

Another useful property of this distribution is that a product XY, where $X, Y \xleftarrow{\$} \chi$, still follows the distribution of χ. Based on this observation, we have

$$\delta_{\mathsf{corr}} = \Pr_{\substack{X_1,\ldots,X_{2n+1} \xleftarrow{\$} \chi \\ E \xleftarrow{\$} \chi}} \left[\left| \sum_{i=1}^{2n+1} X_i + E \right| > \frac{q}{2^{B+2}} \right].$$

We can directly compute this term using Laurent polynomials. Namely, define

$$P(X) := \Pr_{X \xleftarrow{\$} \chi}[X = 1] \cdot X + \Pr_{X \xleftarrow{\$} \chi}[X = -1] \cdot X^{-1} = \frac{1}{2} \cdot \left(X + X^{-1}\right).$$

Then, using the convolution properties, we observe that the probability of $X_1 + \ldots + X_{2n+2} = k$, for some $-2n-2 \leq k \leq 2n+2$, is equal to the k-th coefficient of the polynomial $P(X)^{2n+2}$. Hence, we calculate δ_{corr} by computing $P(X)^{2n+2}$ and summing all the k-th coefficients, such that $2n+2 \geq |k| > \frac{q}{2^{B+2}}$.

We now turn into computing δ_{elwe}. The first step is the analysis of the following random variable $\mathbf{v} = \frac{1}{2} \cdot (e-d) \cdot \mathbf{z}$, where $e, d \xleftarrow{\$} \chi$ and $\mathbf{z} \xleftarrow{\$} \chi^{\bar{n}}$. We denote this distribution as $\mathcal{V}$. By simple calculation we get:

$$\Pr[\mathbf{v} = \mathbf{a}] = \Pr\left[\frac{1}{2} \cdot (e-d) \cdot \mathbf{z} = \mathbf{a}\right] = \begin{cases} \frac{1}{2} \text{ if } \mathbf{a} = \mathbf{0} \\ \frac{1}{2^{\bar{n}+1}} \text{ if } \mathbf{a} \in \{-1,1\}^{\bar{n}} \\ 0 \text{ otherwise} \end{cases}$$

Then, the multivariate Laurent polynomial corresponding to $\mathbf{v}$ has an elegant form:

$$P(X_1, \ldots, X_{\bar{n}}) = \frac{1}{2} + \frac{1}{2^{\bar{n}+1}} \prod_{i=1}^{\bar{n}} (X_i + X_i^{-1}).$$

As before, we observe that δ_{elwe} is the probability that for $\mathbf{v}_1, \ldots, \mathbf{v}_{n+m} \xleftarrow{\$} \mathcal{V}$,

$$2 \cdot (\mathbf{v}_1 + \ldots + \mathbf{v}_{n+m}) = \mathbf{0} \pmod{q} \iff \mathbf{v}_1 + \ldots + \mathbf{v}_{n+m} = \mathbf{0}^7.$$

In terms of the newly defined Laurent polynomials, δ_{elwe} is the constant coefficient of:

$$P(X_1, \ldots, X_{\bar{n}})^{n+m} = \sum_{j=0}^{n+m} \binom{n+m}{j} \frac{1}{2^{n+m-j}} \cdot \frac{1}{2^{(\bar{n}+1)j}} \cdot \prod_{i=1}^{\bar{n}} (X_i + X_i^{-1})^j.$$

[7] This holds as long as $n+m < q/2$ since then no modulo overflow occurs.

We now look at the constant coefficient of each of the $n+m+1$ terms of the sum. The first observation is that

$$(X_i + X_i^{-1})^j = \sum_{k=0}^{j} \binom{j}{k} X_i^{(j-k)} X_i^{-k} = \sum_{k=0}^{j} \binom{j}{k} X_i^{(j-2k)}.$$

Hence, the constant coefficient of the expression above is 0 if j is odd, and $\binom{j}{j/2}$ when j is even. Consequently, the constant coefficient of $\prod_{i=1}^{\bar{n}} (X_i + X_i^{-1})^j$ is either 0, for odd j, or $\binom{j}{j/2}^{\bar{n}}$ for even j. Hence, we conclude that

$$\delta_{\mathsf{elwe}} = \sum_{j \text{ even}} \binom{n+m}{j} \frac{1}{2^{n+m-j}} \cdot \frac{1}{2^{(\bar{n}+1)j}} \cdot \binom{j}{\frac{j}{2}}^{\bar{n}}$$

which can then be computed efficiently for our parameters. Finally, δ_{cpa} can be straightforwardly computed for small primes [8], such as $\approx 2^{15}$.

Hardness of Extended-LWE. We measure the hardness following the methodology used for the original FrodoKEX [BCD+16] for fair comparison, and refer to it for more details on the attacks. Here, the main bottleneck of setting the parameters is the reduction loss between ELWE and plain LWE. Taking this into account for the parameters proposed above, we aim for 307-bit classical LWE security.

We consider the primal and dual BKZ attacks [SE94, CN11]. As a subroutine, the BKZ algorithm with block-size b uses an algorithm for the shortest vector problem (SVP) in lattices of dimension b. As in Frodo [BCD+16], for precautionary purposes we only count the cost of one such call (even though in practice it will run the SVP sub-algorithm polynomially many times). The lower-bound on the time complexity of one call is given by about $b2^{cb}$ CPU cycles, where $c \approx 0.292$ for classical attacks, and $c \approx 0.265$ for quantum attacks (see Laarhoven [Laa16, Section 14.2.10]). For 307-bit classical security, this corresponds to the block size being 1018, and the root Hermite factor being ≈ 1.0020 (in the quantum setting these parameters correspond to 279 bits of security). Further, we estimate the hardness of LWE against known attacks using the LWE estimator by Albrecht et al. [APS15]. Namely, we run the estimator under both "sieving" and "enumeration", and set the final root Hermite factor δ as the largest root Hermite factor returned by the program. Finally, we make sure that $\delta_{\mathsf{cpa}}^{\bar{n}^2} \approx 2^{-256}$ for the decaps-CPA proof.

3.6 Evaluation and Discussion

In this section, we evaluate empirically our protocol both in terms of time and space usage, and then discuss some aspects of our modelling and protocol. Hereafter, we refer to the X3DH-like protocol of Brendel et al. [BFG+22b] as SPQR, and the baseline deniable protocol

[8] One can formally prove that δ_{cpa} can be bounded by $\frac{1}{2^B} + \frac{1}{q}$, but we compute the value directly instead.

Scheme	Cl. (C)	Cl. (Q)	ROM bnd	QROM bnd	Assumption
FrodoKEX+	128	64	$(q_H+d)/2^{192}$	$(q_H+d)/2^{128}$	LWE
Raptor [LAZ19]	114	103	?	✗	NTRU
DualRing-LB [YEL+21]	(128)	(64)	?	✗	MSIS, MLWE
Falafl [Beu20]	128	64	?	✗	MSIS, MLWE

Table 3.4: Security comparison between FrodoKEX+and several post-quantum RS. 'Cl.' stands for claimed number of security bits. DualRing-LB's authors do not seem to make a clear security claim, we thus assume NIST level I. '?' indicates that no bound is explicitly given for the security, '✗' indicates that no proof is provided in the QROM.

(i.e., with ring signatures and without NIZKs) by Hashimoto et al. [HKKP22] as HKKP.

3.6.1 Benchmarks

Security of the Relevant Non-Standard Primitive. Like K-Waay, SPQR and HKKP can each be implemented using only (soon to be) standardised primitives, except for a single primitive in each case, we consider here the security of the relevant non-standard primitives. In the case of K-Waay it is a split-KEM, here implemented using a variant of FrodoKEX passed through the T_{CH} transform (that we call FrodoKEX+), and in the case of both HKKP and SPQR it is a ring signature scheme, or RS (or a designated-verifier signature scheme (DVS) derived from RS). The authors of both SPQR and HKKP proposed possible implementations for the RS without picking one in particular. The most efficient one for a ring of size 2 we are aware of that has an existing C implementation is Raptor [LAZ19] which we use for the benchmarks below. Other candidates would be Falafl [Beu20] or DualRing-LB [YEL+21].

We present in Table 3.4 a summary of the security claims, approximate leading factor in the bounds in the (Q)ROM, and assumptions for these non-standard primitives. We note that none of these primitives are proven secure in the standard model and all are based on lattices.

First, we note that parameters for these RS schemes were chosen *before* the reduction in the ROM was performed. That is, a primitive P based on lattices is built, parameters are chosen such that P satisfies the security claim, then P is used to build a RS in the ROM, which incurs a loss factor that usually depends on the number of queries to the random oracle q_H. In particular, it is common to have at least a q_H factor in the security bound (e.g. if the adversary can make 2^{32} queries to the RO, the security level is reduced by 32 bits). Therefore, the claimed security level does not match the *provable* security level. In the QROM, the security loss is usually greater: square root and q_H^2 or q_H^3 losses are quite common, however these schemes have not been proven secure in this model.

We took a different approach in designing a split-KEM with a conservative assumption (i.e., plain LWE) and choice of parameters. Therefore, FrodoKEX+ with our proposed parameters achieve 128 (resp. 64) bits of classical (resp. quantum) security *after* the (Q)ROM proof. We provide the (approximate) highest terms of both the ROM and QROM security bounds in

Table 3.4. These satisfy our security claims as long as $q_H \leq 2^{64}$ in the ROM and $q_H \cdot d \leq 2^{64}$ in the QROM, where d is the number of public key/ciphertext tuples allowed in the IND-1BatchCCA game. In K-Waay, d corresponds to the number of distinct users trying to communicate with an offline receiver after all prekeys have run out, and thus should typically be small. We note that the leading coefficients in all our bounds are always related to the (Q)ROM security (e.g. finding a collision) and not to the underlying hardness assumptions. Therefore, one can easily make these coefficients smaller by picking a larger digest size n, e.g. 512 bits.

The reason behind the approximations and lack of QROM proofs for PQ ring signatures is likely the youth of the field and the speed at which it is evolving. Still, we believe it is worth noting as it makes any comparison between our protocol and previous ones quite difficult.

Benchmarking. The protocols we benchmarked are: our own implementation of the X3DH protocol; Brendel et al.'s [BFG+22b] protocol SPQR based on PQ KEMs, a signature scheme and DVS; Hashimoto et al.'s [HKKP22] protocol HKKP based on PQ KEMs, a signature scheme and RS; a baseline protocol made only with PQ KEMs and a signature scheme similar to the *non-deniable* variant of HKKP; and our protocol K-Waay based on FrodoKEX+ as a PQ split-KEM, PQ KEMs and a signature scheme.[9]

We chose Kyber-512 as the KEM, both Falcon-512 and Dilithium2 for signatures, and Raptor for ring signatures. We implemented both HKKP and SPQR with signed prekeys as is the case in Signal's implementation of X3DH. That is, a PQ signature key pair is part of the long-term key, and ephemeral keys uploaded to server are signed with it. Note that this is make explicit in K-Waay as the ephemeral keys are signed with the long-term one. The authors of HKKP show that this is not necessary in their protocol, however not doing so weakens perfect forward secrecy.

We built the different protocols in C using the `liboqs` library[10] for Kyber, Falcon, and Dilithium, the Raptor implementation provided by the authors[11], and a modified version of the `lwe-frodo` library[12] with scaled parameters to properly simulate FrodoKEX+. More precisely, the modulus was set to the first power of 2 larger than the modulus in FrodoKEX+, the addition of the noise during decapsulation was also added, and the noise distributions were modified to match the ones of FrodoKEX+. We did not optimise the scheme in any way (e.g. by using AVX instructions or parallelisation) and we leave this as future work. For the sake of completeness, we also provide a reference implementation of FrodoKEX+ in Rust[13] for the interested reader. All benchmarks were run on a virtual machine running Ubuntu 22.04 with 2 cores of an Intel i7-9750H running at 2.60GHz and 4GB of RAM allocated.

[9] SPQR and HKKP do not formally treat (regular) signatures, but we include them for fair comparison and because a practical system would use signatures for prekey bundles.
[10] https://github.com/open-quantum-safe/liboqs
[11] https://github.com/zhenfeizhang/raptor
[12] https://github.com/lwe-frodo/lwe-frodo
[13] https://github.com/lehugueni/frodokexp-rust

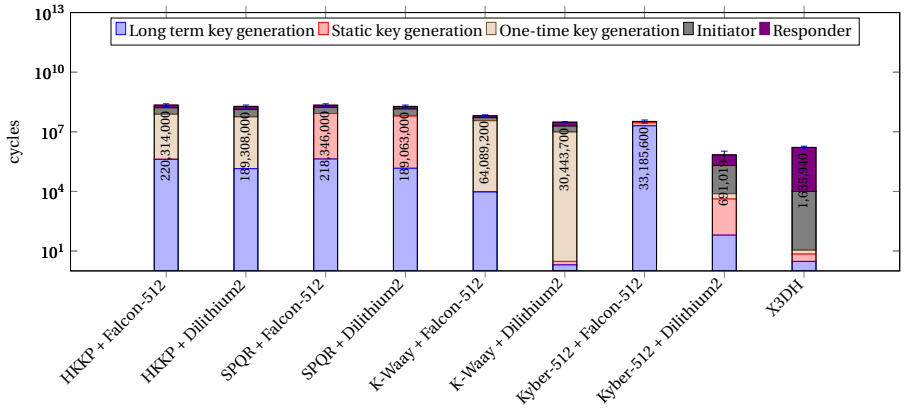

Figure 3.15: Speed benchmark for X3DH protocols.

Speed. For the speed benchmark, we measured how many cycles each protocol takes in one execution. We summarise our results in a logarithmic graph on Figure 3.15 (note that the internal division of the bars is linear).

Fixing the choice of KEM and signature scheme, our protocol K-Waay, depending on the choice of KEM and signature scheme, is between 3 and 6 times faster than the previous proposals even with our relatively conservative parameter choice. In our protocol K-Waay using Dilithium2, most cycles are spent in the ephemeral key generation, while using Falcon makes the static key generation as expensive as the ephemeral key one. Overall, one can see that Falcon, while more compact than Dilithium2, has a great impact on efficiency. For instance, K-Waay with Dilithium2 is faster than the non-deniable scheme using Kyber and Falcon.

Apart from Falcon, we see that the most time-consuming primitives are the non-standard ones, i.e., ring signatures and split-KEM. Hence, we see that the KEM+SIG protocol (HKKP's baseline proposal) performs even better than X3DH, which shows once again that lattice-based schemes can be faster than their classical counterparts. However, we recall that it does not provide deniability. At last, we note that X3DH is the only construction that spends more time in sending and receiving than generating keys. Our protocol's Send and Receive (i.e., BatchReceive with a single input message) procedures are very fast.

Data Size. In Table 3.5, we provide for each scheme the size of the long-term keys, the prekeys (output by Init in our DAKE syntax), and the ciphertext output by the sender. We computed for both HKKP and SPQR the size with and without long-term signatures. We see that K-Waay compares well in term of long-term public key and ciphertext size as both are smaller than signed HKKP and SPQR. However, the prekeys are much larger as one could expect from a LWE-based scheme and due to our conservative setting of parameters.

Scheme	||pk|	|prek|	|ct|
K-Waay + Dilithium	2112	24520	1632
K-Waay + Falcon	1697	22790	1632
HKKP [HKKP22]	1700	1700	4056
HKKP [HKKP22] + Dilithium2	3012	4120	4056
HKKP [HKKP22] + Falcon	2597	2390	4056
SPQR [BFG+22b]	3400	1632	4824
SPQR [BFG+22b] + Dilithium2	4712	4052	4824
SPQR [BFG+22b] + Falcon	4297	2322	4824

Table 3.5: Size comparison in bytes between K-Waay instantiated with FrodoKEX+, HKKP [HKKP22] and SPQR [BFG+22b]. We also computed the sizes for both HKKP and SPQR implemented with signed prekey bundles.

3.6.2 Advantages, Limitations and Discussion

Running Out of Ephemeral Keys. The main disadvantage of our protocol is that running out of ephemeral keys requires the receiver to abort if *any* of the sessions that used the same prekey is bogus. If this happens, then a malicious party could mount some kind of denial of service (DoS) attack against the user that was offline for too long by sending a bogus split-KEM ciphertext. There is an obvious trade-off between the risk of such an attack happening and the number of ephemeral keys uploaded on the server (and thus also storage). We leave the analysis and the mitigation of such a threat as future work, but we believe that if a reasonable amount of prekeys are uploaded, creating fake accounts is difficult (e.g., by requiring a phone number as in Signal), and/or users are online often enough, such an attack would be difficult to mount. Furthermore, several practical mitigations are possible. For instance, if the receiver (i.e., the victim) received a bogus ciphertext among the n ciphertexts sent for the same prekey while offline, they can restart K-Waay with each the n parties but as the initiator, which will probably succeed. The victim could also send n new prekeys to the n initiators directly, making sure the protocol will succeed at the next iteration. This would make the attack less useful as it could only *delay* communication and not prevent it.

We also think it is worth mentioning that the trick we propose might be easy to misimplement. In particular, it is crucial that no information about which split-KEM ciphertext failed leaks if such a situation occurs. That is, precautions should be taken such that leakage via side-channels in the scope of the system designer's threat model are prevented.

Split-KEM Instead of Ring Signatures. The fact that we use a primitive similar to a post-quantum KEM allows us to leverage the extensive literature on the topic and existing safe/optimised implementations. This also gives good security guarantees as post-quantum KEMs have been heavily scrutinised as part of the NIST standardisation process. For example, as mentioned above, our proposed lattice-based implementation is based on a key-exchange variant of FrodoKEM, which is itself the PQ KEM recommended by the German Federal Office for Information Security (BSI) [fIS23]. Overall we think that a split-KEM such as FrodoKEX+

is more mature and closer to being usable in practice than ring signatures.

On the Necessity of Modifying FrodoKEX. Currently, our split-KEM significantly differs from the original FrodoKEX in two aspects: (i) the modulus for our construction has to be prime in order for our reduction from Extended-LWE to LWE to hold[14], and (ii) we have to introduce additional masking terms to prove UNF-1KCA security. However, we believe that both changes are artefacts of the security proofs, and the original FrodoKEX split-KEM should be (up to a reasonable security loss) deniable.

There are alternative reduction techniques from Extended-LWE to LWE in the literature [BJRW21, BLP+13], which do not rely on having an odd modulus at the cost of using discrete Gaussian error distributions with large parameters. It is thus an interesting research problem to efficiently reduce Extended-LWE to LWE for even modulus with small reduction loss. In practice, the most efficient LWE attacks do not consider the structure of the modulus, so intuitively this should translate to the Extended-LWE setting[15].

As for our second main modification, it is not immediately clear how to argue UNF-1KCA security without the additional masking terms. Hence, we leave deniability and UNF-1KCA security of the original FrodoKEX construction as exciting future work.

Deniability. While the signature on the ephemeral public keys might give the impression that our protocol is less deniable than X3DH or previous PQ alternatives, this is actually not the case. The reason is that prekey bundles in these protocols are signed as well, but this detail is abstracted away in the analysis (i.e., it is assumed that all parties have received and authenticated all public keys before the protocol actually starts). While this kind of analysis allows for strong deniability claims, in practice these protocols do not satisfy something stronger than some kind of peer-deniability. The exception is the ring signature based variant by Hashimoto et al. [HKKP22], where the prekey bundle is not necessarily signed. However, in this variant, the authors prove the security of their protocol in a weaker model than their non-deniable, signature-based protocol (i.e., it satisfies a weaker notion of forward secrecy). Overall, if deniability should not come at the price of security, peer-deniability seems like the best notion one can achieve in these DAKEs.

We wished to provide a transparent model for peer-deniability, where the upload of signed ephemeral keys is made explicit. We also strengthen the deniability definition of Brendel et al. [BFG+22b] by allowing the exposure of one of the parties (i.e., the receiving one, which would be the malicious party trying to frame the sender). While our protocol satisfies our stronger (in term of key exposure) notion of deniability, we believe both previous PQ X3DH alternatives satisfy it as well. Indeed, in these schemes, the ephemeral keys are KEM and RS keys only, which are deniable. Hence, exposing these should not harm deniability.

[14] Recall FrodoKEX [BCD+16] uses a power-of-two modulus for efficiency.

[15] Recently, various frameworks have been developed [DDGR20, DSGHK23], which measure concrete hardness of LWE given hints of specific form, such as linear combination of secrets with short random coefficients.

Hashimoto et al. [HKKP22] consider a strong notion of deniability where the adversary is malicious (i.e., can arbitrarily deviate from the protocol) and show how to modify HKKP such that it is secure against such a threat. However, such deniability comes at the expense of NIZKs, which are complex, expensive and are not always proven secure in the QROM when random oracles are used. Moreover, as in other deniable systems against malicious adversaries, non-falsifiable assumptions (i.e., knowledge-type assumptions) are required to prove the security. In addition, it seems difficult to defend against adversaries actively trying to frame a given user in messaging in practice [GPA19, CCHD23]; for example, an adversary could also simply ask questions that would identify the victim with good probability. Because of these reasons, we do not consider such a notion of deniability here.

To contextualise our results, we remark here that cryptographic deniability, which is targeted in this chapter and all previous work on deniable X3DH key exchange, translates to deniability on a *system* level only if the application preserves deniability. For example, Collins et al. [CCHD23] observe that Signal as currently deployed does not provide this kind of 'practical' deniability for ordinary users. Suppose Bob is trying to frame Alice and hands over their phone that contains a transcript of communication between Alice and Bob to a judge. Because Signal authenticates users (either directly or indirectly through Signal sealed sender [Lun18]), unless Bob was able to modify their phone (which depends on the technical expertise of Bob), the judge can deduce that the conversation plausibly took place as in the transcript, regardless of the cryptographic protocols employed underneath. It is interesting future work to further explore deniability on the broader system level and practical deniability [RMA+23, YGS23].

An Optimisation. As presented in Section 3.4, the K-Waay protocol generates a signature for each ephemeral public key uploaded. This can easily be optimised by signing the whole prekey bundle containing several ephemeral keys. This way, the server needs to store only one PQ signature for each user. The downside is that now each user needs to download the whole bundle to verify the signature. This offers a trade-off between data stored at the server and sent to clients.

4 On Active Attack Detection in Messaging with Immediate Decryption

In this chapter, we explore different trade-offs between security and performance for active attack detection in two-party messaging when messages can be delivered in a different order to what they were sent in by parties (i.e., messaging with immediate decryption). An extended abstract corresponding to this work appeared at CRYPTO 2023. The work presented in this chapter is the result of a close collaboration with Simone Colombo, noting that Loïs Huguenin-Dumittan was the primary contributor for the lower bound result which we include for completeness, with further contributions from Loïs, Khashayar Barooti and Serge Vaudenay. A full version of this work can be found on the Cryptology ePrint Archive [BCC+23b].

4.1 Contribution

The asynchronicity of messaging and the unreliability of networks has driven the design of ratcheting-based protocols with *immediate decryption* [ACD19, PP22, BRT23, CZ24], i.e., the support of out-of-order delivery and message loss on the *protocol* level. This property ensures that legitimate messages can be immediately decrypted by the receiver upon arrival and placed correctly among other received messages. Furthermore, communication can continue even if some messages are permanently lost. As highlighted in Chapter 1, the Signal protocol, the current de-facto messaging standard, supports immediate decryption. By contrast, many schemes in the literature fail if even a single message is lost (see [BSJ+17, PR18, DV19, CDV21] for a non-exhaustive list).

The aforementioned security notions do not guarantee message authentication when the adversary impersonates parties, e.g., through state compromise. The lack of authentication implies that parties cannot *detect* active attacks. A recent phishing attack against Signal's phone number verification service enabled attackers to re-register accounts to another device, demonstrating the practicality of impersonation attacks via secret state compromise [Sup22]. Similar attacks that steal verification codes to hijack accounts affect a plethora of messaging applications. The proliferation of spyware such as Pegasus represents an additional—and worrying—threat for secret exfiltration [SRCM+22].

The most widely used mechanisms for detecting active attacks use an *out-of-band* authenticated channel. All deployed mechanisms we are aware in practice [Mar17] and many proposed in the literature [DH21, DGP22] assume such a channel. Solutions like Signal's safety numbers [Mar17] enables parties to authenticate their long-term keys by comparing QR codes in person. However, as observed by Dowling and Hale [DH20, DH21], Signal's approach fails to provide guarantees after a user's state is exposed, since the safety numbers protocol only authenticates the session key output from X3DH.

To remedy this situation, Dowling and Hale [DH21] proposed to add an additional authentication key to each iteration of Signal's asymmetric ratchet for on-demand use in out-of-band authentication. Their construction allows parties to immediately—that is, without additional communication rounds—authenticate their entire *asymmetric ratchet* out of band. However messages forged under symmetric keys will never be detected. The only other construction in the literature to our knowledge, proposed by Dowling, Günter and Poirrier [DGP22], requires *three rounds* of in-band communication before an out-of-band hash comparison can take place. Contrary to Dowling and Hale's solution, this approach authenticates all messages (albeit does not formally treat out-of-order messages), but imposes additional rounds, which is especially problematic in the presence of an active adversary. This raises our first research question:

1. Can we authenticate *all* messages in a *single round* of out-of-band communication to detect active attacks in the immediate decryption setting?

Out-of-band authentication is not always practical or even possible. A convenient alternative is to detect active attacks *in-band*, i.e., using the same channel as the messaging protocol. The adversary can, in the worst case, block all messages sent by honest parties, thereby forcing users to resort to out-of-band communication, but mounting such a persistent attack requires considerable resources. Durak and Vaudenay [DV19] thus introduced RECOVER security: if a party receives a forgery, then this party does not accept subsequent messages sent honestly by his counterpart. Caforio et al. [CDV21] then extended RECOVER security to enable a party to detect whether their *partner* was compromised, i.e., whether they received a forgery. By contrast to out-of-band authentication, no additional messages are required to support attack detection: in-band ciphertexts contain the authentication information. However, these notions and the corresponding constructions assume in-order message delivery and fail on message dropping. This raises a second question, first suggested by Alwen et al. [ACD19]:

2. Can we achieve extended RECOVER security—immediate in-band active attack detection—while supporting immediate decryption?

To detect active attacks, parties need to authenticate their *entire* message history: each message may be a forgery, i.e., the result of an active attack. With immediate decryption, parties cannot be sure which messages their partner has received until they receive an honest reply

from them. Intuitively, each message needs to "contain" the message history up until when it was sent. Looking ahead, we formalise and confirm this intuition, which motivates the exploration of performance/security trade-offs and optimisations. In this regard, existing protocols for both in-band and out-of-band active attack detection represent only a subset of the potential design space. Consequently we also ask:

3. What are the *communication costs* of in- and out-of-band active attack detection for messaging with immediate decryption, and what useful performance/security *trade-offs* can be made?

4.1.1 Summary

In this chapter, we explore the aforementioned questions. In more detail:

- We introduce (Section 4.2) a primitive that we call *authenticated ratcheted communication*, which captures immediate decryption and models communication through both insecure in-band and authentic out-of-band channels.
- In Section 4.3, we formalise in-band active attack detection for immediate decryption, by extending RECOVER security [DV19, CDV21], with two notions, namely r-RID and s-RID security, for detecting active attacks towards the receiver and on reception of messages from the sender after they were attacked; combined, these two notions comprise RID (recover with immediate decryption) security. We propose a scheme secure under these notions.
- We consider *out-of-band* active attack detection for immediate decryption messaging in Section 4.4. We introduce notions r-UNF and s-UNF (which combine to UNF for unforgeable), which are analogous to the notions for in-band detection. Demonstrating their similarity, we construct a UNF-secure scheme from a RID-secure scheme. We also construct a UNF-secure ARC scheme given a RC scheme.
- In Section 4.5, we prove with an information-theoretic argument that ciphertexts in a scheme satisfying either r-RID or r-UNF security must grow *linearly* in the number of messages sent, implying that attaching the set of sent ciphertexts to each message as we do is essentially optimal.
- In Section 4.6, we consider different ways to bypass the aforementioned lower bounds. First, we discuss ways to optimise the s-RID-secure scheme. We show how one can drastically reduce the overhead as long as the communication between the two parties is relatively synchronised. We believe the corresponding scheme is the most practical that we propose. We also explore optimising the *authenticated* out-of-band channel construction, by considering pruning-based optimisations, where parties securely prune messages as soon as they are authenticated. We finally propose a new three-move authentication protocol and a corresponding security notion.

4.1.2 Technical Overview

We assume a network where parties communicate over two types of channels: insecure channels and out-of-band authenticated channels. The adversary has full control over insecure channels. In particular, it can read, deliver, modify and delay messages. Over the authenticated channels, the integrity and authenticity of the messages are protected, that is, the adversary can read, deliver, duplicate and delay messages but not modify them. In the Signal application, the insecure channel is the usual network, whereas the out-of-band channel is that which is used for safety number verification [Mar17], typically in-person.

(Authenticated) Ratcheted Communication. We introduce a syntax for ratcheted communication (RC) in which sent and received messages are associated with totally ordered *ordinals* (epoch/index pairs in formalisations of the Double Ratchet [ACD19]). Ordinals enable our protocol to support *immediate decryption* [ACD19], i.e., message loss and re-ordering on the network. We build on this syntax to define *authenticated ratcheted communication*, or ARC, which comprises two additional functions AuthSend and AuthReceive. A party can use AuthSend to send an authentication tag through the out-of-band channel that the counterpart processes with AuthReceive. AuthSend outputs an ordinal that is at least as large as the last *sent* ordinal for that party. AuthReceive, if successful, should authenticate all messages up to that ordinal; this is captured in UNF security. Our secondary correctness notion ORDINALS enforces these semantics even in presence of forgeries.

RID **Security.** We revisit the definitions of RECOVER security [DV19, CDV21] in the immediate decryption setting. We define two complementary security notions for RID security:

- r-RID ensures that the receiver of a forgery does not accept honest messages with ordinals *larger* than that of the forgery.
- s-RID security enables a party to detect if their counterpart has ever received a forgery (i.e., a forgery from the sender).

If a scheme is both r-RID- and s-RID-secure, then it is RID-secure. These notions are orthogonal to forward security and post-compromise security and we allow the adversary to have full access to each party's secret states and control their randomness.

We propose a construction that transforms any ratcheted communication scheme into a provably RID-secure one. In the construction, both parties locally keep track of messages they have sent and received. Every time they send a message, they attach *all* messages (i.e., the ciphertexts from the underlying RC) they have sent and received so far to their ciphertext. When a party receives a message that "contradicts" what it has sent or received, it can deduce that an active attack took place.

To reduce communication costs, parties send ordinals and hashes of messages, instead of

complete ciphertexts. For r-RID security, a receiver $\overline{\mathsf{P}}$ compares the input message and the supposed set of sent messages contained inside it with what $\overline{\mathsf{P}}$ has received previously. For s-RID, it suffices for a receiver $\overline{\mathsf{P}}$ (who knows exactly what it sent) to check whether the sender claims to have received anything that $\overline{\mathsf{P}}$ did not send. Here, P only needs to send a *single* hash alongside the set of received ordinals (which are generally smaller than hashes), since $\overline{\mathsf{P}}$ can recompute the hash locally. Since the channel is insecure, parties need to perform a series of checks on the ciphertexts to prevent the adversary from tampering with the sets of sent and received messages sent. Both r-RID and s-RID security rely on the collision resistance of the hash function.

UNF **Security.** We define notions analogous to r-RID and s-RID for *authenticated* ratcheted communication schemes. The r-UNF (receiver unforgeable) notion ensures that a party does not accept authentication tags after receiving a forgery, whereas s-UNF (sender unforgeable) ensures that a party does not accept authentication tags coming from a counterpart that received a forgery. We show that a RID-secure scheme can be turned into a UNF-secure scheme. The transformation highlights the similarity between RID and UNF security. In the former, parties authenticate all messages they have sent and received in band, whereas in the latter the messages are authenticated out of band. Concretely, the transformation uses the ciphertext of a RID-secure RC scheme as the authentication tag for an ARC scheme, which, intuitively, moves authentication material to the out-of-band channel. We also provide a direct construction that is analogous to our construction for RID security, except the performance hit here is only taken when sending and receiving authentication tags, rather than for every (in-band) message.

Lower Bounds. We prove a linear lower bound on the ciphertext size of any r-RID-secure RC: assuming unidirectional communication, each ciphertext must capture all information contained in previously sent ones. In fact, the security notion requires that the receiving party is able to immediately detect if *any* subset of previous ciphertexts contains a forgery, since (1) the sender does not know what has been received and (2) ciphertexts can be arbitrarily re-ordered or dropped.

For the proof, we construct an (inefficient) encoder/decoder pair for a list of input messages $(m_1,\ldots,m_n)$ and randomness R, that uses the r-RID RC to compress the input. More precisely, the encoder uses the RC to send messages $(m_1,\ldots,m_n)$ with randomness R to produce a list of ciphertexts $(\mathsf{ct}_1,\ldots,\mathsf{ct}_n)$, and outputs only (ct_n,R). The decoder uses the RC to receive ciphertext ct_n, generates every possible ct_1 for all possible messages m_1 with randomness R and attempts to successfully receive one of them. If this succeeds, it means m_1 was the same message as the one input to the encoder (i.e., the "honest" one). Then, the decoder continues with m_2 and so on, eventually outputting $(m_1,\ldots,m_n,R)$. Finally, by setting the distributions of the messages and randomness as uniform[1], one can argue by Shannon's theorem that the ciphertext space must be exponentially large in $n\cdot|m_i|$. The formal proof is actually more complicated as the r-RID security does not need to be perfect and many ciphertexts might be

[1] A comparable argument can be made instead in terms of the *entropy* of the messages.

successfully received when decoding.

Then, following a nearly identical proof, we prove a linear lower bound on the authentication tag size of any r-UNF-secure ARC. These proofs might be of independent interest.

Practical Active Attack Detection. We explore how to overcome the linear communication complexity that r-RID and r-UNF impose. We first observe that ciphertexts can be much smaller to achieve s-RID or s-UNF security. As noted above, it suffices for P to send a single hash of all received messages with the corresponding ordinals, since partner $\overline{\mathsf{P}}$ can recompute the hash if it stores all messages it sends. Assuming each ordinal uses c space, ciphertexts reduce in size from $O(n(\lambda+c))$ to $O(\lambda+nc)$ given P has received n messages.

We propose an optimised protocol that achieves s-RID security. In the protocol, parties keep track of epochs [ACD19]. Party A starts with $\mathsf{ep}=0$ and B with $\mathsf{ep}=1$. While sending each message parties attach the ep alongside. If $\mathsf{ep}_{\mathsf{A}}=t$, A does not accept any messages with $\mathsf{ep}>t+1$ and if A receives a message with $\mathsf{ep}=t+1$ they update their ep value $t+2$. The observation is that epoch values only increase when both parties have received a message. Using this fact, it can be shown that it suffices to convey information about the messages received in the last *two epochs* to provide s-RID security. Thus, the number of ordinals required in each message depends on the communication pattern: in practice, it should usually suffice to only send a few ordinals and a hash digest achieve s-RID security. To see why the optimisation works, if an honest message was sent from A to B after A received a forged message, either this forgery was received in the last 2 epochs, or there was another forgery and honest message pair after it, as otherwise the ep values would be out of sync.

We observe that parties achieve r-RID/r-UNF-like guarantees after one round of honest communication from s-RID/s-UNF security. If P detects that their partner $\overline{\mathsf{P}}$ has received a forgery, P can let $\overline{\mathsf{P}}$ know, and thus $\overline{\mathsf{P}}$ can learn that they have received a forgery (which is what r-RID/r-UNF guarantee). We also formalise this by proposing a lightweight three-move protocol over the out-of-band channel and a corresponding security model which captures bidirectional message authentication. Participant P (resp. $\overline{\mathsf{P}}$) sends their set of received messages to their partner in the first and second moves. In the second and third moves, $\overline{\mathsf{P}}$ (resp. P) sends a bit that indicates whether the set of received messages was consistent with what they actually sent.

Furthermore, for UNF security, we note that the authentication tags can be compressed over time by including acknowledgements in tags. Since the out-of-band channel is authentic, parties are sure that the authentication information—that is, the sets of sent and received messages—coming from the counterpart is correct. This enables parties to prune already authenticated messages.

4.1.3 Additional Related Work

A growing line of work considers the performance and security of messaging in both the two-party [BSJ+17, PR18, JS18, DV19, CDV21, BRV20] and more general group settings [ACDT20, ACDT21a, AJM22] settings. Some of these works provide similar [JS18] and sometimes weaker [JMM19a] guarantees for in-band active attack detection assuming in-order communication. To our knowledge, in-band active attack detection has not yet been explicitly explored in group messaging, but schemes like MLS ensure that if the state of two parties is forked then their states become incompatible, in some protocol-specific sense.

Naor et al. [NRS20] introduced the concept of immediate key delivery for round-based group key exchange: if some parties go offline, the remaining ones should be able to complete it and successfully output a shared secret. This property is related but distinct from immediate decryption in this work as it focuses on keys instead of messages.

Apart from Durak and Vaudenay and Caforio et al. who introduced the RECOVER notions, Dowling et al. [DHRR22] ensure r-RECOVER, but not s-RECOVER security via signatures, while providing anonymity guarantees even upon state exposure. Dowling et al. [DGP22] frame their authentication guarantees as follows: if no long-term keys are compromised, then all messages exchanged are authentic. Otherwise, active attacks can be detected out-of-band. They achieve this by signing all messages with long-term keys. Our protocols and security notions can be adapted to achieve these guarantees. In distributed computing, the problem is formalised in terms of accountability, which enables parties to detect faulty (Byzantine) nodes [HKD07, CGG+22]. In multi-party computation, a line of work has explored security with *identifiable abort* [IOZ14] which ensures that if parties fail to compute a given function, they can identify the party that caused the failure.

The encoder/decoder technique that we use to prove the lower bounds in Section 4.5 have been used before in cryptography [LN18, JLN19]. While the basic idea is the same, the technical details of the proofs are not comparable as the primitives are different. Related work in group messaging achieves communication lower bounds in symbolic models of execution [BDR20, ANPPP23] and in a black-box impossibility setting [BDG+22].

4.2 (Authenticated) Ratcheted Communication

In this section we introduce the *ratcheted communication* (RC) cryptographic primitive and an extension *authenticated ratcheted communication* (ARC) supporting out-of-band authentication. These primitives augment classic ratcheting-based secure messaging schemes [JMM19a, ACD19, CDV21] in two ways: (1) sent and received messages are associated with *ordinals*, and, for ARC, (2) the syntax encompasses two additional stateful algorithms AuthSend and AuthReceive.

Ordinals associated with messages enable a party to (1) order incoming messages, which is

essential in the immediate decryption setting; (2) keep track of how many messages have been communicated; and (3) infer which messages have been authenticated using the out-of-band channel. Ordinals of the form num can be elements of any set on which a total order is defined. In Alwen et al.'s [ACD19] and Bienstock et al.'s [BFG+22a] modelling of Signal's Double Ratchet protocol, an ordinal num is defined as a pair of integers (e, c) such that $(e, c) < (e', c')$ if $e < e'$ or both $e = e'$ and $c < c'$. We formally define an RC scheme below.

Definition 33 (Ratcheted communication (RC))**.** A *ratcheted communication* (RC) scheme comprises the following efficient algorithms:

- $\mathsf{pp} \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$: The setup algorithm takes as input the security parameter $\lambda \in \mathbb{N}$, expressed in unary, and outputs public parameters pp.
- $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z) \xleftarrow{\$} \mathsf{Init}(\mathsf{pp})$: The initialisation algorithm takes as input public parameters pp and outputs a state st_P for $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$, and public information z.
- $(\mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$: The send algorithm takes as inputs state st_P, associated data ad and a plaintext pt and outputs a new state st'_P, an ordinal num and ciphertext ct.
- $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{pt}) \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$: The receive algorithm takes as inputs a state st_P, associated data ad and ciphertext ct and outputs an acceptance bit $\mathsf{acc} \in \{\mathsf{true}, \mathsf{false}\}$, state st'_P, ordinal num and plaintext pt.

The Receive algorithm returns dummy st'_P, num, pt which are ignored when $\mathsf{acc} = \mathsf{false}$.

The Double Ratchet. Signal's Double Ratchet protocol can be viewed as an RC. In the work of Alwen et al. [ACD19], a secure messaging scheme consists of an initialisation algorithm and party-specific Send and Receive algorithms with no associated data. The Receive algorithms, but not the Send algorithms, output an epoch/index pair $(e, i) \in \mathbb{N}^2$ which plays the role of an ordinal. The Double Ratchet as modelled by Alwen et al. [ACD19] can thus be considered an RC by modifying its Send algorithm to output each (e, i) pair as an ordinal and enforcing that $\mathsf{ad} = \perp$ is always input to Send and Receive.

Authenticated Ratcheted Communication (ARC). In an ARC, parties rely on stateful AuthSend and AuthReceive algorithms to authenticate the communication using a (possibly narrowband) out-of-band authenticated channel. AuthSend outputs an authentication tag and an ordinal, whereas AuthReceive takes an authentication tag as input and outputs an authentication bit and an ordinal. Intuitively, the authentication tag is sent via the out-of-band authenticated channel and it enables the receiver to detect active attacks using the AuthReceive algorithm. Participants can decide when to invoke the algorithms and thus use the authentication tag on-demand, e.g., when an out-of-band channel is available.

AuthSend and AuthReceive outputs ordinals with the same semantics as Send and Receive. Namely, the num that AuthSend outputs is greater or equal to the last num that Send outputs;

besides ordering authentication tags with respect to messages the party has sent or received, the ordinal indicates which messages (all up until num) the authentication tag authenticates. Similarly, for AuthReceive, the ordinal num indicates that all messages with $\mathsf{num}' \leq \mathsf{num}$ have been authenticated with the received tag.

Definition 34 (Authenticated ratcheted communication (ARC))**.** An *authenticated ratcheted communication* (ARC) scheme comprises the following efficient algorithms:

- Setup, Init, Send, Receive are defined as in RC (Definition 33).
- $(\mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{at}) \xleftarrow{\$} \mathsf{AuthSend}(\mathsf{st}_\mathsf{P})$: The authenticated send function takes as input a state st_P and outputs a new state st'_P, an ordinal num and an authentication tag at.
- $(\mathsf{auth}, \mathsf{st}'_\mathsf{P}, \mathsf{num}) \leftarrow \mathsf{AuthReceive}(\mathsf{st}_\mathsf{P}, \mathsf{at})$: The authenticated receiving function takes as inputs state st_P and authentication tag at and outputs an authentication bit $\mathsf{auth} \in \{\mathsf{true}, \mathsf{false}\}$, an updated state st'_P and an ordinal num.

The AuthReceive algorithm returns dummy st'_P, num which the scheme ignores when auth = false.

One could alternatively define AuthSend/AuthReceive to output sets of ordinals corresponding to which messages have been authenticated, rather than single ordinals. Our security notions ensure that this information can be efficiently computed by parties using the ordinals that the algorithms output.

Correctness. We define *correctness* for an RC and ARC scheme with the CORRECT game presented in Figure 4.1. The game takes a security parameter and a schedule sched as inputs. We use a schedule to model the message flow between the participants, which can (1) send a message, (2) receive a message, and for ARC only, (3) send an authentication tag, or (4) receive a sent authentication tag. More precisely, sched is an ordered list of instructions of the form $(\mathsf{P}, \text{"send"}, \mathsf{ad}, \mathsf{pt})$, $(\mathsf{P}, \text{"rec"}, j)$, and for ARC only, $(\mathsf{P}, \text{"authsend"})$, or $(\mathsf{P}, \text{"authrec"}, j)$, where $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$, ad denotes associated data, pt denotes a plaintext, and $j \in \mathbb{N}$ indicates either the $(\mathsf{ad}, \mathsf{ct})$ pair or the at to be received—that is, to be processed by Receive or AuthReceive respectively.

A correct (A)RC scheme must recover the correct plaintext from the corresponding associated data/ciphertext pair. Moreover, the scheme must satisfy the following properties.

- Subsequent calls to the Send algorithm outputs strictly increasing ordinals (line 6 in Figure 4.1).[2]
- Ordinals are equal for corresponding calls to Send (resp. AuthSend for ARC) and Receive (resp. AuthReceive for ARC) (lines 11 and 21).

[2]This could instead require Send to output strictly increasing ordinals w.r.t. Send *and* Receive calls made by P, which is satisfied in Alwen et al.'s work [ACD19], but we opted against this for generality's sake.

Game $\mathsf{CORRECT}_{\mathsf{RC}}(\mathsf{sched})$

1 : $\mathsf{pp} \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$; $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z) \xleftarrow{\$} \mathsf{Init}(\mathsf{pp})$
2 : $\mathsf{ad}_*[\cdot], \mathsf{pt}_*[\cdot], \mathsf{ct}_*[\cdot], \mathsf{at}_*[\cdot] \leftarrow \bot$; $\mathsf{received}[\cdot], \mathsf{sent}[\cdot] \leftarrow \mathsf{false}$; $\mathsf{sent\text{-}num}_* \leftarrow \bot$
3 : **for** $i = 1$ **to** $\mathsf{length}(\mathsf{sched})$:
4 : **if** $\mathsf{sched}[i]$ parses as $(\mathsf{P}, \text{"send"}, \mathsf{ad}, \mathsf{pt})$ for some $\mathsf{ad}, \mathsf{pt}, \mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$:
5 : $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$
6 : **if** $i > 1 \wedge \mathsf{num} \leq \mathsf{sent\text{-}num}_\mathsf{P}$: **return** 1
7 : $\mathsf{sent}[i] \leftarrow \mathsf{true}$; $\mathsf{ad}_\mathsf{P}[i] \leftarrow \mathsf{ad}$; $\mathsf{pt}_\mathsf{P}[i] \leftarrow (\mathsf{num}, \mathsf{pt})$; $\mathsf{ct}_\mathsf{P}[i] \leftarrow \mathsf{ct}$; $\mathsf{sent\text{-}num}_\mathsf{P} \leftarrow \mathsf{num}$
8 : **elseif** $\mathsf{sched}[i]$ parses as $(\mathsf{P}, \text{"rec"}, j)$ for some $j \in \mathbb{N}, \mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$:
9 : **if** $\neg\mathsf{sent}[j] \vee \mathsf{received}[j] \vee \mathsf{at}_{\overline{\mathsf{P}}}[j] \neq \bot$: **continue**
10 : $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{pt}) \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}_{\overline{\mathsf{P}}}[j], \mathsf{ct}_{\overline{\mathsf{P}}}[j])$
11 : **if** $\neg\mathsf{acc} \vee ((\mathsf{num}, \mathsf{pt}) \neq \mathsf{pt}_{\overline{\mathsf{P}}}[j])$: **return** 1
12 : $\mathsf{received}[j] \leftarrow \mathsf{acc}$; $\mathsf{st}_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
13 : **elseif** $\mathsf{sched}[i]$ parses as $(\mathsf{P}, \text{"authsend"})$ for some $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$:
14 : $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{at}) \xleftarrow{\$} \mathsf{AuthSend}(\mathsf{st}_\mathsf{P})$
15 : **if** $\mathsf{num} < \mathsf{sent\text{-}num}_\mathsf{P}$: **return** 1
16 : $\mathsf{sent}[i] \leftarrow \mathsf{true}$; $\mathsf{at}_\mathsf{P}[i] \leftarrow (\mathsf{num}, \mathsf{at})$
17 : **elseif** $\mathsf{sched}[i]$ parses as $(\mathsf{P}, \text{"authrec"}, j)$ for some $j \in \mathbb{N}, \mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$:
18 : **if** $\neg\mathsf{sent}[j] \vee \mathsf{received}[j] \vee \mathsf{at}_{\overline{\mathsf{P}}}[j] = \bot$: **continue**
19 : $(\mathsf{num}_{\overline{\mathsf{P}}}, \mathsf{at}_{\overline{\mathsf{P}}}) \leftarrow \mathsf{at}_{\overline{\mathsf{P}}}[j]$
20 : $(\mathsf{auth}, \mathsf{st}'_\mathsf{P}, \mathsf{num}) \leftarrow \mathsf{AuthReceive}(\mathsf{st}_\mathsf{P}, \mathsf{at}_{\overline{\mathsf{P}}})$
21 : **if** $\neg\mathsf{auth} \vee \mathsf{num} \neq \mathsf{num}_{\overline{\mathsf{P}}}$: **return** 1
22 : $\mathsf{received}[j] \leftarrow \mathsf{true}$; $\mathsf{st}_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
23 : **return** 0

Figure 4.1: Correctness game for an RC/ARC scheme RC. Highlighted statements are only executed for an ARC scheme.

- For ARC, AuthSend outputs an ordinal greater or equal to the ordinal returned by the last call to Send (line 15).

We require that these properties hold even when forgeries are received by one or both parties, and enforce them in the ORDINALS game presented in Figure 4.3. We also capture these properties for clarity in our correctness game.

We formally define correctness for an (A)RC scheme in Definition 35 below.

Definition 35 (CORRECT)**.** Consider the correctness game CORRECT presented in Figure 4.1. An RC (resp. ARC) scheme RC is *correct* if, for all $\lambda \in \mathbb{N}$, and all sequences of the form sched with elements of the form $(\mathsf{P}, \text{"send"}, \mathsf{ad}, \mathsf{pt})$, $(\mathsf{P}, \text{"rec"}, j)$, (resp. also of the form $(\mathsf{P}, \text{"authsend"})$,

$(\mathsf{P}, \text{"authrec"}, j))$, for $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$, we have

$$\Pr[\mathsf{CORRECT}_{\mathsf{RC}}(\mathsf{sched}) \Rightarrow 1] = 0 \,.$$

Correctness states that AuthSend must output an ordinal greater or equal to the ordinal that the last call to Send returned. If AuthSend does not increase the ordinal, then it is clear which messages are authenticated; if the ordinal increases in AuthSend, the application designer must keep track of the last num that Send returned to infer what the tag authenticates. Nonetheless, the latter case may be desirable to ensure that all ordinals output by Send and AuthSend are distinct.

Oracles. Our security notions for RC and ARC build on a common set of oracles, introduced in Figure 4.2. The SEND (resp. RECEIVE) oracle enables the adversary to send (resp. receive) a message on behalf of a party P. In SEND, the caller can specify the randomness used by Send or let the challenger sample randomness uniformly. For ARC, AUTHSEND enables the adversary to send an authentication tag on behalf of a party P, whereas AUTHRECEIVE handles AuthReceive. The oracles $\mathsf{EXP}_{\mathsf{pt}}(j)$ and $\mathsf{EXP}_{\mathsf{st}}(j)$ expose plaintexts and states to the adversary, respectively.

The oracles of Figure 4.2 model a communication network composed of insecure in-band and authentic out-of-band channels. The SEND and RECEIVE oracles enable the adversary to read, deliver, modify and delay messages, but AUTHSEND and AUTHRECEIVE do not allow the modification of authentication tags.

We assume an *always-authentic* out-of-band channel. To our knowledge, all deployed solutions for out-of-band authentication and relevant literature [DH20, DGP22] assume this. One could conceivably define a stronger model where the out-of-band channel is authentic only in some cases, e.g., the tampering rate is bounded, or multiple out-of-band channels exist but the adversary can compromise only a subset of them.

Ordinals. For RC and ARC schemes, we require, even in the presence of an adversary that injects forgeries, that Send and Receive (as well as AuthSend and AuthReceive for ARC schemes) output correct ordinals. We consider these properties in CORRECT (Figure 4.1), but they must hold also in presence of forgeries. We formalise this notion with the ORDINALS game in Figure 4.3.

In this game the challenger verifies three predicates, which correspond to the conditions for correct ordinals presented above. In Definition 36 we formalise ORDINALS security for (A)RC schemes.

Definition 36 (ORDINALS)**.** Consider the ORDINALS game in Figure 4.3. We say that an (authenticated) ratcheted communication scheme RC is ORDINALS secure if, for all possibly

Oracle SEND(P, ad, pt, r)

1: $i \leftarrow i+1$
2: **if** $r = \varepsilon$: $r \xleftarrow{\$} \mathscr{R}$
3: $(st_P, num, ct) \leftarrow Send(st_P, ad, pt; r)$
4: $state[i] \leftarrow st_P$
5: $plaintext[i] \leftarrow pt$
6: $log[i] \leftarrow ("send", P, num, ad, ct)$
7: **return** (num, ct)

Oracle AUTHSEND(P)

1: $i \leftarrow i+1$
2: $(st_P, num, at) \xleftarrow{\$} AuthSend(st_P)$
3: $auth[(P, i)] \leftarrow at$
4: $state[i] \leftarrow st_P$
5: $log[i] \leftarrow ("authsend", P, num, at)$
6: **return** (num, at)

Oracle $EXP_{pt}(j)$

1: $i \leftarrow i+1$
2: $log[i] \leftarrow ("ptexp", j)$
3: **return** $plaintext[j]$

Oracle RECEIVE(P, ad, ct)

1: $(acc, st, num, pt) \leftarrow Receive(st_P, ad, ct)$
2: **if** $\neg acc$: **return** $\perp$
3: $i \leftarrow i+1$
4: $st_P \leftarrow st$; $state[i] \leftarrow st_P$
5: $plaintext[i] \leftarrow pt$
6: $log[i] \leftarrow ("rec", P, num, ad, ct)$
7: **return** num

Oracle AUTHRECEIVE(P, j)

1: $at \leftarrow auth[(\overline{P}, j)]$
2: **if** $at = \perp$: **return** $\perp$
3: $(auth, st, num) \leftarrow AuthReceive(st_P, at)$
4: **if** $\neg auth$: **return** $\perp$
5: $i \leftarrow i+1$
6: $st_P \leftarrow st$; $state[i] \leftarrow st_P$
7: $log[i] \leftarrow ("authrec", P, num, at)$
8: **return** num

Oracle $EXP_{st}(j)$

1: $i \leftarrow i+1$
2: $log[i] \leftarrow ("stexp", j)$
3: **return** $state[j]$

Figure 4.2: Oracles which use variables state, plaintext, log, auth, st_* and i, all initialised in games where the oracles are used. AUTHSEND and AUTHRECEIVE are only used when considering ARC.

unbounded adversaries $\mathscr{A}$ we have

$$\Pr[\mathsf{ORDINALS}_{\mathsf{RC}}(\mathscr{A}) \Rightarrow 1] = 0 .$$

The ORDINALS game in Figure 4.3 is not suited to the case where ordinals can be arbitrary and in particular collide between parties. Thus, a given party and their partner must be associated with disjoint ordinals: models of practical protocols like the Double Ratchet achieve this by associating one party with even epochs and the counterpart with odd epochs.

Game $\mathsf{ORDINALS}_{\mathsf{RC}}(\mathscr{A})$

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$; $(\mathsf{st}_A, \mathsf{st}_B, z) \leftarrow \mathsf{Init}(\mathsf{pp})$
$\mathsf{state}[\cdot], \mathsf{plaintext}[\cdot], \mathsf{log}[\cdot], \mathsf{auth}[\cdot], \mathsf{st}_* \leftarrow \bot$
$i \leftarrow 0$
$\mathscr{A}^{\mathsf{SEND},\mathsf{RECEIVE},\mathsf{EXP}_{\mathsf{pt}},\mathsf{EXP}_{\mathsf{st}},\mathsf{AUTHSEND},\mathsf{AUTHRECEIVE}}(\mathsf{pp}, z)$
if $\exists\, \mathsf{P}, \mathsf{num}, \mathsf{num}', \mathsf{ad}, \mathsf{ct}, x, y$ s.t.
  $\mathsf{not\text{-}increasing}(\mathsf{log}, \mathsf{P}, \mathsf{num}, \mathsf{num}', x, y) \vee \mathsf{different}(\mathsf{log}, \mathsf{P}, \mathsf{num}, \mathsf{num}', \mathsf{ad}, \mathsf{ct}, \mathsf{at}) \vee$
  $\mathsf{auth\text{-}monotonic}(\mathsf{log}, \mathsf{P}, \mathsf{num}', y)$:
    return 1
return 0

$\mathsf{different}(\mathsf{log}, \mathsf{P}, \mathsf{num}, \mathsf{num}', \mathsf{ad}, \mathsf{ct}, \mathsf{at})$

return $((\text{"send"}, \mathsf{P}, \mathsf{num}, \mathsf{ad}, \mathsf{ct}) \in \mathsf{log} \wedge (\text{"rec"}, \overline{\mathsf{P}}, \mathsf{num}', \mathsf{ad}, \mathsf{ct}) \in \mathsf{log}) \vee$
$((\text{"authsend"}, \mathsf{P}, \mathsf{num}, \mathsf{at}) \in \mathsf{log} \wedge (\text{"authrec"}, \overline{\mathsf{P}}, \mathsf{num}', \mathsf{at}) \in \mathsf{log})) \wedge (\mathsf{num} \neq \mathsf{num}')$

$\mathsf{not\text{-}increasing}(\mathsf{log}, \mathsf{P}, \mathsf{num}, \mathsf{num}', x, y)$

return $(\text{"send"}, \mathsf{P}, \mathsf{num}, \_, \_) = \mathsf{log}[x] \vee (\text{"rec"}, \mathsf{P}, \mathsf{num}, \_, \_) = \mathsf{log}[x]) \wedge$
  $(\text{"send"}, \mathsf{P}, \mathsf{num}', \_, \_) = \mathsf{log}[y] \wedge (0 < x < y) \wedge (\mathsf{num} \geq \mathsf{num}')$

$\mathsf{auth\text{-}monotonic}(\mathsf{log}, \mathsf{P}, \mathsf{num}', y)$

$\mathsf{num} \leftarrow \max\{\bot, \mathsf{num}'' : (\text{"send"}, \mathsf{P}, \mathsf{num}'', \_, \_) = \mathsf{log}[x] \wedge 0 < x < y\}$
return $(\text{"authsend"}, \mathsf{P}, \mathsf{num}', \_, \_) = \mathsf{log}[y] \wedge (\mathsf{num} > \mathsf{num}')$

Figure 4.3: ORDINALS game. Highlighted statements are only considered for an ARC.

4.3 In-Band Active Attack Detection: RID

In this section we consider *in-band* active attack detection in the immediate decryption setting.

Caforio et al. [CDV21] define RECOVER security, which encompasses both r-RECOVER security and s-RECOVER security, but their notions and constructions only support in-order message delivery. Intuitively, r-RECOVER security prevents a party from being able to deliver an honest message *after* delivering a forgery, and s-RECOVER security allows a party to detect and stop communication when their partner has delivered a forgery. We extend these notions to handle out-of-order message delivery by introducing r-RID and s-RID, which we present in Figure 4.4 and illustrate in Figure 4.5. Combined, these two properties ensure RID security. Note that these definitions are orthogonal to the usual forward and post-compromise security notions that the ratcheting literature considers [BSJ+17, ACD19].

The winning condition in RID consists of three predicates:

- forgery verifies whether a forgery was accepted by one of the participants. In the predicate, we denote the impersonated party as P and the recipient of the forgery as $\overline{\mathsf{P}}$.
- bad-$\overline{\mathsf{P}}$ checks whether the recipient of the forgery manages to detect the attack. This predicate corresponds to r-RID security.
- bad-P establishes whether P, i.e., the participant that the adversary impersonates to send the forgery, fails to detect the attack. Since $\overline{\mathsf{P}}$ is the recipient of the forgery, the detection of the attack by P relies on a ciphertext sent by $\overline{\mathsf{P}}$ and honestly delivered. This predicate corresponds to s-RID security.

The game imposes that if forgery returns true, then at least one between bad-P and bad-$\overline{\mathsf{P}}$ must return true for the adversary to win the game.

Definition 37 (RID)**.** An RC RC is r-RID (resp. s-RID/RID) if, for all efficient adversaries $\mathscr{A}$, we have:

$$\mathsf{Adv}^{\mathrm{r-rid}}_{\mathsf{RC}}(\mathscr{A}) = \Pr[\text{r-}\mathsf{RID}_{\mathsf{RC}}(\mathscr{A}) \Rightarrow 1] = \mathsf{negl}$$
$$(\text{resp. } \mathsf{Adv}^{\mathrm{s-rid}}_{\mathsf{RC}}(\mathscr{A}) = \Pr[\text{s-}\mathsf{RID}_{\mathsf{RC}}(\mathscr{A}) \Rightarrow 1] / \mathsf{Adv}^{\mathrm{rid}}_{\mathsf{RC}}(\mathscr{A}) = \Pr[\mathsf{RID}_{\mathsf{RC}}(\mathscr{A}) \Rightarrow 1] \text{ })$$

where game r-RID (resp. s-RID/RID) is defined in Figure 4.4.

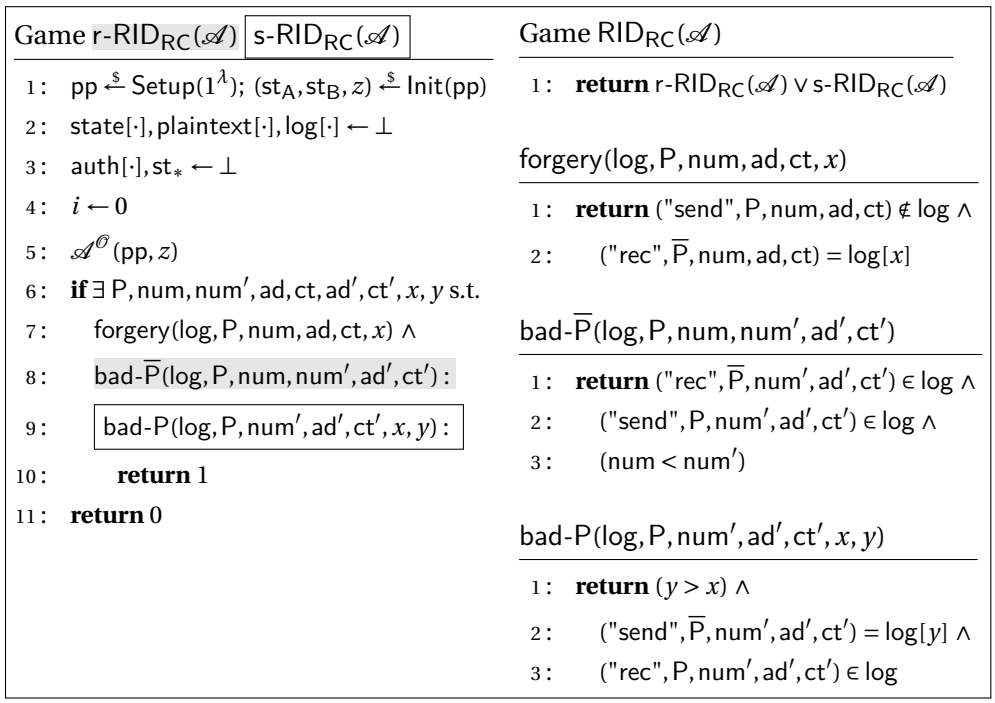

Figure 4.4: r-RID, s-RID and RID games for $\mathcal{O} = \{\mathsf{SEND}, \mathsf{RECEIVE}, \mathsf{EXP}_{\mathsf{pt}}, \mathsf{EXP}_{\mathsf{st}}\}$.

Although r-RID may seem stronger than s-RID at first glance, the two notions are not comparable. There exist schemes which provide r-RID and not s-RID security and vice versa, e.g.,

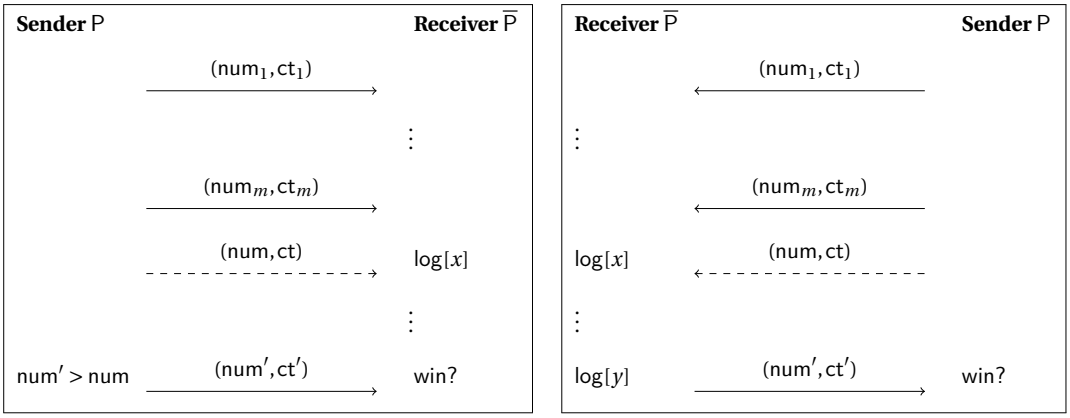

Figure 4.5: Visualising r-RID (left) and s-RID (right). Each figure showcases an adversary's winning condition in the respective game. The dashed arrows are forged messages. If $\overline{P}$ accepts the message at time "win?" then the adversary wins.

the scheme proposed in Figure 4.6 if the checks for either r-RID or s-RID are removed from the checks subroutine given the underlying RC is not r-RID or s-RID secure, respectively (the Double Ratchet is neither, for example).

However, we note the following link between the two notions. Suppose we use an s-RID scheme. This means that P is able to detect that $\overline{P}$ received a forged message. Then, if P sends an "abort" message to $\overline{P}$, $\overline{P}$ would be able to detect the forgery after *one* honest round-trip of messages. In other words, s-RID RC schemes can be transformed (by adding an "abort" message) into RC schemes with a weak variant of r-RID security: r-RID after a honest round-trip.

On Fine-Grained Security. Suppose A sends 5 messages with $num \in \{1,\ldots,5\}$, B receives a forgery with num = 1000, and then A sends 5 messages with $num \in \{6,\ldots,10\}$. If B never sends, i.e., A is the sender and B the receiver, RID-security only guarantees that the forgery might be detected when A sends the honest message with num′ = 1001 (observe the condition "num < num′" in predicate bad-$\overline{P}$ in Figure 4.4). Intuitively, B should be able to detect the forgery on receipt of the honest message with num = 6 since this message is "independent" of the forgery with num = 1000. By the not-increasing predicate of the ORDINALS security, all messages that A sends after one round-trip will have num > 1000, so such an attack will nevertheless be eventually detected. Fine-grained security capturing these scenarios can be formalised by tracking state exposures and message delivery timing at the cost of greater definitional complexity; we leave it open to do so. Some forgeries will be defeated by our construction below but it is likely required to leverage the security of the underlying RC to build a scheme providing this kind of security. Looking ahead, this remark also applies to UNF ARC schemes defined in Section 4.4.

4.3.1 RID-Secure RC

In this subsection we build a RID-secure RC scheme that relies on an ORDINALS-secure RC scheme and a collision-resistant hash function H (Definition 9). We present our transformation in Figure 4.6.

Scheme Description. Each party P keeps track of every message it has sent and received (in S and R, respectively). This information is communicated to $\overline{\mathsf{P}}$ every time P calls Send (via variables S and R′).

The Send procedure prepares the set R′, which contains the ordinals and a hash of all received messages (line 3). This step can be optimised by using an incremental hash function as we discuss in Section 4.6.1. Next, it calls RC.Send with input $(\mathsf{ad}', \mathsf{pt})$ where $\mathsf{ad}' = (\mathsf{ad}, \mathsf{S}, \mathsf{R}')$ is the associated data. The ciphertext ct contains both ct′ and sets S and R′. Finally, it adds the pair (num, h) to S (line 8), where the hash h is computed as $\mathsf{H.Eval}(\mathsf{st}_\mathsf{P}.\mathsf{hk}, (\mathsf{ad}, \mathsf{ct}))$, where $\mathsf{ct} = (\mathsf{ct}', \mathsf{S}, \mathsf{R}')$. Intuitively, (num, h) acts as a summary of P's state after calling RC.Send which can be checked by $\overline{\mathsf{P}}$ for inconsistency.

When $\overline{\mathsf{P}}$ invokes Receive, the procedure calls RC.Receive, which outputs $\mathsf{num} \neq \perp$ if the call is successful. Since ct contains $\mathsf{R}^{\overline{\mathsf{P}}}$, $\overline{\mathsf{P}}$ checks that what P received so far was correct (line 3 in checks). In addition, using the S set contained in the ciphertext ct, $\overline{\mathsf{P}}$ can further check whether the ciphertexts it received so far have indeed been sent by P. This is verified from lines 5 to 18 of checks. Some checks detect tampering of ct by the adversary (e.g. ct.S should not contain ordinals larger than the one of the current ciphertext, or if ct was sent earlier than another ciphertext already received, ct.S should be consistent with messages already acknowledged, etc.). If everything verifies, Receive stores (num, h) in R and adds ct.S to the set of acknowledged messages (lines 9 and 10).

Associated Data. The sets S and R′ included in the ciphertext are also included in the authenticated data passed to the underlying RC. This is actually not needed for RID security, but for authentication and confidentiality. Although we do not define these notions here, this should be done in practice and remains as future work to formalise.

On Errors. Note that our scheme outputs a generic error symbol $\perp$ in all cases. In particular, our construction outputs the same symbol regardless of whether the error was due to detecting an active attack, or from the situation where the adversary did not expose any states and simply sent a malformed ciphertext. The latter situation entails a denial of service attack vector if errors are treated the same way in both cases, so in practice (and in future work) they should be differentiated between.

Security Analysis. Correctness of $\mathsf{RC}_{\mathsf{RID}}$ follows from the correctness of the underlying RC scheme RC and the fact that the checks always outputs false when only honest messages are received. Similarly, ORDINALS-security follows from the ORDINALS security of RC, as $\mathsf{RC}_{\mathsf{RID}}$ outputs the num that RC outputs. As the next theorems state, the construction of Figure 4.6

$\mathsf{RC}_{\mathsf{RID}}.\mathsf{Setup}(1^\lambda)$

$\mathsf{pp}_0 \xleftarrow{\$} \mathsf{RC}.\mathsf{Setup}(1^\lambda)$
$\mathsf{hk} \xleftarrow{\$} \mathsf{H}.\mathsf{KGen}(1^\lambda)$
$\mathsf{hk}' \xleftarrow{\$} \mathsf{H}.\mathsf{KGen}(1^\lambda)$
$\mathsf{pp} \leftarrow (\mathsf{pp}_0, \mathsf{hk}, \mathsf{hk}')$
return pp

$\mathsf{RC}_{\mathsf{RID}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$

$(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{hk}', \mathsf{S}, \mathsf{R}, \_, \_) \leftarrow \mathsf{st}_\mathsf{P}$
$\mathsf{nums}' \leftarrow \{\mathsf{num}' : (\mathsf{num}', \_) \in \mathsf{R}\}$
$\mathsf{R}' \leftarrow (\mathsf{nums}', \mathsf{H}.\mathsf{Eval}(\mathsf{hk}', \mathsf{R}))$
$\mathsf{ad}' \leftarrow (\mathsf{ad}, \mathsf{S}, \mathsf{R}')$
$(\mathsf{st}_\mathsf{P}.\mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{ct}') \xleftarrow{\$} \mathsf{RC}.\mathsf{Send}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}', \mathsf{pt})$
$\mathsf{ct} \leftarrow (\mathsf{ct}', \mathsf{S}, \mathsf{R}')$
$h \leftarrow \mathsf{H}.\mathsf{Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))$
$\mathsf{st}_\mathsf{P}.\mathsf{S} \leftarrow \mathsf{S} \cup \{(\mathsf{num}, h)\}$
return $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{ct})$

$\mathsf{RC}_{\mathsf{RID}}.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$

$(\mathsf{ct}', \mathsf{S}^{\overline{\mathsf{P}}}, \mathsf{R}^{\overline{\mathsf{P}}}) \leftarrow \mathsf{ct}$
$(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \_, \_, \mathsf{R}, \mathsf{S}_{\mathsf{ack}}, \_) \leftarrow \mathsf{st}_\mathsf{P}$
$\mathsf{ad}' \leftarrow (\mathsf{ad}, \mathsf{S}^{\overline{\mathsf{P}}}, \mathsf{R}^{\overline{\mathsf{P}}})$
$(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{pt}) \leftarrow \mathsf{RC}.\mathsf{Receive}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}', \mathsf{ct}')$
if $\neg\mathsf{acc}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
$h \leftarrow \mathsf{H}.\mathsf{Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))$
if $\mathsf{checks}(\mathsf{st}_\mathsf{P}, \mathsf{ct}, h, \mathsf{num})$:
  return $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
$\mathsf{st}_\mathsf{P}.\mathsf{R} \leftarrow \mathsf{R} \cup \{(\mathsf{num}, h)\}$
$\mathsf{st}_\mathsf{P}.\mathsf{S}_{\mathsf{ack}} \leftarrow \mathsf{S}_{\mathsf{ack}} \cup \mathsf{S}^{\overline{\mathsf{P}}}$
$\mathsf{st}_\mathsf{P}.\mathsf{st}'_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
return $(\mathsf{acc}, \mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{pt})$

$\mathsf{RC}_{\mathsf{RID}}.\mathsf{Init}(\mathsf{pp})$

$(\mathsf{pp}_0, \mathsf{hk}, \mathsf{hk}') \leftarrow \mathsf{pp}$
$(\mathsf{st}'_\mathsf{A}, \mathsf{st}'_\mathsf{B}, z') \leftarrow \mathsf{RC}.\mathsf{Init}(\mathsf{pp}_0)$
$\mathsf{max\text{-}num} \leftarrow 0$
$\mathsf{S}, \mathsf{R}, \mathsf{S}_{\mathsf{ack}} \leftarrow \emptyset$
$\mathsf{st}_\mathsf{A} \leftarrow (\mathsf{st}'_\mathsf{A}, \mathsf{hk}, \mathsf{hk}', \mathsf{S}, \mathsf{R}, \mathsf{S}_{\mathsf{ack}}, \mathsf{max\text{-}num})$
$\mathsf{st}_\mathsf{B} \leftarrow (\mathsf{st}'_\mathsf{B}, \mathsf{hk}, \mathsf{hk}', \mathsf{S}, \mathsf{R}, \mathsf{S}_{\mathsf{ack}}, \mathsf{max\text{-}num})$
$z \leftarrow (z', \mathsf{pp})$
return $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z)$

$\mathsf{checks}(\mathsf{st}_\mathsf{P}, \mathsf{ct}, h, \mathsf{num})$

$(\mathsf{nums}', h') \leftarrow \mathsf{ct}.\mathsf{R}$
$\mathsf{R}^* \leftarrow \{(\mathsf{num}', \_) \in \mathsf{st}_\mathsf{P}.\mathsf{S} : \mathsf{num}' \in \mathsf{nums}')\}$
$\mathsf{s\text{-}bool} \leftarrow (\mathsf{H}.\mathsf{Eval}(\mathsf{st}_\mathsf{P}.\mathsf{hk}', \mathsf{R}^*) \neq h')$
$\mathsf{R}' \leftarrow \{(\mathsf{num}', \_) \in \mathsf{st}_\mathsf{P}.\mathsf{R} : \mathsf{num}' \leq \mathsf{num}\}$
$\mathsf{r\text{-}bool} \leftarrow (\mathsf{R}' \not\subseteq \mathsf{ct}.\mathsf{S})$
$\mathsf{r\text{-}bool} \leftarrow \mathsf{r\text{-}bool} \vee$
  $(\exists(\mathsf{num}^*, \_) \in \mathsf{ct}.\mathsf{S} : \mathsf{num}^* \geq \mathsf{num})$
if $\mathsf{num} < \mathsf{st}_\mathsf{P}.\mathsf{max\text{-}num}$:
  $\mathsf{r\text{-}bool} \leftarrow \mathsf{r\text{-}bool} \vee ((\mathsf{num}, h) \notin \mathsf{st}.\mathsf{S}_{\mathsf{ack}})$
  $\mathsf{r\text{-}bool} \leftarrow \mathsf{r\text{-}bool} \vee (\mathsf{ct}.\mathsf{S} \not\subseteq \mathsf{st}.\mathsf{S}_{\mathsf{ack}})$
  ${\mathsf{S}_{\mathsf{ack}}}' \leftarrow \{(\mathsf{num}', \_) \in \mathsf{st}_\mathsf{P}.\mathsf{S}_{\mathsf{ack}} :$
    $\mathsf{num}' < \mathsf{num}\}$
  $\mathsf{r\text{-}bool} \leftarrow \mathsf{r\text{-}bool} \vee ({\mathsf{S}_{\mathsf{ack}}}' \not\subseteq \mathsf{ct}.\mathsf{S})$
else :
  $\mathsf{st}_\mathsf{P}.\mathsf{max\text{-}num} \leftarrow \mathsf{num}$
  $\mathsf{r\text{-}bool} \leftarrow \mathsf{r\text{-}bool} \vee$
    $(\exists(\mathsf{num}', \_) \in \mathsf{st}.\mathsf{S}_{\mathsf{ack}} \setminus \mathsf{ct}.\mathsf{S} :$
      $\mathsf{num}' < \mathsf{st}_\mathsf{P}.\mathsf{max\text{-}num})$
return $\mathsf{r\text{-}bool} \vee \mathsf{s\text{-}bool}$

Figure 4.6: RID-secure RC scheme $\mathsf{RC}_{\mathsf{RID}}$ based on a RC scheme RC (Definition 33) and a hash function H (Definition 8). $\mathsf{RC}_{\mathsf{RID}}$ requires the following variables: max-num represents the largest received num; S is the set of (num, h) pairs; R is the set of received (num, h) pairs; $\mathsf{S}_{\mathsf{ack}}$ is the set of (num, h) which are expected to be received (according to the received ciphertext ct). All sets are append-only.

is r-RID-secure (Theorem 7) and s-RID-secure (Theorem 8). The construction is therefore RID-secure.

Theorem 7. Consider collision resistant hash function H used to build $\mathsf{RC}_{\mathsf{RID}}$ (defined in Figure 4.6). Then, we have that for every efficient adversary $\mathscr{A}$, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathrm{r-rid}}_{\mathsf{RC}_{\mathsf{RID}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{cr}}_{\mathsf{H}}(\mathscr{B}) .$$

Proof. Let us assume there exists an adversary $\tilde{\mathscr{A}}$ playing the r-RID game, running in time $\tilde{t}$ and making at most $\tilde{q}$ queries. Let us call the advantage of this adversary $\tilde{\epsilon}$, hence we have

$$\Pr[\text{r-RID}_{\mathsf{RC}_{\mathsf{RID}}}(\tilde{\mathscr{A}}) \Rightarrow 1] = \tilde{\epsilon}.$$

Let E be an event that occurs when $\text{r-RID}_{\mathsf{RC}_{\mathsf{RID}}}(\tilde{\mathscr{A}})$ outputs 1 The proof strategy is to construct an adversary $\mathscr{A}^*$, running in time $\approx \tilde{t}$ such that

$$\Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{A}^*) \Rightarrow 1 \mid E] = 1.$$

By definition of r-RID, E occurring means that there exist $\mathsf{P}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}), (\mathsf{num}', \mathsf{ad}', \mathsf{ct}'), x$ such that $\text{bad-}\overline{\mathsf{P}}(\mathsf{log}, \mathsf{P}, \mathsf{num}, \mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ and $\mathsf{forgery}(\mathsf{log}, \mathsf{P}, \mathsf{num}, \mathsf{ad}, \mathsf{ct}, x)$ are both true. This means that the message with ordinal num was not sent by P but received at some point by $\overline{\mathsf{P}}$ ($\mathsf{log}[x] = (\text{"rec"}, \overline{\mathsf{P}}, \mathsf{num}, \mathsf{ad}, \mathsf{ct})$), and the message with ordinal num' was also received and was actually sent by P. Moreover, $\mathsf{num} < \mathsf{num}'$.

We separate the two cases where 1) the message with ordinal num (the forged message) is received before the message with ordinal num' (the honest message), and 2) the message with ordinal num' is received first. We first analyse the former case.

We argue that unless the adversary found a collision, the message with ordinal num' (the honest message) would not have been delivered. Suppose that the honest message *was* delivered. Let st_{P} be the state of the receiver P while receiving message num', and $\mathsf{st}_{\overline{\mathsf{P}}}$ be the state of the sender $\overline{\mathsf{P}}$ while sending the message $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$. As $(\text{"rec"}, \mathsf{P}, \mathsf{num}', \mathsf{ad}', \mathsf{ct}') \in \mathsf{log}$, it means $\mathsf{RC}_{\mathsf{RID}}.\mathsf{Receive}(\mathsf{st}_{\mathsf{P}}, \mathsf{ad}', \mathsf{ct}') \rightarrow (\mathsf{true}, \mathsf{st}'_{\mathsf{P}}, \mathsf{num}', \mathsf{pt}')$, which implies $\mathsf{checks}(\mathsf{st}_{\mathsf{P}}, \mathsf{ct}', \mathsf{num}', \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}', \mathsf{ad}', \mathsf{ct}'))$ returned false.

Note that as $\mathsf{num} \leq \mathsf{num}'$, we have

$$(\mathsf{num}, \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))) \in \mathsf{R}' \text{ and } (\mathsf{num}, \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))) \in \mathsf{ct}'.\mathsf{S},$$

as otherwise r-bool would have been set to true in line 5. Let $h_f := \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))$. As $(\mathsf{num}, h_f) \in \mathsf{ct}'.\mathsf{S}$, we have

$$(\mathsf{num}, h_f) \in \mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{S} \tag{4.1}$$

as ct' was sent by $\overline{\mathsf{P}}$. This would mean that $\overline{\mathsf{P}}$ did send a message with ordinal num, let us call

it $(\mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h)$. Hence, we have that,

$$(\mathsf{H.Eval}(\mathsf{hk}, (\mathsf{ad}_h, \mathsf{ct}_h, \mathsf{num})), \mathsf{num}) \in \mathsf{st}_{\overline{P}}.\mathsf{S} \tag{4.2}$$

By combining (4.1) and (4.2) and the fact that num can appear only once in $\mathsf{st}_{\overline{\mathsf{P}}}$ (due to the ORDINALS security of $\mathsf{RC}_{\mathsf{RID}}$), we get that $\mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h)) = \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))$ which gives a collision. This is because $(\mathsf{ad}_h, \mathsf{ct}_h) \neq (\mathsf{ad}, \mathsf{ct})$ as $("\mathsf{send}", \overline{\mathsf{P}}, \mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h) \in \mathsf{log}$ and $("\mathsf{send}", \overline{\mathsf{P}}, \mathsf{num}, \mathsf{ad}, \mathsf{ct}) \notin \mathsf{log}$.

Now we discuss the case where $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ is received before $(\mathsf{num}, \mathsf{ad}, \mathsf{ct})$. As $\mathsf{num} \leq \mathsf{num}'$, this would mean that while receiving $(\mathsf{num}, \mathsf{ad}, \mathsf{ct})$, $\mathsf{max\text{-}num} \geq \mathsf{num}' \geq \mathsf{num}$. This would mean $(\mathsf{num}, h) \in \mathsf{st}_{\mathsf{P}}.\mathsf{S}_{\mathsf{ack}}$, otherwise the condition on line 9 would have not been satisfied. As $\mathsf{S}_{\mathsf{ack}}$ is only updated by adding the elements in $\mathsf{S}^{\overline{\mathsf{P}}}$ when a message is received, and as (num, h_f) is not in $\mathsf{ct}^*.\mathsf{S}$, for any honest ct^*, there should exist a forged message $(\mathsf{num}'', \mathsf{ad}'', \mathsf{ct}'')$ received before $(\mathsf{ad}, \mathsf{ct}, \mathsf{num})$ such that $(\mathsf{num}, h_f,) \in \mathsf{ct}''.\mathsf{S}$. As we considered $(\mathsf{num}, \mathsf{num}')$ to be the first pair of messages violating the r-RID property, we know that $\mathsf{num}'' > \mathsf{num}' > \mathsf{num}$.

We split the two cases where $(\mathsf{num}'', \mathsf{ad}'', \mathsf{ct}'')$ is received before $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ and the case where it is received after $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$. Let us consider the first case. We argue in this case $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ (the honest message) would not be accepted. As num'' is received before num', $\mathsf{num}' < \mathsf{num}'' \leq \mathsf{max\text{-}num}$. And as $\mathsf{r\text{-}bool} = \mathsf{false}$, $\mathsf{S}'_{\mathsf{ack}} \subseteq \mathsf{ct}'.\mathsf{S}$ (line 12). However $(\mathsf{num}, h_f) \in \mathsf{S}'_{\mathsf{ack}}$, as it was in $\mathsf{ct}''.\mathsf{S}$, hence it should also be in $\mathsf{ct}'.\mathsf{S}$, which would mean $h_f = \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_h, \mathsf{ct}_h, \mathsf{num})$ which is again a collision.

Now let us consider the case where $(\mathsf{num}'', \mathsf{ad}'', \mathsf{ct}'')$ is received after the message $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$. We argue that $(\mathsf{num}'', \mathsf{ad}'', \mathsf{ct}'')$ should not have been accepted. We split the cases where $\mathsf{num}'' \geq \mathsf{max\text{-}num}$ and $\mathsf{num}'' < \mathsf{max\text{-}num}$. Let us consider the later first. As $(\mathsf{ad}'', \mathsf{ct}'', \mathsf{num}'')$ was accepted, and hence $\mathsf{r\text{-}bool} = \mathsf{false}$, $\mathsf{ct}''.\mathsf{S} \subset \mathsf{S}_{\mathsf{ack}}$ (line 10). Now $(\mathsf{num}, h_f) \in \mathsf{ct}''.\mathsf{S}$, so $(\mathsf{num}, h_f) \in \mathsf{S}_{\mathsf{ack}}$. As without loss of generality we can imagine $(\mathsf{num}'', \mathsf{ad}'', \mathsf{ct}'')$ being the first message vouching for (h_f, num), this would mean (num, h_f) was added to $\mathsf{S}_{\mathsf{ack}}$ by an honest message, i.e. $h_f = h_h$ which leads to a collision again.

Finally for the case in which $\mathsf{num}'' \geq \mathsf{max\text{-}num}$, again, considering that $(\mathsf{num}'', \mathsf{ad}'', \mathsf{ct}'')$ is the first message vouching for (h_f, num), we have $(\mathsf{num}, h_f) \in \mathsf{S}_{\mathsf{ack}} \setminus \mathsf{ct}''.\mathsf{S}$ (and so r-bool would be set to true) unless $h_f = h_h$. Moreover, at this point $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ has already been received so, $\mathsf{max\text{-}num} \geq \mathsf{num}' > \mathsf{num}$. Hence, unless $h_f = h_h$, r-bool would be set to true in line 7. This concludes the proof that $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$, $(\mathsf{num}, \mathsf{ad}, \mathsf{ct})$ are accepted if and only if $\mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h)) = \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))$.

Now we describe the CR adversary $\mathscr{A}^*$. $\mathscr{A}^*$ runs the initialisation of $\mathsf{RC}_{\mathsf{RID}}$ by replacing the sampling step of hk with the hk given by the CR_{H} game, then runs $\tilde{\mathscr{A}}$ as a subroutine, and computes $("\mathsf{rec}", \mathsf{P}, \mathsf{num}, \mathsf{ad}_f, \mathsf{ct}_f) \in \mathsf{log}$, and $("\mathsf{send}", \overline{\mathsf{P}}, \mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h) \in \mathsf{log}$ such that $(\mathsf{ad}_f, \mathsf{ct}_f) \neq (\mathsf{ad}_h, \mathsf{ct}_h)$ and $h_f = h_h$ if possible. Given E, this pair always exists as we have $\Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{A}^*) \Rightarrow 1 \mid E] = 1$. Moreover, as $\mathscr{A}^*$ is just running $\tilde{\mathscr{A}}$ as a subroutine and not doing

anything extra, the time it runs is also $\approx \tilde{t}$. Finally, we have

$$\begin{aligned}\Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{A}^*) \Rightarrow 1] &\geq \Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{A}^*) \Rightarrow 1 \mid E] \cdot \Pr[E] \\ &= \Pr[\text{r-RID}_{\mathsf{RC}_{\mathsf{RID}}}(\tilde{\mathscr{A}}) \Rightarrow 1] = \tilde{\epsilon}\ .\end{aligned}$$

Hence, $\tilde{\epsilon} \leq \Pr[\mathsf{CR}_{\mathsf{H}}^{\mathscr{A}^*}(1^\lambda) \Rightarrow 1]$. □

Theorem 8. Consider collision resistant hash function H used to build $\mathsf{RC}_{\mathsf{RID}}$ (defined in Figure 4.6). Then, we have that for every efficient adversary $\mathscr{A}$, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\text{s-rid}}_{\mathsf{RC}_{\mathsf{RID}}}(\mathscr{A}) \leq \mathsf{Adv}^{\text{cr}}_{\mathsf{H}}(\mathscr{B})\ .$$

Proof. The proof strategy is essentially the same as the one taken for the proof of Theorem 7. For any adversary $\tilde{\mathscr{A}}$ playing the s-RID game, we construct an adversary $\mathscr{A}^*$ playing the CR game with comparable complexity. We first describe the adversary $\mathscr{A}^*$ in terms of $\tilde{\mathscr{A}}$ and proceed by proving that $\mathscr{A}^*$ wins at least as often as $\tilde{\mathscr{A}}$.

As with the previous proof we define an event E that occurs only when $\text{s-RID}_{\mathsf{RC}_{\mathsf{RID}}}(\tilde{\mathscr{A}}) \Rightarrow 1$, and we prove that $\Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{A}^*) \Rightarrow 1 \mid E] = 1$.

The event $\text{s-RID}_{\mathsf{RC}_{\mathsf{RID}}}(\tilde{\mathscr{A}}) \Rightarrow 1$, means there exists $\mathsf{P}, \mathsf{num}, \mathsf{ad}, \mathsf{ct}, \mathsf{num}', \mathsf{ad}', \mathsf{ct}', x, y$ such that $x < y$, $(\mathsf{num}, \mathsf{pt}, \mathsf{ct})$ is a forged message received by $\overline{\mathsf{P}}$ and logged in $\mathsf{log}[x]$, and $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ is an honest message sent by $\overline{\mathsf{P}}$ (logged in $\mathsf{log}[y]$) and received by P. As $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ was received, P's checks call returned false.

Let us define $h_f = \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))$. When receiving the forged message, $\overline{\mathsf{P}}$ adds (num, h_f) to $\mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{R}$. As $y > x$, (num, h_f) is in $\mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{R}$ when $\overline{\mathsf{P}}$ sends. Hence $\mathsf{num} \in \mathsf{nums}'$ for the honest message $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ sent by $\overline{\mathsf{P}}$. Now as $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ was accepted, we have, due to line 3 of checks, that $\mathsf{H.Eval}(\mathsf{hk}', \mathsf{R}^*) = \mathsf{H.Eval}(\mathsf{hk}', \mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{R})$.

If $\mathsf{R}^* \neq \mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{R}$ we have already found a collision. So let us assume that $\mathsf{R}^* = \mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{R}$. Now as $(\mathsf{num}, h_f) \in \mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{R}$, we also have that $(\mathsf{num}, h_f) \in \mathsf{R}^* \subset \mathsf{st}_{\mathsf{P}}.\mathsf{S}$.

This would mean that there exists an honest message $(\mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h)$ such that $\mathsf{H.Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h)) = h_f$. But note that as $(\mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h)$ is an honest message, $(\text{"send"}, \mathsf{P}, \mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h) \in \mathsf{log}$ but $(\text{"send"}, \mathsf{P}, \mathsf{num}, \mathsf{ad}, \mathsf{ct}) \notin \mathsf{log}$ as the message was forged, hence $(\mathsf{num}, \mathsf{ad}_h, \mathsf{ct}_h) \neq (\mathsf{num}, \mathsf{ad}, \mathsf{ct})$, which again yields a collision pair.

Now the CR adversary $\mathscr{A}^*$ does the following: they run the s-RID adversary $\tilde{\mathscr{A}}$ as a subroutine with the hk given by the CR challenger. They later find $\mathsf{P}, \mathsf{num}, \mathsf{ad}, \mathsf{ct}\ , \mathsf{num}', \mathsf{ad}', \mathsf{ct}', x, y$ satisfying the condition, in case they exist. Now (for example) by exposing the states of the parties once $(\mathsf{num}', \mathsf{ad}', \mathsf{ct}')$ was sent by $\overline{\mathsf{P}}$ they can find the collision pair described above. Hence we

have,

$$\Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{A}^*) \Rightarrow 1] \geq \Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{A}^*) \Rightarrow 1 \mid E] \cdot \Pr[E] = \Pr[\text{s-}\mathsf{RID}_{\mathsf{RC}_{\mathsf{RID}}}(\tilde{\mathscr{A}}) \Rightarrow 1] \tag{4.3}$$

Observe that the run-time of $\mathscr{A}^*$ is roughly the run-time of $\tilde{\mathscr{A}}$. This concludes the proof. □

Optimisation. The s-RID notion imposes less overhead than r-RID: the construction can be further optimised and still provide s-RID security. We describe an optimisation in Section 4.6.2 that allows for the set of ordinals in $\mathscr{R}$ of each party to be pruned each round trip of communication, which particularly improves communication complexity when parties are well-synchronised.

4.4 Out-Of-Band Active Attack Detection: UNF

In-band active attack detection is not always possible, as an adversary may block all honest messages sent by one or more parties. For example, modern messaging solutions in practice use a (possibly malicious) third party server to relay messages between participants, thereby introducing a single point of failure for in-band communication. This motivates us to consider out-of-band active attack detection, whereby parties can exchange authentication tags out-of-band, and to define *unforgeable* security (UNF).

An ARC scheme is *unforgeable* if, as soon as one of the two parties accepts a forgery, both parties can detect this out-of-band. We formalise this security notion through the UNF game (Figure 4.7), which, similarly to RID, encompasses r-UNF and s-UNF. The winning condition in UNF consists of three predicates: forgery, bad-$\overline{\mathsf{P}}$ (corresponding to r-UNF) and bad-P (corresponding to s-UNF) that are essentially the same as the predicates that we use to define RID security (Definition 37), except they rely on authentication tags instead of ciphertexts for forgery detection (and thus active attack detection is performed on-demand).

Definition 38 (UNF)**.** An ARC ARC is r-UNF (resp. s-UNF/UNF) if, for all efficient adversaries $\mathscr{A}$, we have:

$$\mathsf{Adv}^{\mathrm{r-unf}}_{\mathsf{ARC}}(\mathscr{A}) = \Pr[\text{r-}\mathsf{UNF}_{\mathsf{ARC}}(\mathscr{A}) \Rightarrow 1] = \mathsf{negl}$$
$$\text{(resp. } \mathsf{Adv}^{\mathrm{s-unf}}_{\mathsf{ARC}}(\mathscr{A}) = \Pr[\text{s-}\mathsf{UNF}_{\mathsf{ARC}}(\mathscr{A}) \Rightarrow 1] / \mathsf{Adv}^{\mathrm{unf}}_{\mathsf{ARC}}(\mathscr{A}) = \Pr[\mathsf{UNF}_{\mathsf{ARC}}(\mathscr{A}) \Rightarrow 1] \text{)}$$

where game r-UNF (resp. s-UNF/UNF) is defined in Figure 4.7.

As for RC schemes, we do not define message indistinguishability [DV19, ACD19] for ARC schemes. In our constructions, the schemes include in the authentication tag only *public* material, i.e., messages that have already been sent over the insecure channel. Since the adversary already has access to the entire transcript of the insecure channel, the authentication material should not give any additional advantage in a message indistinguishability game.

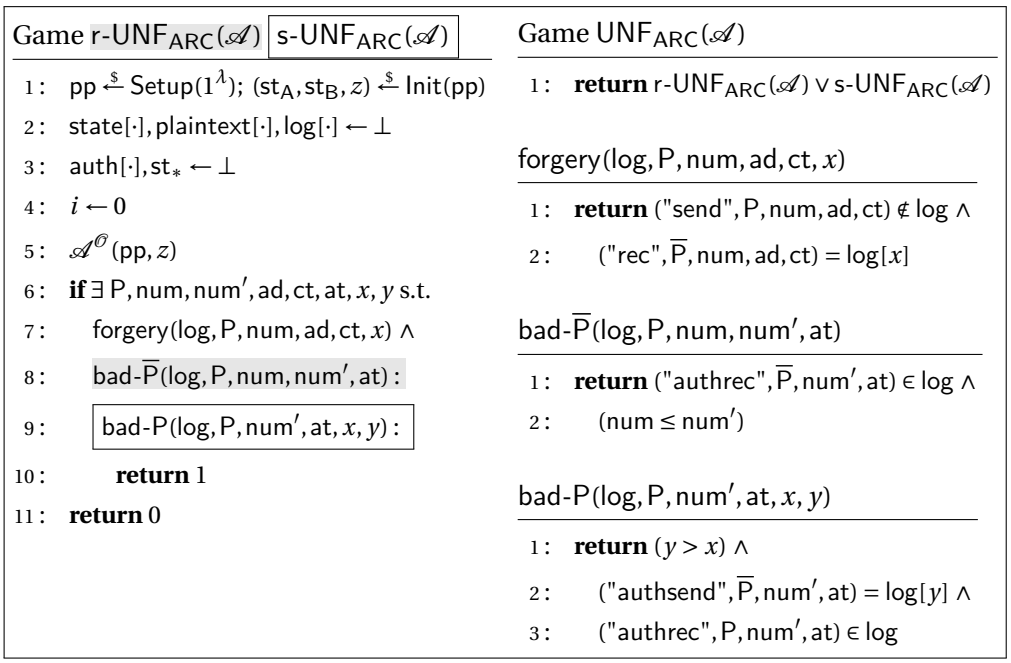

Figure 4.7: r-UNF, s-UNF and UNF games for $\mathscr{O}$ = {SEND, RECEIVE, EXP$_{pt}$, EXP$_{st}$, AUTHSEND, AUTHRECEIVE}.

4.4.1 UNF-Secure ARC from a RID-Secure RC

Highlighting the similarity between RID security and UNF security, we show in this subsection that one can directly use a RID-secure RC to build a UNF-secure ARC. The ARC scheme uses the Setup, Gen, Init, Send, Receive function of the RC. To send an authentication tag with AuthSend, the ARC scheme calls the Send function on a dummy message to generate a ciphertext ct that acts as the authentication tag. The function AuthReceive is then implemented as a Receive call on the authentication tag/ciphertext. The construction is detailed in Figure 4.8.

Then, we can show the following theorem, which also implies that the scheme of Figure 4.8 is r-UNF- and s-UNF-secure.

Theorem 9. Let RC$_{RID}$ be a RC scheme and ARC$_{UNF}$ be the ARC scheme built out of RC as shown in Figure 4.8. If RC$_{RID}$ is RID, ORDINALS-secure and correct, then ARC$_{UNF}$ is UNF-, ORDINALS-secure and correct.

Proof. Correctness follows from the correctness of the underlying RC$_{RID}$ and the use of domain separation for tags and ciphertexts.

We sketch a proof showing RID security of RC$_{RID}$ implies UNF security of ARC$_{UNF}$. For any adversary $\mathscr{A}$ playing the UNF game with ARC$_{UNF}$, we build a RID adversary $\mathscr{B}$ for RC$_{RID}$. Each query made by $\mathscr{A}$ to the oracles SEND, RECEIVE, EXP$_{pt}$, EXP$_{st}$ are forwarded by $\mathscr{B}$ to its own corresponding oracles (and domain separation is correctly implemented where needed).

$\mathsf{ARC}_{\mathsf{UNF}}.\mathsf{Setup}(1^\lambda)$

1 : **return** $\mathsf{RC}_{\mathsf{RID}}.\mathsf{Setup}(1^\lambda)$

$\mathsf{ARC}_{\mathsf{UNF}}.\mathsf{Init}(\mathsf{pp})$

1 : **return** $\mathsf{RC}_{\mathsf{RID}}.\mathsf{Init}(\mathsf{pp})$

$\mathsf{ARC}_{\mathsf{UNF}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$

1 : $\mathsf{ct}' \leftarrow \mathsf{RC}_{\mathsf{RID}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$
2 : $\mathsf{ct} \leftarrow (0, \mathsf{ct}')$
3 : **return** ct

$\mathsf{ARC}_{\mathsf{UNF}}.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$

1 : $(b, \mathsf{ct}') \leftarrow \mathsf{ct}$
2 : **if** $b \neq 0$: **return** $(\mathsf{false}, \bot, \bot, \bot)$
3 : **return** $\mathsf{RC}_{\mathsf{RID}}.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct}')$

$\mathsf{ARC}_{\mathsf{UNF}}.\mathsf{AuthSend}(\mathsf{st}_\mathsf{P})$

1 : $(\mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{ct}) \leftarrow \mathsf{RC}_{\mathsf{RID}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, 0, 0)$
2 : **return** $(\mathsf{st}'_\mathsf{P}, \mathsf{num}, (1, \mathsf{ct}))$

$\mathsf{ARC}_{\mathsf{UNF}}.\mathsf{AuthReceive}(\mathsf{st}_\mathsf{P}, \mathsf{at})$

1 : $(b, \mathsf{at}') \leftarrow \mathsf{at}$
2 : **if** $b \neq 1$: **return** $(\mathsf{false}, \bot, \bot)$
3 : $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{pt}) \leftarrow \mathsf{RC}_{\mathsf{RID}}.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, 0, \mathsf{at}')$
4 : **return** $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num})$

Figure 4.8: UNF-secure ARC scheme $\mathsf{ARC}_{\mathsf{UNF}}$ based on a RID-secure RC scheme $\mathsf{RC}_{\mathsf{RID}}$.

Queries of the form AUTHSEND(P) are simulated by $\mathscr{B}$ querying "$\mathsf{at}' \leftarrow \mathsf{SEND}(\mathsf{P}, 0, 0)$" and setting "$\mathsf{at} \leftarrow (1, \mathsf{at}')$", which perfectly simulates the generation of a tag in $\mathsf{ARC}_{\mathsf{UNF}}$. Finally, AUTHRECEIVE queries are simulated using the RECEIVE oracle on the tag/ciphertext. $\mathscr{B}$ can perfectly simulate the UNF game for $\mathscr{A}$.

Now, let us assume that the UNF adversary $\mathscr{A}$ wins with the forgery and bad-P predicates both evaluating to true. It means a forgery was received by a party P, then, later, that party sent a tag (i.e. a ciphertext in the RID game played by $\mathscr{B}$) that is honestly and successfully delivered to a party $\overline{\mathsf{P}}$. That implies that in the RID game played by $\mathscr{B}$, a party received a forgery, then sent a message that was successfully delivered, which is a winning condition for $\mathscr{B}$.

The second case is when the UNF adversary $\mathscr{A}$ wins with the forgery and bad-$\overline{\mathsf{P}}$ predicates both evaluating to true. This means that a forgery was received by a party P with ordinal num, then a tag with ordinal $\mathsf{num}' \geq \mathsf{num}$ was successfully received by $\overline{\mathsf{P}}$. Note that in our $\mathsf{ARC}_{\mathsf{UNF}}$ construction the tags are ciphertexts, thus the ordinals are strictly increasing, i.e., $\mathsf{num}' > \mathsf{num}$. Therefore, in the RID game played by $\mathscr{B}$, a forgery with ordinal num was received by P, then later a honest ciphertext with ordinal $\mathsf{num}' > \mathsf{num}$ was successfully delivered to P, making the bad-$\overline{\mathsf{P}}$ predicate in the RID game true.

Hence, for any adversary $\mathscr{A}$ winning the UNF game, there exists a RID adversary $\mathscr{B}$ that wins with at least the same probability.

Finally, ORDINALS-security follows from the ORDINALS security of $\mathsf{RC}_{\mathsf{RID}}$ and the fact that $\mathsf{ARC}_{\mathsf{UNF}}.\mathsf{Send}$ and $\mathsf{ARC}_{\mathsf{UNF}}.\mathsf{AuthSend}$ calls directly output num from $\mathsf{RC}_{\mathsf{RID}}.\mathsf{Send}$. □

4.4.2 UNF-Secure ARC from Any RC

We present an unoptimised UNF-secure ARC scheme $\mathsf{ARC}_{\mathsf{base}}$ given a RC scheme (Definition 33). We present our scheme in Figure 4.9. We later describe a communication-optimised scheme in Section 4.6.3.

$\mathsf{ARC}_{\mathsf{base}}.\mathsf{Setup}(1^\lambda)$

1 : $\mathsf{pp}_0 \xleftarrow{\$} \mathsf{RC.Setup}(1^\lambda)$; $\mathsf{hk} \xleftarrow{\$} \mathsf{H.KGen}(1^\lambda)$
2 : **return** $(\mathsf{pp}_0, \mathsf{hk})$

$\mathsf{ARC}_{\mathsf{base}}.\mathsf{Init}(\mathsf{pp})$

1 : $(\mathsf{pp}_0, \mathsf{hk}) \leftarrow \mathsf{pp}$
2 : $(\mathsf{st}'_\mathsf{A}, \mathsf{st}'_\mathsf{B}, z') \xleftarrow{\$} \mathsf{RC.Init}(\mathsf{pp}_0)$
3 : $\mathsf{num}, \mathsf{max\text{-}num} \leftarrow \bot$; $\mathsf{S}, \mathsf{R}, \mathsf{S}_{\mathsf{ack}} \leftarrow \emptyset$
4 : $\mathsf{st}_\mathsf{A} \leftarrow (\mathsf{st}'_\mathsf{A}, \mathsf{hk}, \mathsf{S}, \mathsf{R}, \mathsf{S}_{\mathsf{ack}}, \mathsf{num}, \mathsf{max\text{-}num})$
5 : $\mathsf{st}_\mathsf{B} \leftarrow (\mathsf{st}'_\mathsf{B}, \mathsf{hk}, \mathsf{S}, \mathsf{R}, \mathsf{S}_{\mathsf{ack}}, \mathsf{num}, \mathsf{max\text{-}num})$
6 : $z \leftarrow (z', \mathsf{pp})$
7 : **return** $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z)$

$\mathsf{ARC}_{\mathsf{base}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$

1 : $(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{S}, \_, \_, \_, \_) \leftarrow \mathsf{st}_\mathsf{P}$
2 : $(\mathsf{st}_\mathsf{P}.\mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{ct}) \xleftarrow{\$} \mathsf{RC.Send}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$
3 : $h \leftarrow \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{ad}, \mathsf{ct}))$
4 : $\mathsf{st}_\mathsf{P}.\mathsf{S} \leftarrow \mathsf{S} \cup \{(\mathsf{num}, h)\}$
5 : $\mathsf{st}_\mathsf{P}.\mathsf{num} \leftarrow \mathsf{num}$
6 : **return** $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{ct})$

$\mathsf{ARC}_{\mathsf{base}}.\mathsf{AuthSend}(\mathsf{st}_\mathsf{P})$

1 : $(\_, \_, \mathsf{S}, \mathsf{R}, \_, \mathsf{num}, \_) \leftarrow \mathsf{st}_\mathsf{P}$
2 : $\mathsf{at} \leftarrow (\mathsf{S}, \mathsf{R}, \mathsf{num})$
3 : **return** $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{at})$

$\mathsf{ARC}_{\mathsf{base}}.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$

1 : $(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \_, \mathsf{R}, \mathsf{S}_{\mathsf{ack}}, \_, \mathsf{max\text{-}num}) \leftarrow \mathsf{st}_\mathsf{P}$
2 : $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{pt}) \leftarrow \mathsf{RC.Receive}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$
3 : **if** $\neg\mathsf{acc}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
4 : $h \leftarrow \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{ad}, \mathsf{ct}))$
5 : **if** $\mathsf{num} \le \mathsf{max\text{-}num} \wedge (\mathsf{num}, h) \notin \mathsf{S}_{\mathsf{ack}}$:
6 : **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
7 : $\mathsf{st}_\mathsf{P}.\mathsf{R} \leftarrow \mathsf{R} \cup \{(\mathsf{num}, h)\}$
8 : $\mathsf{st}_\mathsf{P}.\mathsf{st}'_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
9 : **return** $(\mathsf{acc}, \mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{pt})$

$\mathsf{ARC}_{\mathsf{base}}.\mathsf{AuthReceive}(\mathsf{st}_\mathsf{P}, \mathsf{at})$

1 : $(\_, \_, \mathsf{S}, \mathsf{R}, \mathsf{S}_{\mathsf{ack}}, \mathsf{num}, \mathsf{max\text{-}num}) \leftarrow \mathsf{st}_\mathsf{P}$
2 : $(\mathsf{S}^{\overline{\mathsf{P}}}, \mathsf{R}^{\overline{\mathsf{P}}}, \mathsf{num}^{\overline{\mathsf{P}}}) \leftarrow \mathsf{at}$
3 : // $\overline{\mathsf{P}}$ received a forgery
4 : **if** $\mathsf{R}^{\overline{\mathsf{P}}} \not\subseteq \mathsf{S}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \mathsf{num})$
5 : $\mathsf{R}^{\mathsf{P}}_{\subseteq} \leftarrow \{(\mathsf{num}, \_) \in \mathsf{R} : \mathsf{num} \le \mathsf{num}^{\overline{\mathsf{P}}}\}$
6 : // P received a forgery
7 : **if** $\mathsf{R}^{\mathsf{P}}_{\subseteq} \not\subseteq \mathsf{S}^{\overline{\mathsf{P}}}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \mathsf{num})$
8 : $\mathsf{st}_\mathsf{P}.\mathsf{S}_{\mathsf{ack}} \leftarrow \mathsf{S}_{\mathsf{ack}} \cup \mathsf{S}^{\overline{\mathsf{P}}}$
9 : $\mathsf{st}_\mathsf{P}.\mathsf{max\text{-}num} \leftarrow \max\{\mathsf{max\text{-}num}, \mathsf{num}^{\overline{\mathsf{P}}}\}$
10 : **return** $(\mathsf{true}, \mathsf{st}_\mathsf{P}, \mathsf{num}^{\overline{\mathsf{P}}})$

Figure 4.9: UNF-secure ARC scheme $\mathsf{ARC}_{\mathsf{base}}$ based on a RC scheme RC (Definition 33). The scheme uses the following additional variables compared to RC: S is the set of sent values stored as (num, h); R is the set of received values stored as (num, h); $\mathsf{S}_{\mathsf{ack}}$ is the set of (num, h) expected to be received (according to the received authentication tag at); num is the ordinal corresponding to the party's last *sent* message; max-num represents the largest num received in an at. For simplicity of exposition, we omit the optimisation where R is sent as a single hash and n ordinals as done in Figure 4.6 for RID security.

Scheme Description. Procedures Send and Receive make use of the respective RC procedures. Send stores the hash of $(\mathsf{ad}, \mathsf{ct})$ for the message being sent, together with the num that the

underlying RC.Send algorithm outputs. A tuple composed of num and this hash is stored in a set S, which is in turn stored by the calling party. Send also updates ordinal num in the caller's state. The Receive procedure verifies if the RC.Receive algorithm accepts the inputs and that the received message is not a forgery on a previously authenticated message, which, as we will describe, is contained in $\mathsf{S}_{\mathsf{ack}}$. If both checks pass, Receive stores the hash of $(\mathsf{ad}, \mathsf{ct})$ together with the ordinal num returned by RC.Receive in a set R.

AuthSend includes in the authentication tag (at) the hashes of the caller's sent and received messages together with the last ordinal num returned by RC.Send. Since the adversary can reorder messages both in the normal channel and in the out-of-band channel, num indicates to the recipient of the authentication tag which messages they should compare against at. AuthReceive parses the authentication tag and checks whether the messages received by the counterpart are in the caller's local set S. Then, it verifies whether its local set of received messages, excluding the messages not encompassed by at, is a subset of the messages sent by the counterpart. If one of these conditions is not satisfied, then a forgery is detected. The sent messages authenticated by the counterpart are stored in a set $\mathsf{S}_{\mathsf{ack}}$. $\mathsf{ARC}_{\mathsf{base}}$.Receive uses this set to avoid forgeries on already authenticated ordinals.

The size of the authentication tags and the state of each party in the scheme of Figure 4.9 is linear in the number of sent and received messages. We show in Section 4.6.3 that this can be reduced by pruning. Messages can nevertheless be efficiently exchanged out-of-band in practice, e.g., using Bluetooth. Otherwise, parties can send authentication information over the insecure channel and authenticate it using the out-of-band channel by hashing and comparing digests [PV06].

Security Analysis. We now analyze the security properties of the scheme in Figure 4.9. Correctness of the scheme follows from the correctness of the underlying RC scheme. Similarly, ORDINALS security follows from the ORDINALS security of RC, as the scheme of Figure 4.9 outputs the same num that RC outputs.

The UNF-security of $\mathsf{ARC}_{\mathsf{base}}$ (Figure 4.9) is derived, as before, from the collision resistance of the hash function that the scheme uses, which we prove in what follows.

Theorem 10 (UNF security of $\mathsf{ARC}_{\mathsf{base}}$)**.** Consider collision resistant hash function H used to build $\mathsf{ARC}_{\mathsf{base}}$ (defined in Figure 4.9). Then, we have that for every efficient adversary $\mathscr{A}$, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathrm{unf}}_{\mathsf{ARC}_{\mathsf{base}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{cr}}_{\mathsf{H}}(\mathscr{B}) \ .$$

Proof. Assume an adversary $\mathscr{A}$ playing the UNF game (Figure 4.7), which makes at most q oracle queries and runs in time at most t. We assume the advantage of $\mathscr{A}$ is ϵ, hence by Definition 38 we have $\Pr[\mathsf{UNF}_{\mathsf{ARC}_{\mathsf{base}}}(\mathscr{A}) \Rightarrow 1] = \epsilon$. We construct an adversary $\mathscr{B}$, running in time approximately equal to t, which, running $\mathscr{A}$ as a subroutine, wins the collision resistance game for H (Definition 9), that is $\Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{B}^{\mathscr{A}}) \Rightarrow 1] = 1$.

The UNF adversary $\mathscr{A}$ wins when one party accepts a forgery (predicate forgery) and at least one of the two parties fails to detect the forgery (predicates bad-P and bad-$\overline{\mathsf{P}}$). Suppose there exist P, num, num′, ad, ct, at, x, y such that forgery(log, P, num, ad, ct, x) = true. We analyze the predicates bad-P and bad-$\overline{\mathsf{P}}$ separately, starting with the latter.

The forgery predicate states that ("send", P, num, ad′, ct′) ∈ log and ("rec", $\overline{\mathsf{P}}$, num, ad, ct) = log[x] for some $x \in \mathbb{N}$, where (ad′, ct′) ≠ (ad, ct). This means that (num, H.Eval(hk, (ad′, ct′)) ∈ S^{P} and (num, H.Eval(hk, (ad, ct)) ∈ $\mathsf{R}^{\overline{\mathsf{P}}}$, otherwise the forgery is trivially detected because (num, _) ∉ $\mathsf{R}^{\overline{\mathsf{P}}}$. Moreover, by the bad-$\overline{\mathsf{P}}$ predicate we know that $\mathsf{R}^{\overline{\mathsf{P}}}_{\subseteq} \subseteq \mathsf{S}^{\mathsf{P}}$ for any num ≤ num′, which implies that (num, H.Eval(hk, (ad′, ct′))) = (num, H.Eval(hk, (ad, ct))). By ORDINALS security, (num, ·) can appear only once in S^{P}, respectively in $\mathsf{R}^{\overline{\mathsf{P}}}$, and by assumption (ad′, ct′) ≠ (ad, ct), therefore we have a collision for H.Eval(hk, ·).

We now analyze the bad-P predicate. The forgery predicate states that ("send", P, num, ad′, ct′) ∈ log and ("rec", $\overline{\mathsf{P}}$, num, ad, ct) = log[x] for some $x \in \mathbb{N}$, where (ad′, ct′) ≠ (ad, ct), otherwise the forgery is trivially detected. This implies that (num, H.Eval(hk, (ad, ct)) ∈ $\mathsf{R}^{\overline{\mathsf{P}}}$ when $\mathsf{ARC}_{\mathsf{base}}$.Receive(_, ad, ct) → (true, _, num, _). By the bad-P predicate we know that $\overline{\mathsf{P}}$ sends an authentication tag at after accepting (ad, ct), since ("authsend", $\overline{\mathsf{P}}$, num′, at) = log[y] and $y > x$, which means that (num, H.Eval(hk, (ad, ct)) is in the $\mathsf{R}^{\overline{\mathsf{P}}}$ that at contains. The rest of the argument follows the same approach as the previous paragraph.

$\mathscr{B}^{\mathscr{A}}$(hk)

1: $(1^\lambda, \mathsf{hk}_0) \leftarrow \mathsf{hk}$; pp $\xleftarrow{\$}$ $\mathsf{ARC}_{\mathsf{base}}$.Setup$(1^\lambda)$; $(\mathsf{pp}_0, \mathsf{hk}') \leftarrow \mathsf{pp}$; $\mathsf{pp}' \leftarrow (\mathsf{pp}_0, \mathsf{hk})$
2: $(\mathsf{st}_A, \mathsf{st}_B, z) \xleftarrow{\$}$ $\mathsf{ARC}_{\mathsf{base}}$.Init(pp′)
3: state[·], plaintext[·], log[·], auth[·], $\mathsf{st}_* \leftarrow \bot$; $i \leftarrow 0$
4: $\mathscr{A}^{\mathscr{O}}$(pp, z)
5: **if** ∃ num, P, ad, ct, ad′, ct′ : ("send", P, num, ad, ct) ∈ log ∧
6: ("rec", $\overline{\mathsf{P}}$, num, ad′, ct′) ∧ (ad, ct) ≠ (ad′, ct′) :
7: **return** (ad, ct), (ad′, ct′)
8: **else abort**

Figure 4.10: CR adversary $\mathscr{B}$ where $\mathscr{O}$ = {SEND, RECEIVE, AUTHSEND, AUTHRECEIVE, $\mathsf{EXP}_{\mathsf{pt}}$, $\mathsf{EXP}_{\mathsf{st}}$} for the proof of Theorem 10.

We describe in Figure 4.10 the adversary $\mathscr{B}$ which plays the collision resistance game. $\mathscr{B}$ runs the $\mathsf{ARC}_{\mathsf{base}}$.Setup procedure and replaces the hash key hk′ that the procedure returns with the hk that the adversary receives from the CR challenger. After running $\mathscr{A}$ as a subroutine, $\mathscr{B}$ analyzes the log array to find the (ad, ct), (ad′, ct′) pairs that represents a forgery and returns those. If $\mathscr{A}$ wins the UNF game, then $\mathscr{B}$ wins the CR game, that is

$$\Pr[\mathsf{CR}_{\mathsf{H}}(\mathscr{B}) \Rightarrow 1] \geq \Pr[\mathsf{UNF}_{\mathsf{ARC}_{\mathsf{base}}}(\mathscr{A}) \Rightarrow 1] = \epsilon.$$

Moreover, $\mathscr{B}$ runs $\mathscr{A}$ as a subroutine and executes a negligible additional amount of work. □

4.5 Lower Bounds for Active Attack Detection

We study in this section the size of (1) ciphertexts of any r-RID-secure RC and (2) authentication tags of any r-UNF-secure ARC. In particular, all of our constructions so far (and hereafter) achieving one of these properties incurs (at least in the worst case) linear growth in the ciphertext or tag length in terms of the number of messages sent. We show here that one cannot hope for better by proving two lower bounds. More precisely, we show that the ciphertext space (resp. tag space) of a r-RID RC (resp. r-UNF ARC) grows exponentially in the number of messages sent. Note that we cannot prove a lower bound on the ciphertext size directly as it is always possible that *some* ciphertext is small. However, our bounds imply that at least n bits are required to represent any ciphertext or tag in their respective domain after the n-th message.

4.5.1 Communication Cost for r-RID Security

In what follows, we consider a RC that is perfectly correct:[3] for all randomness r, valid states st_P and associative data ad, the function $\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \cdot; \mathsf{r})$ mapping a plaintext to a ciphertext is injective.

The next theorem proves that the ciphertext size in a r-RID RC grows linearly in the number of messages sent (multiplied by either the security parameter or message size).

Theorem 11. Let Π be a perfectly correct RC, n_s and λ be fixed, and T_{λ,n_s} be the time complexity of the (efficient) adversary given on the left of Figure 4.12. In addition, let $\gamma \in \mathbb{Z}$ be such that, for all adversaries $\mathscr{A}$ running in at most time T_{λ,n_s} which query oracle SEND at most n_s times, we have

$$\Pr[\text{r-RID}_\Pi(\mathscr{A}) \Rightarrow 1] \leq \frac{1}{2^\gamma}.$$

Let $\mathscr{M} = \{0,1\}^n$ and $\mathscr{C} = \{0,1\}^k$ (without loss of generality) be the plaintext and ciphertext space associated to Π, respectively. Then,

$$k \geq n + (n_s - 1)(\gamma - 2), \text{ if } \gamma \leq n$$
$$k \geq 2 + n_s(n-2), \text{ if } \gamma > n.$$

A third lower bound gives

$$k \geq nn_s - \frac{1}{1 - \frac{2^n n_s}{2^\gamma}},$$

which is tighter for low values of n (e.g. $n = 1, 2$) and when $\gamma > n + \log(n_s)$.

[3] We discuss how to relax this requirement directly after our proof.

Proof. We show that if k is smaller than the given bounds, one can build an encoder/decoder for a uniform source such that the expected bit-length of a codeword is strictly lower than the entropy (i.e., the log of the size of the sampling set), contradicting Shannon's source coding theorem [Sha48].

More formally, we consider a source that samples uniformly at random from the set $\{0,1\}^{n\times n_s} \times \{0,1\}^r$, where r is the maximal number of bits (i.e., random coins) needed by the two procedures Setup and Init of Π and n_s invocations of Send. We present an encoder and decoder for such a source in Figure 4.11 (the non-boxed instructions in the encoder, and the decoder shown on the left). For the sake of explanation, assume that the RC used in the encoder/decoder has perfect r-RID security. Then, the sender sends n_s honestly generated ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_{n_s}$, and the receiver receives the last ciphertext ct_{n_s}. By perfect r-RID security, for any $i < n_s$, any ciphertext $\mathsf{ct}'_i \neq \mathsf{ct}_i$ should be rejected by the receiver. Thus, one can build an (inefficient) decoder that tests all ct'_i to find the correct one and recovers the corresponding message. In a sense, all ct_i must be encoded in the last ciphertext ct_{n_s}. The actual encoding is more complicated as if the r-RID security is not perfect, there will be a number of false positives (i.e., $\mathsf{ct}'_i \neq \mathsf{ct}_i$ but ct'_i is accepted by the receiver). Note that w.l.o.g., we omit the associated data throughout the proof (or assume $\mathsf{ad} = \bot$) as it plays no role.

Lemma 12. Our encoder (Figure 4.11) is perfectly correct, i.e.,

$$\Pr[\mathsf{Decode}(\mathsf{Encode}(m_1, \ldots, m_{n_s}, R)) = (m_1, \ldots, m_{n_s}, R)] = 1 \,.$$

Proof. The value R output by Decode is the same as the one input in Encode. Since the initial states depend only on R and Π is correct, ct_{n_s} will decrypt to m_{n_s}. The states st_A^i will be identical in both the encoding and decoding procedures as they are generated from st_A^{i-1}, the previously recovered message m_{i-1} and randomness R_i. This implies that the sets of accepting messages S_i will be the same as they depend only on st_B^1 and st_A^{i-1}. In addition, by the perfect correctness of Π, each message m_i will be in the corresponding set S_i. Hence, the decoder can recover each message m_i by reading S_i at the index given in the input. □

Lemma 13. Let C be the random variable corresponding to the codeword length output by Encode. In addition, let $F_i := S_i \setminus \{m_i\}$ be the set of false positives, where S_i and m_i are as in Encode. Then, $\mathbb{E}[C] \leq k + r + \sum_{i=1}^{n_s-1} 1 + \log(\mathbb{E}[|F_i|] + 1)$.

Proof. By construction, the encoder outputs a codeword of $k + r + \sum_{i=1}^{n_s-1} \lceil \log(|S_i|) \rceil$ bits. Therefore, we have

$$\mathbb{E}[C] = k + r + \sum_{i=1}^{n_s-1} \mathbb{E}[\lceil \log(1 + |F_i|) \rceil] \leq k + r + \sum_{i=1}^{n_s-1} 1 + \mathbb{E}[\log(1 + |F_i|)]$$

which is upper bounded by $k + r + \sum_{i=1}^{n_s-1} (1 + \log(\mathbb{E}[|F_i|] + 1))$, by the linearity of expectation and the definition of F_i, the fact that $\lceil x \rceil \leq 1 + x$, and Jensen's inequality applied consecutively. □

Encode$(m_1,\dots,m_{n_s},R)$

1: parse $(R_{-1},R_0,\dots,R_{n_s}) \leftarrow R$; $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda;R_{-1})$; $(\mathsf{st}_\mathsf{A}^0,\mathsf{st}_\mathsf{B}^0,z) \leftarrow \mathsf{Init}(\mathsf{pp};R_0)$
2: **for** $i \in \{1,\dots,n_s\}$ **do** // send the n_s messages
3: $(\mathsf{st}_\mathsf{A}^i,\mathsf{num},\mathsf{ct}_i) \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{A}^{i-1},m_i;R_i)$
4: $(\mathsf{acc},\mathsf{st}_\mathsf{B}^1,\mathsf{num},m'_{n_s}) \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{B}^0,\mathsf{ct}_{n_s})$ // Receive ct_{n_s}: $m'_{n_s} = m_{n_s}$ by perfect correctness
5: // Collecting false positives and correct messages:
6: **for** $i \in \{1,\dots,n_s-1\}$ **do**
7: $S_i \leftarrow \emptyset$
8: **for** $m \in \{0,1\}^n$ **do**
9: $(\_,\_,\mathsf{ct}') \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{A}^{i-1},m;R_i)$
10: $(\mathsf{acc},\_,\_,m') \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{B}^1,\mathsf{ct}')$
11: **if** acc:
12: $\boxed{\textbf{if } m \neq m_i : \textbf{return } (0,m_1,\dots,m_{n_s},R)}$
13: $S_i \leftarrow S_i \cup \{m\}$
14: $L_i \leftarrow \mathsf{sort}(S_i)$
15: $e_i \leftarrow$ index of m_i in L_i (in binary with $\lceil \log(|L_i|) \rceil$ bits)
16: encode ct_{n_s} with k bits
17: $\boxed{\textbf{return } (1,\mathsf{ct}_{n_s},R)}$
18: **return** $(\mathsf{ct}_{n_s},R,e_0\|\dots\|e_{n_s-1})$

Decode(ct_{n_s},R,E)

1: parse $(R_{-1},R_0,\dots,R_{n_s}) \leftarrow R$
2: $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda;R_{-1})$
3: $(\mathsf{st}_\mathsf{A}^0,\mathsf{st}_\mathsf{B}^0,z) \leftarrow \mathsf{Init}(\mathsf{pp};R_0)$
4: $(\mathsf{acc},\mathsf{st}_\mathsf{B}^1,\mathsf{num},m_{n_s}) \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{B}^0,\mathsf{ct}_{n_s})$
5: // Collecting false positives:
6: **for** $i \in \{1,\dots,n_s-1\}$ **do**
7: $S_i \leftarrow \emptyset$
8: **for** $m \in \{0,1\}^n$ **do**
9: $(\_,\_,\mathsf{ct}') \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{A}^{i-1},m;R_i)$
10: $(\mathsf{acc},\_,\_,m') \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{B}^1,\mathsf{ct}')$
11: **if** acc: $S_i \leftarrow S_i \cup \{m\}$
12: $L_i \leftarrow \mathsf{sort}(S_i)$
13: $e_i \leftarrow$ read next $\lceil \log(|L_i|) \rceil$ bits of E
14: $m_i \leftarrow L_i[e_i]$
15: $(\mathsf{st}_\mathsf{A}^i,\_,\_) \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{A}^{i-1},m_i;R_i)$
16: **return** $(m_1,\dots,m_{n_s},R)$

$\boxed{\textbf{Decode}(b,\mathsf{data},R)}$

1: **if** $b = 0$:
2: $(m_1,\dots,m_{n_s}) \leftarrow \mathsf{data}$
3: **return** $(m_1,\dots,m_{n_s},R)$
4: **else** $\mathsf{ct}_{n_s} \leftarrow \mathsf{data}$
5: parse $(R_{-1},R_0,\dots,R_{n_s}) \leftarrow R$
6: $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda;R_{-1})$; $(\mathsf{st}_\mathsf{A}^0,\mathsf{st}_\mathsf{B}^0,z) \leftarrow \mathsf{Init}(\mathsf{pp};R_0)$
7: $(\mathsf{acc},\mathsf{st}_B^1,\mathsf{num},m_{n_s}) \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{B}^0,\mathsf{ct}_{n_s})$
8: // Collecting false positives:
9: **for** $i \in \{1,\dots,n_s-1\}$ **do**
10: $S_i \leftarrow \emptyset$
11: **for** $m \in \{0,1\}^n$ **do**
12: $(\_,\_,\mathsf{ct}') \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{A}^{i-1},m;R_i)$
13: $(\mathsf{acc},\_,\_,m') \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{B}^1,\mathsf{ct}')$
14: **if** acc: $m_i \leftarrow m$
15: $(\mathsf{st}_\mathsf{A}^i,\_,\_) \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{A}^{i-1},m_i;R_i)$
16: **return** $(m_1,\dots,m_{n_s},R)$

Figure 4.11: Encoder without (resp. with) boxed instructions and decoder on the left (resp. right) for proving the first 2 (resp. third) lower bound(s) in Theorem 11.

Finally, we show the following key lemma.

Lemma 14. Let F_i, $i \in [n_s - 1]$ be defined as above and n, γ as in the statement of the Theorem (Theorem 11). Then, $\mathbb{E}[|F_i|] \leq 2^{n-\gamma}$.

Proof. We proceed by contradiction. That is, we show that if $\mathbb{E}[|F_i|] > 2^{n-\gamma}$, then there exists an adversary $\mathscr{A}_i$ s.t. $\Pr[\mathsf{r\text{-}RID}_{\Pi}(\mathscr{A}_i) \Rightarrow 1] > \frac{1}{2^{\gamma}}$.

$\mathscr{A}_i$	$\mathscr{A}_i$
1 : $m_1, \ldots, m_{n_s} \xleftarrow{\$} \{0,1\}^{n \times n_s}$	1 : $m_1, \ldots, m_{n_s} \xleftarrow{\$} \{0,1\}^{n \times n_s}$
2 : $R_{-1}, R_0, \ldots, R_{n_s} \xleftarrow{\$} \{0,1\}^r$	2 : $R_{-1}, R_0, \ldots, R_{n_s} \xleftarrow{\$} \{0,1\}^r$
3 : **for** $j \in \{1, \ldots, n_s\}$ **do**	3 : **for** $j \in \{1, \ldots, n_s\}$ **do**
4 : **if** $j = i$:	4 : **if** $j = i$:
5 : $\mathsf{st}_{\mathsf{A}}^{i-1} \leftarrow \mathsf{EXP}_{\mathsf{st}}(\mathsf{A})$	5 : $\mathsf{st}_{\mathsf{A}}^{i-1} \leftarrow \mathsf{EXP}_{\mathsf{st}}(\mathsf{A})$
6 : $(\mathsf{ct}_j, \mathsf{num}_j) \leftarrow \mathsf{SEND}(\mathsf{A}, \emptyset, m_j, R_j)$	6 : $(\mathsf{ct}_j, \mathsf{num}_j) \leftarrow \mathsf{SEND}(\mathsf{A}, \emptyset, m_j, R_j)$
7 : $\mathsf{RECEIVE}(\mathsf{B}, \emptyset, \mathsf{ct}_{n_s})$	7 : $(\mathsf{num}_{\mathsf{at}}, \mathsf{at}) \leftarrow \mathsf{AUTHSEND}(\mathsf{A})$
8 : $m \xleftarrow{\$} \{0,1\}^n$	8 : $i_{\mathsf{at}} \leftarrow$ index of at
9 : $\_, \_, \mathsf{ct} \leftarrow \mathsf{Send}(\mathsf{st}_{\mathsf{A}}^{i-1}, m; R_i)$	9 : $\mathsf{AUTHRECEIVE}(\mathsf{B}, i_{\mathsf{at}})$
10 : $\mathsf{RECEIVE}(\mathsf{B}, \emptyset, \mathsf{ct})$	10 : $m \xleftarrow{\$} \{0,1\}^n$
11 : **return** B	11 : $\_, \_, \mathsf{ct} \leftarrow \mathsf{Send}(\mathsf{st}_{\mathsf{A}}^{i-1}, m; R_i)$
	12 : $\mathsf{RECEIVE}(\mathsf{B}, \emptyset, \mathsf{ct})$
	13 : **return** B

Figure 4.12: r-RID adversary for the proof of Theorem 11 (resp. Theorem 12) on the left (resp. on the right).

We present such an adversary $\mathscr{A}_i$ on the left of Figure 4.12. The adversary samples n_s messages $m_1, \ldots, m_{n_s}$ at random and, letting A and B be the two parties, makes A send these with randomness $R_1, \ldots, R_{n_s}$, respectively. Then, $\mathscr{A}_i$ makes B receive the last ciphertext ct_{n_s}. Next, $\mathscr{A}_i$ samples a random message m, sends it using state $\mathsf{st}_{\mathsf{A}}^{i-1}$ and randomness R_i to get a ciphertext ct and makes B receive it. Now, as ct_{n_s} is sent after ct (ct_{n_s} is sent with $\mathsf{st}_{\mathsf{A}}^{n_s-1}$ and ct with $\mathsf{st}_{\mathsf{A}}^{i-1}$), ct_{n_s} and ct will decrypt respectively to num_{n_s} and num_i s.t. $\mathsf{num}_{n_s} > \mathsf{num}_i$ by correctness. Then, if $m \neq m_i$, then ct is different from the i-th ciphertext ct_i (as we assume $\mathsf{Send}(\mathsf{st}_{\mathsf{A}}^{i-1}, \cdot; R_i)$ is injective). Therefore, if ct is accepted and $m \neq m_i$, then ct and num_i satisfy the forgery predicate of the r-RID game in Figure 4.4. In addition, as ct_{n_s} is sent and delivered honestly, forgery and bad-$\overline{\mathsf{P}}$ in the r-RID game hold with respect to $\mathsf{B}, \mathsf{num}_i, \mathsf{num}_{n_s}, \perp, \mathsf{ct}, \perp, \mathsf{ct}_{n_s}$, relevant x and any y (as in line 6 of Figure 4.4), so the adversary wins. We call this event win.

We now compute the probability that win happens, which is the probability that $\mathsf{ct}_i \neq \mathsf{ct}$ and B accepts ct. Let $m_1, \ldots, m_{n_s}$ and the whole randomness R ($R = R_{-1}, R_0, \ldots, R_{n_s}$) be fixed. As before, let S_i be the set of messages m s.t. $\mathsf{Receive}(\mathsf{st}_B^1, \mathsf{ct})$ accepts, for $\mathsf{ct} = \mathsf{Send}(\mathsf{st}_A^{i-1}, m; R_i)$. Note that since S_i depends only $(m_1, \ldots, m_{n_s}, R)$ (which are now fixed), it is deterministic.

Therefore, conditioned on $m_1, \ldots, m_{n_s}, R$, we have

$$\Pr_m[\mathsf{win}] = \Pr_m[m \in S_i \wedge m \neq m_i] = \Pr_m[m \in F_i] = \frac{|F_i|}{2^n}$$

as m is sampled uniformly at random. Hence, overall

$$\Pr_{m,m_1,\ldots,m_{n_s},R}[\mathsf{win}] = \mathbb{E}_{m_1,\ldots,m_{n_s},R}[\Pr_m[m \in F_i]] = \frac{\mathbb{E}[|F_i|]}{2^n} .$$

Note that both the source and the adversary sample $m_1, \ldots, m_{n_s}, R$ uniformly at random. Finally, if $\mathbb{E}[|F_i|] > 2^{n-\gamma}$, then $\Pr[\mathsf{win}] > \frac{2^{n-\gamma}}{2^n} = 2^{-\gamma}$, which leads to the contradiction. □

By the previous lemma, we have $\log(\mathbb{E}[|F_i|]+1) \leq \log(2^{n-\gamma}+1) \leq \max(0, n-\gamma)+1$. Plugging this result into Lemma 13, we get $\mathbb{E}[C] \leq k + r + (n_s - 1)(\max(0, n-\gamma)+2)$. In addition, as our encoder outputs a uniquely decodable code, we know that $n_s n + r \leq \mathbb{E}[C]$ by Shannon's source coding theorem. Hence, we get

$$k + r + (n_s - 1)(n - \gamma + 2) \geq n_s n + r \iff k \geq n + (n_s - 1)(\gamma - 2)$$

if $\gamma \leq n$ and otherwise

$$k + r + (n_s - 1)2 \geq n_s n + r \iff k \geq 2 + n_s(n-2).$$

Now that the first two lower bounds have been shown, we prove the final bound in the following lemma.

Lemma 15. Let k, n, n_s, γ as in the statement of the theorem. Then,

$$k \geq n n_s - \frac{1}{1 - \frac{2^n n_s}{2^\gamma}} .$$

Proof. In order to prove this lemma, we build another encoder/decoder pair very similar to the previous one. They are shown in Figure 4.11 (*with* the boxed instructions for the encoder and the boxed decoder on the right). The only difference in the encoder is that if *one* false positive is found, the encoder outputs a bit set to zero and the trivial encoding of the input. Let's call this event fail. If fail does not occur, a bit set to 1, the last ciphertext, and the randomness are output.

In the decoder, either the first bit of the input is set to 0 and the input is returned straightaway, or $b = 1$ and the decoder proceeds as before. However, as there are no false positives, the m_i can be recovered without using the indices e_i (i.e., the correct message would be the only element of S_i). Overall, the expected codeword length is $\mathbb{E}[C] = 1 + \alpha n n_s + (1-\alpha)k + r$, where $\alpha := \Pr[\mathsf{fail}]$, as if fail occurs a trivial encoding (on $1 + n n_s + r$ bits) is used, and otherwise the

encoder outputs $1+k+r$ bits. By Shannon source coding theorem, we obtain

$$1+\alpha n n_s+(1-\alpha)k+r \geq n n_s+r \iff k \geq n n_s-\frac{1}{1-\alpha}$$

In addition, we have $\alpha := \Pr[\mathsf{fail}] = \Pr[\cup_{i=1}^{n_s-1}\{|F_i| \geq 1\}] \leq \sum_{i=1}^{n_s-1}\Pr[|F_i| \geq 1]$ as fail occurs if at least one of the sets of false positives F_i contains an element. Then, we have $\Pr[|F_i| \geq 1] \leq \mathbb{E}[|F_i|] \leq 2^{n-\gamma}$, where the first inequality follows from Markov's inequality and the second from Lemma 14. Overall, we get $\alpha \leq \frac{n_s 2^n}{2^\gamma}$. Hence,

$$k \geq n n_s-\frac{1}{1-\alpha} \geq n n_s-\frac{1}{1-\frac{2^n n_s}{2^\gamma}}\,.$$

Finally, some algebra shows that this bound is tighter than the second one when

$$2^\gamma \geq 2^n n_s \frac{2-2n_s}{3-2n_s}\,,$$

that is, when γ is larger than $\approx n+\log(n_s)$. □

This completes the proof. □

On Imperfect Correctness. For simplicity in the proof, we only considered RC schemes which are perfectly correct. Note, however, that it should be possible to obtain a slightly worse bound for RC schemes that are *not* perfectly (computationally or statistically) correct. In more detail, perfect correctness is used twice in the proof of Theorem 11: (1) in the encoder to argue that the encoded messages will decrypt properly and (2) in the reduction (Figure 4.12) to argue that $m \neq m_i \Rightarrow \mathsf{ct} \neq \mathsf{ct}_i$. Then, if the probability that a correctness error arises is at most δ, we can argue as follows. In case (1), we can simply use the trick used to prove the 3rd bound (i.e., output the trivial encoding if the encoded message does not decrypt properly) to get the same bounds $-1/(1-\delta \cdot n_s)$. Then, in case (2), we will have at the end $\Pr[\mathsf{win}] > 2^{-\gamma}-\delta$ (the bound calculated at the end of our proof of Lemma 14) as we need to take into account the probability that m triggers a correctness error. This should incur an additional $\approx -\log(\delta)$ loss in the bound. Overall, the proof still holds with $\delta > 0$, and if it is small then the bounds remain nearly identical.

Beyond Uniformly Sampled Messages. As stated, our lower bound assumes that the n_s messages encoded are sampled uniformly at random from the message space. We note that a comparable bound holds if the distribution is different from uniform given the bound is stated in terms of the entropy of the messages.

4.5.2 Communication Cost for r-UNF Security

We consider a perfectly correct ARC (i.e., the function $\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \cdot; \mathsf{r})$ is injective for all randomness r, valid states st_P and associative data ad). The following theorem states that the tag size of a secure ARC grows linearly in the number of messages (times either the security parameter or message size).

Theorem 12. Let Π be a perfectly correct ARC, n_s and λ be fixed, and T_{λ,n_s} be the time complexity of the (efficient) adversary given on the right of Figure 4.12. In addition, let $\gamma \in \mathbb{Z}$ be such that for all adversaries $\mathscr{A}$ running in at most time T_{λ,n_s} which query oracle SEND at most n_s times, we have: $\Pr[\mathsf{r\text{-}UNF}_\Pi(\mathscr{A}) \Rightarrow 1] \leq \frac{1}{2^\gamma}$. Let $\mathscr{M} = \{0,1\}^n$ and $\mathscr{T} = \{0,1\}^k$ be the plaintext and tag space associated to Π, respectively. Then, $k \geq n_s(\gamma - 2)$, if $\gamma \leq n$, and $k \geq n_s(n-2)$, if $\gamma > n$. A third lower bounds gives

$$k \geq n n_s - \frac{1}{1 - \frac{2^n(n_s+1)}{2^\gamma}},$$

which is tighter for low values of n (e.g. $n = 1, 2$) and when $\gamma > n + \log(n_s)$.

Proof. We only provide the idea of the proof, as it is nearly identical to the one of Theorem 11. The intuition is that if a party $\overline{\mathsf{P}}$ sends n_s messages and then an authentication tag at to P, then at must contain information about all the ciphertexts previously sent. This is because $\overline{\mathsf{P}}$ cannot know in the worst case which messages were ever received by P.

More precisely, the only difference with the proof of Theorem 11 is that we use at instead of ct_{n_s}; the rest follows similarly. In particular, the encoder outputs the randomness, the tag and the indices of the encoded messages in the sets of correctly received messages. Then, the decoder receives the authentication tag and tries to receive all possible ciphertexts ct_1. Among the ones that are successfully received, it extracts the correct message using the index provided by the encoder. Next it moves to receiving all possible ciphertexts ct_2 and so on, until the n_s messages are received. In addition, we give on the right of Figure 4.12 the adversary that can be used to prove an upper bound on the number of messages that are correctly received (i.e., the number of messages in the sets S_i used in the proof of Theorem 11). □

4.6 Optimisations and Performance/Security Trade-Offs

In Section 4.5, we showed that r-RID and r-UNF impose a linear communication complexity on RC and ARC schemes respectively. In this section we explore ways to bypass these lower bounds and propose practical approaches for active attack detection. We first argue that s-RID/s-UNF security can be achieved at a much lower cost than r-RID/r-UNF security. Noting that ciphertexts grow without bound in the schemes presented up to this point, we propose two methods of pruning, or garbage collection, to reduce communication overhead. The first involves pruning the set R that is sent for s-RID security every full round trip in

Figure 4.6. The second involves pruning unnecessary values included in authentication tags in our construction in Figure 4.9 for UNF security. We then conclude the chapter by proposing a lightweight three-move protocol that authenticates communication in both directions over the out-of-band channel.

4.6.1 On the Practicality of s-RID and s-UNF Security

Security Notion:	r-RID	s-RID
Overhead	$\mathcal{O}(n\lambda + c_n)$	$\mathcal{O}(\lambda + c_n)$
Optimised overhead (Section 4.6.2)	N/A	$\mathcal{O}\left(\lambda + c_{n_{\mathsf{fresh}}}\right)$

Table 4.1: Overhead induced by the two RID security notions. We assume that n messages are received and c_i is the space needed to encode i ordinals. The variable n_{fresh} refers to the number of messages received in the last two epochs.

We focus here on s-RID security, but similar arguments hold for s-UNF security. The scheme $\mathsf{RC}_{\mathsf{RID}}$ (Figure 4.6) achieves s-RID security by sending to the counterpart the list of received ordinals and an *hash* of the set R. Recall informally that this suffices for security because a party can immediately detect when their counterpart has received a forgery (by keeping in state their sent messages and recomputing the hash). Table 4.1 summarises the overhead incurred by the two notions together with the optimisation presented in the next subsection (Section 4.6.2), where we show that it is enough to only send information about messages received during the two last epochs. This significantly reduces the overhead for the scenarios in which the communication is "balanced".

Ordinals. The RID-secure RC of Figure 4.6 sends the set of *received* ordinals for authentication (line 2 of the $\mathsf{RC}_{\mathsf{RID}}$.Send algorithm). Since ordinals are elements of a set on which a total order is defined, a simple optimisation—that could reduce the overhead by up to 50%—consists in sending the smallest set among the set of received ordinals *or* the set of not received ordinals alongside a bit indicating which type of set has been sent. This optimisation applies to all schemes that send sets of ordinals.

One can reduce ciphertext size further by optimising for the "good case" scenario where messages are delivered in-order; in this case, ordinals can be encoded in ranges. For epochs with no lost messages, it suffices to encode only the last index. In any case, as the size of a single message in today's secure messaging applications can be several kilobytes or more, especially when audio and video is used, the overhead that s-RID imposes seems reasonable. We leave a deeper and more concrete analysis to determine the impact of RID/UNF security in practice to future work.

Incremental Hashing. In Figure 4.6, the entire set of received messages is hashed (using a regular hash function) every time a message is sent by P. Consequently, when P receives a new message, the entire hash must be re-computed when P sends their next message. To

avoid this, the scheme can instead use an *incremental hash function* (Definition 10) such that, when a message is received, an efficient operation only depending on the new message and the previous digest can be executed to derive the new digest. Hash digests can be as small as a group element [CDv+03]. This enables parties to prune their set of sent/received messages in state. For example, if P receives a message m claiming that $\overline{\mathsf{P}}$ has received the first k messages from P, and P has received messages for all possible ordinals that precede the ordinal of m, then P can safely store just the incrementally-hashed value corresponding to the first k messages, since $\overline{\mathsf{P}}$ can no longer claim to have only received a strict subset of the k messages.

The security proof then follows almost exactly as in the proof that use the collision resistance of the hash function by instead assuming *set collision resistance* (Definition 11) and arguing the that a set collision can be constructed in the exact same situations as a collision when using a (regular) hash function.

4.6.2 Epoch-Based Optimisation for s-RID Security

In this subsection we show how to design an optimised s-RID-secure RC scheme given a correct and ORDINALS-secure RC scheme. A formal description is given in Figure 4.13.

At initialisation, one party is associated with $\mathsf{ep} = 0$, say P, and the other, say $\overline{\mathsf{P}}$, with $\mathsf{ep} = 1$. Each time a party sends a message they also attach their current ep to the message. Upon receiving a message, a party P with $\mathsf{ep} = t$ checks whether the ep attached to the message received from $\overline{\mathsf{P}}$ is at most $t+1$. If the $\overline{\mathsf{P}}$'s ep is exactly $t+1$ the party P updates their ep to $t+2$. The ep value of the parties are always one apart at each point in time. Note this is exactly how epochs were defined by Alwen et al. [ACD19] except each epoch was associated with an iteration of the Diffie-Hellman-based asymmetric ratchet.

To achieve s-RID security the sender does the following. Whenever sending a message with $\mathsf{ep} = t$, the sender attaches the num and the accumulated hash of all messages they received during the time their ep was t and $t-2$. That is, the parties do the same as the original s-RID construction, but only for the messages they have received in the last 2 epochs.

Although this optimisation does not change the worst-case complexity of Figure 4.6, if the direction of the conversation changes frequently enough, the overhead significantly decreases.

We continue by stating the main theorem of this subsection and providing a proof sketch.

Theorem 13. Consider collision resistant hash function H used to build $\mathsf{RC}_{\mathsf{s\text{-}RID}}$ (defined in Figure 4.13). Then, we have that for every efficient adversary $\mathscr{A}$, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathsf{s-rid}}_{\mathsf{RC}_{\mathsf{s\text{-}RID}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathscr{B}) .$$

Proof sketch. Let $[(\mathsf{ct}_f, \cdot, t_f), (\mathsf{ct}_h, \cdot, t_h)]$, be the closest pair of sent-received messages contra-

$\mathsf{RC}_{\text{s-RID}}.\mathsf{Setup}(1^\lambda)$

1 : $\mathsf{pp}_0 \xleftarrow{\$} \mathsf{RC}.\mathsf{Setup}(1^\lambda)$
2 : $\mathsf{hk} \xleftarrow{\$} \mathsf{H}.\mathsf{KGen}(1^\lambda)$
3 : $\mathsf{hk}' \xleftarrow{\$} \mathsf{H}.\mathsf{KGen}(1^\lambda)$
4 : $\mathsf{pp} \leftarrow (\mathsf{pp}_0, \mathsf{hk}, \mathsf{hk}')$
5 : **return** pp

$\mathsf{RC}_{\text{s-RID}}.\mathsf{Init}(\mathsf{pp})$

1 : $(\mathsf{pp}_0, \mathsf{hk}, \mathsf{hk}') \leftarrow \mathsf{pp}$
2 : $(\mathsf{st}'_\mathsf{A}, \mathsf{st}'_\mathsf{B}, z') \xleftarrow{\$} \mathsf{RC}.\mathsf{Init}(\mathsf{pp}_0)$
3 : $\mathsf{ep}_{\mathsf{ack}} \leftarrow 0$
4 : $\mathsf{ep} \leftarrow 0$
5 : $\mathsf{S}, \mathsf{R}_{\mathsf{curr}}, \mathsf{R}_{\mathsf{prev}} \leftarrow \emptyset$
6 : $\mathsf{st}_\mathsf{A} \leftarrow (\mathsf{st}'_\mathsf{A}, \mathsf{hk}, \mathsf{hk}', \mathsf{S}, \mathsf{R}_{\mathsf{curr}}, \mathsf{R}_{\mathsf{prev}}, \mathsf{ep})$
7 : $\mathsf{st}_\mathsf{B} \leftarrow (\mathsf{st}'_\mathsf{B}, \mathsf{hk}, \mathsf{hk}', \mathsf{S}, \mathsf{R}_{\mathsf{curr}}, \mathsf{R}_{\mathsf{prev}}, \mathsf{ep})$
8 : $z \leftarrow (z', \mathsf{pp})$
9 : **return** $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z)$

$\mathsf{RC}_{\text{s-RID}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$

1 : $(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{hk}', \mathsf{S}, \mathsf{R}_{\mathsf{curr}}, \mathsf{R}_{\mathsf{prev}}, \mathsf{ep}) \leftarrow \mathsf{st}_\mathsf{P}$
2 : $\mathsf{nums}' \leftarrow \{\mathsf{num}' : (\mathsf{num}', \_) \in \mathsf{R}_{\mathsf{curr}} \cup \mathsf{R}_{\mathsf{prev}}\}$
3 : $\mathsf{R}' \leftarrow (\mathsf{nums}', \mathsf{H}.\mathsf{Eval}(\mathsf{hk}', \mathsf{R}_{\mathsf{curr}} \cup \mathsf{R}_{\mathsf{prev}}))$
4 : $\mathsf{ad}' \leftarrow (\mathsf{ad}, \mathsf{ep}, \mathsf{R}')$
5 : $(\mathsf{st}_\mathsf{P}.\mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{ct}') \xleftarrow{\$} \mathsf{RC}.\mathsf{Send}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}', \mathsf{pt})$
6 : $\mathsf{ct} \leftarrow (\mathsf{ct}', \mathsf{ep}, \mathsf{R}')$
7 : $h \leftarrow \mathsf{H}.\mathsf{Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))$
8 : $\mathsf{st}_\mathsf{P}.\mathsf{S} \leftarrow \mathsf{S} \cup \{(\mathsf{num}, h)\}$
9 : **return** $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{ct})$

$\mathsf{RC}_{\text{s-RID}}.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$

1 : $(\mathsf{ct}', \mathsf{ep}^{\overline{\mathsf{P}}}, \mathsf{R}^{\overline{\mathsf{P}}}) \leftarrow \mathsf{ct}$
2 : $(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{hk}', \mathsf{S}, \mathsf{R}_{\mathsf{curr}}, \mathsf{R}_{\mathsf{prev}}, \mathsf{ep}) \leftarrow \mathsf{st}_\mathsf{P}$
3 : $\mathsf{ad}' \leftarrow (\mathsf{ad}, \mathsf{ep}^{\overline{\mathsf{P}}}, \mathsf{R}^{\overline{\mathsf{P}}})$
4 : $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{pt}) \leftarrow \mathsf{RC}.\mathsf{Receive}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}', \mathsf{ct}')$
5 : **if** $\neg\mathsf{acc}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
6 : $h \leftarrow \mathsf{H}.\mathsf{Eval}(\mathsf{hk}, (\mathsf{num}, \mathsf{ad}, \mathsf{ct}))$
7 : **if** $\mathsf{checks}(\mathsf{st}_\mathsf{P}, \mathsf{ct}, h, \mathsf{num})$:
8 : **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
9 : $\mathsf{st}_\mathsf{P}.\mathsf{R}_{\mathsf{curr}} \leftarrow \mathsf{R}_{\mathsf{curr}} \cup \{(\mathsf{num}, h)\}$
10 : $\mathsf{st}_\mathsf{P}.\mathsf{st}'_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
11 : // Advance epochs accordingly
12 : **if** $\mathsf{ep}^{\overline{\mathsf{P}}} = \mathsf{st}_\mathsf{P}.\mathsf{ep} + 1$:
13 : $\mathsf{st}_\mathsf{P}.\mathsf{ep} \leftarrow \mathsf{st}_\mathsf{P}.\mathsf{ep} + 2$
14 : $\mathsf{st}_\mathsf{P}.\mathsf{R}_{\mathsf{prev}} \leftarrow \mathsf{R}_{\mathsf{curr}}$
15 : $\mathsf{st}_\mathsf{P}.\mathsf{R}_{\mathsf{curr}} \leftarrow \emptyset$
16 : **return** $(\mathsf{acc}, \mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{pt})$

$\mathsf{checks}(\mathsf{st}_\mathsf{P}, \mathsf{ct}, h, \mathsf{num})$

1 : $(\mathsf{nums}', h') \leftarrow \mathsf{ct}.\mathsf{R}$
2 : $\mathsf{ep}' \leftarrow \mathsf{ct}.\mathsf{ep}$
3 : **if** $\mathsf{ep}' > \mathsf{st}_\mathsf{P}.\mathsf{ep} + 1$:
4 : $\mathsf{s\text{-}bool} \leftarrow 1$
5 : $\mathsf{R}^* \leftarrow \{(\mathsf{num}', \_) \in \mathsf{st}_\mathsf{P}.\mathsf{S} : \mathsf{num}' \in \mathsf{nums}'\}$
6 : $\mathsf{s\text{-}bool} \leftarrow \mathsf{s\text{-}bool} \vee (\mathsf{H}.\mathsf{Eval}(\mathsf{st}_\mathsf{P}.\mathsf{hk}', \mathsf{R}^*) \neq h')$
7 : **return** $\mathsf{s\text{-}bool}$

Figure 4.13: Optimised s-RID-secure RC scheme $\mathsf{RC}_{\text{s-RID}}$ given a correct and ORDINALS-secure RC scheme RC.

dicting the s-RID condition. That is, $(\mathsf{ct}_f, \cdot, t_f)$ is a forgery received by $\overline{\mathsf{P}}$ before they sent the honest message $(\mathsf{ct}_h, \cdot, t_h)$ which was received by P. We consider the time when $(\mathsf{ct}_h, \cdot, t_h)$ was sent by $\overline{\mathsf{P}}$. As mandated by the construction $(\mathsf{ct}_h, \cdot, t_h)$ contained num values and accumulated hash of all messages $\overline{\mathsf{P}}$ has received at epochs t_h and $t_h - 2$. Following the same argument as the proof of Theorem 8 one can show that no forgeries, including $(\mathsf{ct}_f, \cdot, t_f)$, were received by $\overline{\mathsf{P}}$ while $\mathsf{ep}_{\overline{\mathsf{P}}} \in \{t_h, t_h - 2\}$.

The two messages that changes $\mathsf{ep}_{\overline{\mathsf{P}}}$ from $t_h - 4$ to $t_h - 2$ and from $t_h - 2$ to t_h, were honest messages by P. Let us call them $(\mathsf{ct}_{t_h-3}, \cdot, t_h - 3)$ and $(\mathsf{ct}_{t_h-1}, \cdot, t_h - 1)$ respectively. Note that,

both these messages were received after $(\mathsf{ct}_f, \cdot, t_f)$ was received and before $(\mathsf{ct}_f, \cdot, t_f)$ was sent, as otherwise $(\mathsf{ct}_h, \cdot, t_h)$ would contradict $(\mathsf{ct}_f, \cdot, t_f)$. Note that between sending the messages $(\mathsf{ct}_{t_h-3}, \cdot, t_h - 3)$ and $(\mathsf{ct}_{t_h-1}, \cdot, t_h - 1)$, ep_P was changed meaning P received a message with $\mathsf{ep} = t_h - 2$ that caused the change of ep_P. Let us call this message $(\mathsf{ct}_{t_h-2}, \cdot, t_h - 2)$. We argue this message should have been forged.

Let us assume by contradiction that this message was honest. Note that $(\mathsf{ct}_{t_h-2}, \cdot, t_h - 2)$ was sent after $(\mathsf{ct}_{t_h-3}, \cdot, t_h - 3)$ was received, hence after $(\mathsf{ct}_f, \cdot, t_f)$ was received. Now if $(\mathsf{ct}_{t_h-2}, \cdot, t_h - 2)$ is honest it forms a pair with $(\mathsf{ct}_f, \cdot, t_f)$ which violates the s-RID condition and has less distance from the original pair which is a contradiction. So $(\mathsf{ct}_{t_h-2}, \cdot, t_h - 2)$ must have been forged. A visualisation of the scenario can be found in Figure 4.14.

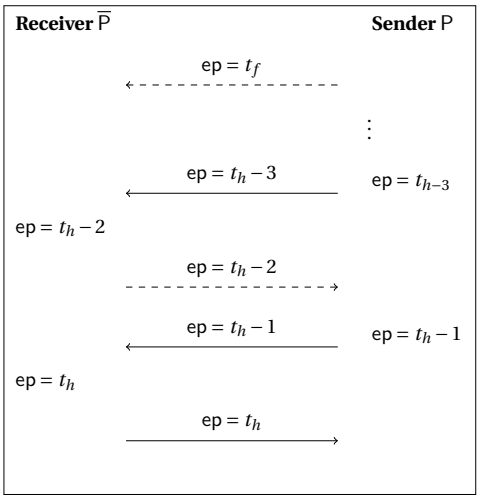

Figure 4.14: Visualising the proof sketch of Theorem 13. The dotted messages are forged and the others are honest messages. Intuitively we argue if the last message did not contradict the first message, the fourth message would have contradicted the third, therefore violating s-RID in the other direction.

One other observation is that, $(\mathsf{ct}_{t_h-2}, \cdot, t_h - 2)$ was received before $(\mathsf{ct}_{t_h-1}, \cdot, t_h - 1)$ was sent, hence, before $(\mathsf{ct}_h, \cdot, t_h)$ was received. This shows that the pair $[(\mathsf{ct}_{t_h-2}, \cdot, t_h - 2), (\mathsf{ct}_{t_h-1}, \cdot, t_h - 1)]$ also violates the s-RID (for $\overline{\mathsf{P}}$ and not P) and has less distance than the original pair. □

4.6.3 Pruning for UNF Security

In this subsection, we present a scheme that optimises bandwidth consumption for UNF security (Figure 4.15). A complete description and security proof are given below. Our scheme takes advantage of the fact that messages sent out-of-band cannot be forged. Suppose that P sends an authentication tag to $\overline{\mathsf{P}}$, then $\overline{\mathsf{P}}$ acknowledge the reception of the tag to P. At this point, P no longer needs to send the information that $\overline{\mathsf{P}}$ has already obtained. Our scheme supports out-of-order communication even on the authenticated channels. Our approach is complicated by this and the fact that parties have to keep track of, e.g., which tags their

partner has received to determine what is safe to prune from state (in $\mathsf{S_{at}Seen}$), which incurs relatively small overhead in typical executions.

Scheme Description. Send stores the hash of $(\mathsf{ad}, \mathsf{ct})$ for the message being sent, together with the corresponding num that the underlying RC.Send call returns. It stores (num, h) in a set S, which is in turn stored in the party's internal state. The Send algorithm also updates the ordinal num in the state.

The Receive procedure verifies whether RC.Receive accepts the inputs and verifies that the received message is not a forgery based on any information stored from previous AuthReceive calls. Given this passes, Receive stores the hash of $(\mathsf{ad}, \mathsf{ct})$ together with the ordinal num returned by RC.Receive in a set R.

The AuthSend procedure is similar to the unoptimised one, except that (1) it stores the set of sent messages S authenticated *within the current* at into an array $\mathsf{S_{at}}$, indexed by counter $\mathsf{cnt_{at}}$, and (2) it empties the set $\mathsf{S_{at}Seen}$ locally; the goal of this set is to communicate which tags that the caller has received from their counterpart.

Before processing the message, AuthReceive first verifies whether $\mathsf{cnt_{at}}^{\overline{\mathsf{P}}} \leq \mathsf{max\text{-}cnt_{at}}$. The goal of this check is to avoid processing old authentication tags, since if the check is true, AuthReceive must have previously authenticated the tag's content in a newer (in terms of $\mathsf{cnt_{at}}^{\overline{\mathsf{P}}}$) tag. However, pruning still needs to be performed before returning in this case since the contents of set $\mathsf{S_{at}Seen}$ is only sent once before being flushed in a given AuthSend call.

Otherwise, given the sent and received messages of the two parties are consistent, AuthReceive first stores the counter of the input tag in $\mathsf{S_{at}Seen}$, which will be sent to the counterpart in the next call to AuthSend. Then it removes the already authenticated messages from memory. The caller already authenticated the subset $\mathsf{R}^{\mathsf{P}}_{\subseteq}$ and it can thus remove the corresponding messages from its set R, which represents now the set of currently unauthenticated received messages. Similarly, AuthReceive uses the set of authentication tags that the counterpart already processed to prune the set of sent messages.

In detail, the pruning of sent messages works as follows. When a party P receives a set of messages $\mathsf{S_{at}}^{\overline{\mathsf{P}}}[\mathsf{cnt_{at}}] \leftarrow \mathsf{S}$ sent by the counterpart with the authentication tag number $\mathsf{cnt_{at}}$, it stores them in a set $\mathsf{S_{ack}}$. Then, when P sends a subsequent authentication tag back to $\overline{\mathsf{P}}$, it informs $\overline{\mathsf{P}}$ that the authentication tag $\mathsf{cnt_{at}}$ was received using the $\mathsf{S_{at}Seen}^{\mathsf{P}}$ set. When this tag is delivered, $\overline{\mathsf{P}}$ can remove the acknowledged messages $\mathsf{S_{at}}^{\overline{\mathsf{P}}}[\mathsf{cnt_{at}}]$ for all counters in $\mathsf{S_{at}Seen}^{\mathsf{P}}$ from its set $\mathsf{S}^{\overline{\mathsf{P}}}$. This reduces the size of the authentication tag as, on every round-trip on the out-of-band channel, all authenticated messages can be removed from the sets S and R. To reduce the size even further, we can use the same hashing optimisation for the received set that we used in Figure 4.6.

Security Analysis. We informally argue that the scheme prunes only messages that have already been authenticated. The procedures use sets $\mathsf{st_P}.\mathsf{S}$ and $\mathsf{st_P}.\mathsf{R}$ to detect active attacks.

$\mathsf{ARC}_{\mathsf{OP}}.\mathsf{Setup}(1^\lambda)$

$\mathsf{pp}_0 \leftarrow \mathsf{RC.Setup}(1^\lambda)$; $\mathsf{hk} \leftarrow \mathsf{H.KGen}(1^\lambda)$
return $(\mathsf{pp}_0, \mathsf{hk})$

$\mathsf{ARC}_{\mathsf{OP}}.\mathsf{Init}(\mathsf{pp})$

$(\mathsf{pp}_0, \mathsf{hk}) \leftarrow \mathsf{pp}$
$(\mathsf{st}'_\mathsf{A}, \mathsf{st}'_\mathsf{B}, z') \leftarrow \mathsf{RC.Init}(\mathsf{pp})$
$\mathsf{num}, \mathsf{max\text{-}num}, \mathsf{cnt}_\mathsf{at}, \mathsf{max\text{-}cnt}_\mathsf{at} \leftarrow 0$
$\mathsf{S}, \mathsf{R}, \mathsf{S}_\mathsf{ack}, \mathsf{S}_\mathsf{at}, \mathsf{S}_\mathsf{at}\mathsf{Seen} \leftarrow \emptyset$
$\mathsf{st}_\mathsf{A} \leftarrow (\mathsf{st}'_\mathsf{A}, \mathsf{hk}, \mathsf{S}, \mathsf{R}, \mathsf{S}_\mathsf{ack}, \mathsf{num}, \mathsf{max\text{-}num},$
  $\mathsf{cnt}_\mathsf{at}, \mathsf{max\text{-}cnt}_\mathsf{at}, \mathsf{S}_\mathsf{at}, \mathsf{S}_\mathsf{at}\mathsf{Seen})$
$\mathsf{st}_\mathsf{B} \leftarrow (\mathsf{st}'_\mathsf{B}, \mathsf{hk}, \mathsf{S}, \mathsf{R}, \mathsf{S}_\mathsf{ack}, \mathsf{num}, \mathsf{max\text{-}num},$
  $\mathsf{cnt}_\mathsf{at}, \mathsf{max\text{-}cnt}_\mathsf{at}, \mathsf{S}_\mathsf{at}, \mathsf{S}_\mathsf{at}\mathsf{Seen})$
$z \leftarrow (z', \mathsf{pp})$
return $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z)$

$\mathsf{ARC}_{\mathsf{OP}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$

$(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{S}, \cdots) \leftarrow \mathsf{st}_\mathsf{P}$
$(\mathsf{st}_\mathsf{P}.\mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{ct}) \leftarrow \mathsf{RC.Send}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$
$h \leftarrow \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{ad}, \mathsf{ct}))$
$\mathsf{st}_\mathsf{P}.\mathsf{S} \leftarrow \mathsf{S} \cup \{(\mathsf{num}, h)\}$
$\mathsf{st}_\mathsf{P}.\mathsf{num} \leftarrow \mathsf{num}$
return $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{ct})$

$\mathsf{ARC}_{\mathsf{OP}}.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$

$(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \_, \mathsf{R}, \mathsf{S}_\mathsf{ack}, \_, \mathsf{max\text{-}num}, \cdots) \leftarrow \mathsf{st}_\mathsf{P}$
$(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{pt}) \leftarrow \mathsf{RC.Receive}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$
if $\neg\mathsf{acc}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
$h \leftarrow \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{ad}, \mathsf{ct}))$
if $\mathsf{num} \le \mathsf{max\text{-}num} \wedge (\mathsf{num}, h) \notin \mathsf{S}_\mathsf{ack}$:
  return $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
$\mathsf{st}_\mathsf{P}.\mathsf{R} \leftarrow \mathsf{R} \cup \{(\mathsf{num}, h)\}$
$\mathsf{st}_\mathsf{P}.\mathsf{st}'_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
return $(\mathsf{acc}, \mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{pt})$

$\mathsf{ARC}_{\mathsf{OP}}.\mathsf{AuthSend}(\mathsf{st}_\mathsf{P})$

$(\_, \_, \mathsf{S}, \mathsf{R}, \_, \mathsf{num}, \_, \mathsf{cnt}_\mathsf{at}, \_, \mathsf{S}_\mathsf{at}\mathsf{Seen}) \leftarrow \mathsf{st}_\mathsf{P}$
$\mathsf{at} \leftarrow (\mathsf{S}, \mathsf{R}, \mathsf{num}, \mathsf{cnt}_\mathsf{at}, \mathsf{S}_\mathsf{at}\mathsf{Seen})$
$\mathsf{st}_\mathsf{P}.\mathsf{cnt}_\mathsf{at} \leftarrow \mathsf{st}_\mathsf{P}.\mathsf{cnt}_\mathsf{at} + 1$
$\mathsf{st}_\mathsf{P}.\mathsf{S}_\mathsf{at}[\mathsf{st}_\mathsf{P}.\mathsf{cnt}_\mathsf{at}] \leftarrow \mathsf{S}$
$\mathsf{st}_\mathsf{P}.\mathsf{S}_\mathsf{at}\mathsf{Seen} \leftarrow \emptyset$
return $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{at})$

$\mathsf{ARC}_{\mathsf{OP}}.\mathsf{AuthReceive}(\mathsf{st}_\mathsf{P}, \mathsf{at})$

$(\_, \_, \mathsf{S}, \mathsf{R}, \mathsf{S}_\mathsf{ack}, \mathsf{num}, \mathsf{max\text{-}num},$
  $\mathsf{cnt}_\mathsf{at}, \mathsf{max\text{-}cnt}_\mathsf{at}, \mathsf{S}_\mathsf{at}, \_) \leftarrow \mathsf{st}_\mathsf{P}$
$(\mathsf{S}^{\overline{\mathsf{P}}}, \mathsf{R}^{\overline{\mathsf{P}}}, \mathsf{num}^{\overline{\mathsf{P}}}, \mathsf{cnt}_\mathsf{at}{}^{\overline{\mathsf{P}}}, \mathsf{S}_\mathsf{at}\mathsf{Seen}^{\overline{\mathsf{P}}}) \leftarrow \mathsf{at}$
$\mathsf{R}^\mathsf{P}_\subseteq \leftarrow \{(\mathsf{num}, \_) \in \mathsf{R} : \mathsf{num} \le \mathsf{num}^{\overline{\mathsf{P}}}\}$
if $\mathsf{cnt}_\mathsf{at}{}^{\overline{\mathsf{P}}} \le \mathsf{max\text{-}cnt}_\mathsf{at}$:
  $\mathsf{prune}(\mathsf{st}_\mathsf{P}, \mathsf{R}^{\overline{\mathsf{P}}}, \mathsf{cnt}_\mathsf{at}{}^{\overline{\mathsf{P}}}, \mathsf{S}_\mathsf{at}\mathsf{Seen}^{\overline{\mathsf{P}}}, \mathsf{R}^\mathsf{P}_\subseteq)$
  return $(\mathsf{true}, \mathsf{st}_\mathsf{P}, \mathsf{num}^{\overline{\mathsf{P}}})$
// $\overline{\mathsf{P}}$ received a forgery
if $\mathsf{R}^{\overline{\mathsf{P}}} \not\subseteq \mathsf{S}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \mathsf{num})$
// P received a forgery
if $\mathsf{R}^\mathsf{P}_\subseteq \not\subseteq \mathsf{S}^{\overline{\mathsf{P}}}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \mathsf{num})$
$\mathsf{st}_\mathsf{P}.\mathsf{S}_\mathsf{ack} \leftarrow \mathsf{st}_\mathsf{P}.\mathsf{S}_\mathsf{ack} \cup \mathsf{S}^{\overline{\mathsf{P}}}$
$\mathsf{st}_\mathsf{P}.\mathsf{max\text{-}num} \leftarrow \max\{\mathsf{max\text{-}num}, \mathsf{num}^{\overline{\mathsf{P}}}\}$
$\mathsf{st}_\mathsf{P}.\mathsf{max\text{-}cnt}_\mathsf{at} \leftarrow \max\{\mathsf{cnt}_\mathsf{at}{}^{\overline{\mathsf{P}}}, \mathsf{st}_\mathsf{P}.\mathsf{max\text{-}cnt}_\mathsf{at}\}$
$\mathsf{prune}(\mathsf{st}_\mathsf{P}, \mathsf{R}^{\overline{\mathsf{P}}}, \mathsf{cnt}_\mathsf{at}{}^{\overline{\mathsf{P}}}, \mathsf{S}_\mathsf{at}\mathsf{Seen}^{\overline{\mathsf{P}}}, \mathsf{R}^\mathsf{P}_\subseteq)$
return $(\mathsf{true}, \mathsf{st}_\mathsf{P}, \mathsf{num}^{\overline{\mathsf{P}}})$

$\mathsf{prune}(\mathsf{st}_\mathsf{P}, \mathsf{R}^{\overline{\mathsf{P}}}, \mathsf{cnt}_\mathsf{at}{}^{\overline{\mathsf{P}}}, \mathsf{S}_\mathsf{at}\mathsf{Seen}^{\overline{\mathsf{P}}}, \mathsf{R}^\mathsf{P}_\subseteq)$

$\mathsf{st}_\mathsf{P}.\mathsf{S}_\mathsf{at}\mathsf{Seen} \leftarrow \mathsf{st}_\mathsf{P}.\mathsf{S}_\mathsf{at}\mathsf{Seen} \cup \{\mathsf{cnt}_\mathsf{at}{}^{\overline{\mathsf{P}}}\}$
$\mathsf{st}_\mathsf{P}.\mathsf{R} \leftarrow \mathsf{st}_\mathsf{P}.\mathsf{R} \setminus \mathsf{R}^\mathsf{P}_\subseteq$
for $i \in \mathsf{S}_\mathsf{at}\mathsf{Seen}^{\overline{\mathsf{P}}}$ **do**
  $\mathsf{st}_\mathsf{P}.\mathsf{S} \leftarrow \mathsf{st}_\mathsf{P}.\mathsf{S} \setminus \mathsf{st}_\mathsf{P}.\mathsf{S}_\mathsf{at}[i]$; $\mathsf{st}_\mathsf{P}.\mathsf{S}_\mathsf{at}[i] \leftarrow \emptyset$

Figure 4.15: Optimised UNF-secure ARC scheme $\mathsf{ARC}_{\mathsf{OP}}$ based on a RC scheme RC (Definition 33). The sets S, R and S_ack are as in Figure 4.6. The variable max-num represents the largest num received in an at. The counters cnt_at and $\mathsf{max\text{-}cnt}_\mathsf{at}$ keep track of how many tags have been sent and largest cnt_at received inside of a tag, respectively. $\mathsf{S}_\mathsf{at}\mathsf{Seen}$ is the list of cnt_at of received at since the last sent one; $\mathsf{S}_\mathsf{at}[i]$ contains the content of S sent in the ith at.

AuthReceive prunes $\mathsf{st}_\mathsf{P}.\mathsf{R}$ by removing elements in $\mathsf{R}^\mathsf{P}_\subseteq$; since the procedure authenticates the elements in $\mathsf{R}^\mathsf{P}_\subseteq$ at line 11, it is safe to prune $\mathsf{st}_\mathsf{P}.\mathsf{R}$. AuthReceive prunes $\mathsf{st}_\mathsf{P}.\mathsf{S}$ by removing elements in $\mathsf{st}_\mathsf{P}[i]$ for $i \in \mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\overline{\mathsf{P}}}$. The set $\mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\overline{\mathsf{P}}}$ contains counters of the authentication tags that P sent to $\overline{\mathsf{P}}$ and $\overline{\mathsf{P}}$ correctly received. Moreover, the AuthReceive procedure updates $\mathsf{st}.\mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\overline{\mathsf{P}}}$ at line 1, i.e., after the integrity checks. Since the AuthSend stores the set of sent messages S authenticated within the current at into the array S_{at}, pruning $\mathsf{st}_\mathsf{P}.\mathsf{S}$ only removes messages that have already been received and authenticated by $\overline{\mathsf{P}}$.

Recall that the adversary can only delete and replay authentication tags in the out-of-band channel. We informally discuss how the scheme handles these cases. Assume P and $\overline{\mathsf{P}}$ exchange some messages, P receives an authentication tag $\mathsf{at}^{\overline{\mathsf{P}}}$ from $\overline{\mathsf{P}}$ and then sends the authentication tag at^{P}; the adversary removes at^{P} from the channel. Since the adversary removes at^{P}, P does not acknowledge the reception of $\mathsf{at}^{\overline{\mathsf{P}}}$ to $\overline{\mathsf{P}}$ (because AuthSend empties $\mathsf{st}_\mathsf{P}.\mathsf{S}_{\mathsf{at}}\mathsf{Seen}$ at every invocation). Consequently, $\overline{\mathsf{P}}$ does not prune $\mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{S}$: these messages will be authenticated with the next authentication tag and security is preserved. The AuthReceive procedure handles adversarial reordering of authentication tags with counters $\mathsf{cnt}_{\mathsf{at}}$ at line 5.

We formally state our security claim for $\mathsf{ARC}_{\mathsf{OP}}$ below.

Theorem 14. Consider collision resistant hash function H used to build $\mathsf{ARC}_{\mathsf{OP}}$ (defined in Figure 4.15). Then, we have that for every efficient adversary $\mathscr{A}$, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathrm{unf}}_{\mathsf{ARC}_{\mathsf{OP}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{cr}}_{\mathsf{H}}(\mathscr{B}) .$$

Moreover, $\mathsf{ARC}_{\mathsf{OP}}$ is correct and ORDINALS secure.

Proof. Correctness and ORDINALS-security for the transformation of Figure 4.15 follow from the scheme in Figure 4.9.

The scheme is the same as Figure 4.9 modulo the optimisations we introduced. The proof of the theorem thus reduces to showing that the optimisations preserve the security properties of the unoptimised ARC scheme $\mathsf{ARC}_{\mathsf{base}}$ (Figure 4.9). Observe that $\mathsf{st}_\mathsf{P}.\mathsf{R}$ and $\mathsf{st}_\mathsf{P}.\mathsf{S}$ are used to detect active attacks. We start by showing that pruning these sets does not undermine UNF security.

- The set $\mathsf{st}_\mathsf{P}.\mathsf{R}$ is pruned by removing elements from $\mathsf{R}^\mathsf{P}_\subseteq$, which was authenticated on line 11 of AuthReceive. We therefore know that messages in $\mathsf{R}^\mathsf{P}_\subseteq$ are honest. Thus, we can stop sending them to $\overline{\mathsf{P}}$ hereafter.
- $\mathsf{st}_\mathsf{P}.\mathsf{S}$ is pruned by all sets $\mathsf{st}_\mathsf{P}.\mathsf{S}_{\mathsf{at}}[i]$ for $i \in \mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\overline{\mathsf{P}}}$. By construction we know that $\mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\overline{\mathsf{P}}}$ contains counters for which $\overline{\mathsf{P}}$ *accepted* the authentication tags, since those are included in line 1 of prune. Therefore, $\{\mathsf{st}_\mathsf{P}.\mathsf{S}_{\mathsf{at}}[i]\}_{i \in \mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\overline{\mathsf{P}}}}$ contains all messages in $\mathsf{st}_\mathsf{P}.\mathsf{S}$ that $\overline{\mathsf{P}}$ correctly received and authenticated, which the procedure stores in $\mathsf{st}_{\overline{\mathsf{P}}}.\mathsf{S}_{\mathsf{ack}}$ for future checks. Hence, P can safely stop sending those and prune S correspondingly.

We proceed by showing that UNF security still holds. By the arguments above, an authentication tag at that the AuthSend procedure generates *after* another authentication tag at′, will contain only messages that have *not* been authenticated in at′. Therefore the check on line 5 preserves security.

The check on line 9 verifies whether $\mathsf{R}^{\overline{\mathsf{P}}} \subseteq \mathsf{S}$. Without pruning, this property is met in the absence of forgeries as shown for $\mathsf{ARC}_{\mathsf{base}}$. Assume for contradiction that $\mathsf{R}^{\overline{\mathsf{P}}}$ contains a message not authenticated yet, but S does not contain this message due to pruning. This means that the message was removed from S by removing one of the values in $\mathsf{st}_{\mathsf{P}}.\mathsf{S}_{\mathsf{at}}$ whose counter $\mathsf{cnt}_{\mathsf{at}}$ was present in $\mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\overline{\mathsf{P}}}$. Since the counter is present in $\mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\overline{\mathsf{P}}}$, we know that $\overline{\mathsf{P}}$ accepted the authentication tag containing $\mathsf{cnt}_{\mathsf{at}}$, i.e., $\overline{\mathsf{P}}$ correctly received and authenticated the message. But this means by construction that $\overline{\mathsf{P}}$ pruned the message from R on line 2 of prune, which leads to a contradiction. Therefore the check preserves UNF security.

The check on line 11 verifies whether $\mathsf{R}^{\mathsf{P}}_{\subseteq} \subseteq \mathsf{S}^{\overline{\mathsf{P}}}$. Without pruning, this property is met in absence of forgeries as shown for ARC. Note that P removes from R only messages that have been authenticated (on line 2 of prune), therefore $\mathsf{R}^{\mathsf{P}}_{\subseteq}$ only contains unauthenticated messages. Similarly, by the argument presented in the paragraph above, $\mathsf{S}^{\overline{\mathsf{P}}}$ contains messages included in at's whose counter was not included in $\mathsf{S}_{\mathsf{at}}\mathsf{Seen}^{\mathsf{P}}$, and therefore unauthenticated messages. We conclude that this check also preserves UNF security. □

Note that $\mathsf{ARC}_{\mathsf{OP}}$ (Figure 4.15) sends all the authentication material through the out-of-band channel. This might be impractical when the authenticated out-of-band channel is narrowband, e.g., if parties use QR-codes to authenticate the communication. We can improve the scheme by using both channels: use the insecure channel to send the authentication data and the possibly narrowband authenticated channel to verify the integrity of the data, e.g., using the protocol of Pasini and Vaudenay [PV06]. While the idea of using both channels for authentication is natural [BSSW02], some security risks might arise when the scheme does not correctly match the two messages. Since UNF security depends on both the messages, and therefore on the messages being correctly matched, it might be safer to enforce this property at the scheme level which we leave as future work to formalise.

4.6.4 Lightweight Bidirectional Authentication

In this subsection, we propose a three-move bidirectional authentication protocol. Figure 4.16 describes the protocol at a high level. Parties include in the authentication tag only the set of *received* messages. The receiver of the tag compares then the set of received messages from the counterpart with the set of sent messages. We envision this approach could be used when participants meet in person or online and can both authenticate the respective views of the conversation at the same time. Signal, among other messaging solutions, already requires such an exchange for parties executing its safety numbers protocol that are aiming for mutual verification [Mar17].

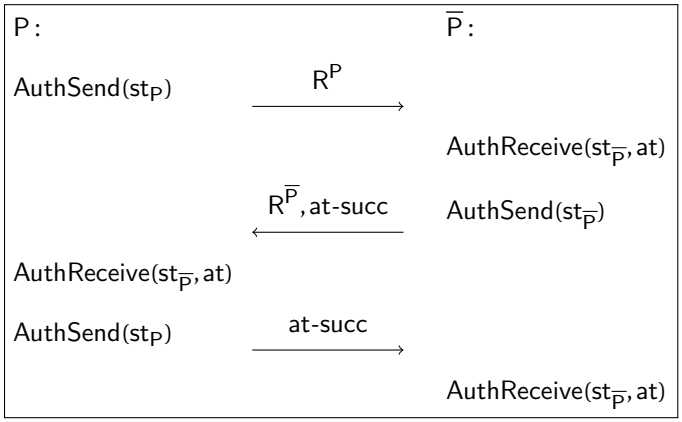

Figure 4.16: Overview of the three-move authentication procedure. The boolean at-succ indicates whether the counterpart's set of received messages is a subset of the caller's local set of sent messages.

In what follows, we propose an appropriate security model, describe the protocol in detail and provide a security proof in our model.

Security Model. We modify the UNF game (Section 4.4) by requiring the adversary to run authentication sessions in sequence. We present the corresponding game in Figure 4.17.

When the adversary, through the AUTHSEND′ oracle, starts the authentication protocol with initial sender P, the adversary's access to AUTH∗, SEND′ and RECEIVE′ oracles is restricted until the three-move authentication session is completed between P and $\overline{P}$. This is encoded in the next-oob-op variable. This simplifies the exposition, but it is not necessary in particular to restrict SEND′ and RECEIVE′ calls in practice, even though parties who authenticate in person would generally not send messages during this time. To handle this, parties can simply buffer messages during authentication that are authenticated in the next authentication session. However, buffering messages implies that an attack carried out during the out-of-band authentication will not be detected until the next authentication protocol. For this reason we block in-band communication during the three-move protocol and we encourage this restriction to be maintained also in practice. We note also that parties are guaranteed slightly weaker security than in the UNF game. Namely, after receiving the first authentication message, the receiver can deduce that their counterpart has not received a forgery but not that they themselves have until they receive the third message in the protocol.

In Definition 39 we define 3M-UNF-security for ARC schemes.

Definition 39 (3M-UNF)**.** An ARC ARC is 3M-UNF if, for all efficient adversaries $\mathscr{A}$, we have:

$$\mathsf{Adv}^{\mathrm{3m-unf}}_{\mathsf{ARC}}(\mathscr{A}) = \Pr[\mathsf{3M\text{-}UNF}_{\mathsf{ARC}}(\mathscr{A}) \Rightarrow 1] = \mathsf{negl}\,,$$

where game 3M-UNF is defined in Figure 4.17.

Game 3M-UNF$_{\mathsf{ARC}}(\mathscr{A})$

1: auth-state[$\cdot$] $\leftarrow 0$; next-oob-op $\leftarrow \bot$
2: play UNF$_{\mathsf{ARC}}$ with $\mathscr{A}^{\mathcal{O}}(\mathsf{pp}, z)$ and predicates forgery, bad-P, bad-$\overline{\mathsf{P}}$

Oracle SEND$'$(P, ad, pt, r)

1: **if** auth-state[P] $\neq 0$: **return** $\bot$
2: **return** SEND(P, ad, pt, r)

Oracle RECEIVE$'$(P, ad, ct)

1: **if** auth-state[P] $\neq 0$: **return** $\bot$
2: **return** RECEIVE(P, ad, ct)

Oracle AUTHSEND$'$(P)

1: **if** next-oob-op $\notin \{(\mathsf{P}, \text{"authsend"}), \bot\}$:
2: **return** $\bot$
3: $i \leftarrow i+1$
4: $(\mathsf{st}_{\mathsf{P}}, \mathsf{num}, \mathsf{at}) \xleftarrow{\$} \mathsf{AuthSend}(\mathsf{st}_{\mathsf{P}})$
5: auth[(P, i)] $\leftarrow$ at; state[i] $\leftarrow \mathsf{st}_{\mathsf{P}}$
6: Init $\leftarrow$ (auth-state[P] $\stackrel{?}{=} 0$) // Boolean
7: auth-state[P] $\leftarrow$ auth-state[P] $+ 1 \bmod 3$
8: log[i] $\leftarrow$ ("authsend", P, num, at, Init)
9: next-oob-op $\leftarrow (\overline{\mathsf{P}}, \text{"authrec"}, i)$
10: **return** (num, at)

forgery(log, P, num, ad, ct, x)

1: As in Figure 4.7

Oracle AUTHRECEIVE$'$(P, j)

1: **if** next-oob-op $\neq$ (P, "authrec", j) :
2: **return** $\bot$
3: at $\leftarrow$ auth[($\overline{\mathsf{P}}$, j)]
4: **if** at $= \bot$: **return** $\bot$
5: (auth, st, num) $\leftarrow$ AuthReceive(st_{P}, at)
6: **if** $\neg$auth : **return** $\bot$
7: $i \leftarrow i+1$
8: auth-state[P] $\leftarrow$ auth-state[P] $+ 1 \bmod 3$
9: **if** auth-state[P] = auth-state[$\overline{\mathsf{P}}$] $= 0$:
10: // Last step of the protocol completed
11: next-oob-op $\leftarrow \bot$
12: **else**
13: next-oob-op $\leftarrow$ (P, "authsend")
14: $\mathsf{st}_{\mathsf{P}} \leftarrow$ st; state[i] $\leftarrow \mathsf{st}_{\mathsf{P}}$
15: log[i] $\leftarrow$ ("authrec", P, num, at)
16: **return** num

bad-P(log, P, num$'$, at, x, y)

1: **return** $(y > x) \wedge$
2: ("authsend", $\overline{\mathsf{P}}$, num$'$, at, $\_$) $=$ log[y] $\wedge$
3: ("authrec", P, num$'$, at) $\in$ log

bad-$\overline{\mathsf{P}}$(log, P, num, num$'$, at)

1: **return** num $\leq$ num$'$ $\wedge$
2: ("authrec", $\overline{\mathsf{P}}$, num$'$, at) $\in$ log $\wedge$
3: ("authsend", P, num$'$, at, false) $\in$ log

Figure 4.17: 3M-UNF game for $\mathcal{O} = \{\mathsf{SEND}', \mathsf{RECEIVE}', \mathsf{AUTHSEND}', \mathsf{AUTHRECEIVE}', \mathsf{EXP}_{\mathsf{pt}}, \mathsf{EXP}_{\mathsf{st}}\}$. Highlighted statements correspond to differences relative to the UNF game (Figure 4.7).

The oracles in Figure 4.17 mandate that the participants send all messages in authentication via the out-of-band channel. By providing no security guarantees on the first message (i.e., delaying guarantees for the receiver until receiving the third message), it is possible to send the first message in the protocol over the in-band channel and then authenticate it in the second message with an additional hash [PV06]. Consequently, the protocol can be made essentially non-interactive out-of-band: the counterpart to the initiator can simply send the message out-of-band, and the bit can be determined easily via determining e.g. QR code scanning success/failure. By contrast, solutions like safety numbers require *both* parties to

scan QR-codes out-of-band.

Scheme Description. We present a 3M-UNF-secure scheme in Figure 4.18. The AuthSend and AuthReceive procedures encode the three-move authentication protocol of Figure 4.16. To identify the different states of the bidirectional authentication, we borrow the terminology from TCP and refer to SYN, SYN-ACK, and ACK messages and roles. When a party P first calls AuthSend, it takes the SYN role and sends to $\overline{\mathsf{P}}$ the set of *received* messages and the current num, i.e., at $\leftarrow (\mathsf{R}, \mathsf{num})$; this set is stored in a separate set R_{at}. As described below, we use R_{at} in AuthReceive to optimise the scheme. The counterpart $\overline{\mathsf{P}}$ replies with a SYN-ACK message, containing its set of received messages, the current ordinal num and the bit at-succ. The bit at-succ indicates whether P's set of received messages is included in $\overline{\mathsf{P}}$'s set of sent messages (line 10), i.e., at-succ indicates whether the authentication of P's set of received messages was successful. As the counterpart, $\overline{\mathsf{P}}$ stores the current set of received messages in R_{at}. Upon receiving the SYN-ACK message, the initiator P checks whether $\mathsf{at\text{-}succ}^{\overline{\mathsf{P}}} = \mathsf{true}$ and rejects the authentication tag otherwise. P then sends the ACK message at $\leftarrow (\mathsf{num}, \mathsf{at\text{-}succ})$. Finally, $\overline{\mathsf{P}}$ calls AuthReceive to process the ACK message. The party checks the at-succ variable to verify that the set $\mathsf{R}^{\overline{\mathsf{P}}}$ is a subset of S^{P}. If the check passes, the authentication protocol ends.

The optimisation of the scheme consists in pruning the set of received messages as soon as the counterpart authenticates them. This reduces the size of the authentication tags, since parties include in at only the received messages that have not been authenticated yet. The AuthReceive algorithm on line 11 checks whether the counterpart authenticated set of received messages R. If this is the case, all the authenticated messages are stored in $\mathsf{R}_{\mathsf{ack}}$—this set is used in the Receive algorithm to avoid replay attacks—and at the same time those messages are removed from R thanks to set R_{at}, thereby reducing the size of the next authentication tag and memory consumption. After the pruning, the R set contains only received messages that the counterpart *still* needs to authenticate.

Related Work. Dowling et al. [DGP22] propose a scheme that is broadly similar to ours. In particular, their protocol uses three moves in-band to allow parties to agree on a common set of respectively received messages R and R′. Then, to authenticate messages and detect active attacks, parties compare a hash $H(\mathsf{R}, \mathsf{R}')$ (with hash function H) out-of-band. Note however that they do not consider RID security and that they do not formally treat out-of-order message delivery.

The correctness of $\mathsf{ARC}_{\mathsf{3M}}$ can be shown using the underlying correctness of RC.

Security Analysis. ORDINALS security is inherited from the underlying RC scheme. As usual, we argue that 3M-UNF security follows from the collision resistance of the underlying hash function.

Theorem 15 (Unforgeability of $\mathsf{ARC}_{\mathsf{3M}}$)**.** Consider collision resistant hash function H used to build $\mathsf{ARC}_{\mathsf{3M}}$ (defined in Figure 4.18). Then, we have that for every efficient adversary $\mathscr{A}$, one

$\mathsf{ARC}_{\mathsf{3M}}.\mathsf{Setup}(1^\lambda)$

1 : // As in Figure 4.9
2 : **return** $\mathsf{ARC}_{\mathsf{base}}.\mathsf{Setup}(1^\lambda)$

$\mathsf{ARC}_{\mathsf{3M}}.\mathsf{Init}(\mathsf{pp})$

1 : $(\mathsf{pp}_0, \mathsf{hk}) \leftarrow \mathsf{pp}$
2 : $(\mathsf{st}'_\mathsf{A}, \mathsf{st}'_\mathsf{B}, z) \xleftarrow{\$} \mathsf{RC.Init}(\mathsf{pp}_0)$
3 : $\mathsf{num} \leftarrow \bot$
4 : $\mathsf{S}, \mathsf{R}, \mathsf{R}_{\mathsf{ack}}, \mathsf{R}_{\mathsf{at}} \leftarrow \emptyset$
5 : $\mathsf{role\text{-}at}, \mathsf{at\text{-}succ} \leftarrow \bot$
6 : $\mathsf{st}_\mathsf{A} \leftarrow (\mathsf{st}'_\mathsf{A}, \mathsf{hk}, \mathsf{S}, \mathsf{R}, \mathsf{num}, \mathsf{role\text{-}at},$
7 : $\quad \mathsf{at\text{-}succ}, \mathsf{R}_{\mathsf{ack}}, \mathsf{R}_{\mathsf{at}})$
8 : $\mathsf{st}_\mathsf{B} \leftarrow (\mathsf{st}'_\mathsf{B}, \mathsf{hk}, \mathsf{S}, \mathsf{R}, \mathsf{num}, \mathsf{role\text{-}at},$
9 : $\quad \mathsf{at\text{-}succ}, \mathsf{R}_{\mathsf{ack}}, \mathsf{R}_{\mathsf{at}})$
10 : $z \leftarrow (z', \mathsf{pp})$
11 : **return** $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z)$

$\mathsf{ARC}_{\mathsf{3M}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$

1 : // As in Figure 4.9
2 : **return** $\mathsf{ARC}_{\mathsf{base}}.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$

$\mathsf{ARC}_{\mathsf{3M}}.\mathsf{AuthSend}(\mathsf{st}_\mathsf{P})$

1 : $(\_, \_, \_, \mathsf{R}, \mathsf{num}, \mathsf{role\text{-}at}, \mathsf{at\text{-}succ}, \_, \mathsf{R}_{\mathsf{at}}) \leftarrow \mathsf{st}_\mathsf{P}$
2 : **if** $\mathsf{role\text{-}at} = \bot$:
3 : $\quad \mathsf{st}_\mathsf{P}.\mathsf{role\text{-}at} \leftarrow \mathsf{SYN}$
4 : $\quad \mathsf{at} \leftarrow (\mathsf{R}, \mathsf{num})$; $\mathsf{st}_\mathsf{P}.\mathsf{R}_{\mathsf{at}} \leftarrow \mathsf{R}$
5 : **elseif** $\mathsf{role\text{-}at} = \mathsf{SYN\text{-}ACK}$:
6 : $\quad \mathsf{at} \leftarrow (\mathsf{R}, \mathsf{num}, \mathsf{at\text{-}succ})$
7 : $\quad \mathsf{st}_\mathsf{P}.\mathsf{R}_{\mathsf{at}} \leftarrow \mathsf{R}$; $\mathsf{st}_\mathsf{P}.\mathsf{at\text{-}succ} \leftarrow \bot$
8 : **else** // $\mathsf{role\text{-}at} = \mathsf{ACK}$
9 : $\quad \mathsf{at} \leftarrow (\mathsf{num}, \mathsf{at\text{-}succ})$
10 : $\quad \mathsf{st}_\mathsf{P}.\mathsf{role\text{-}at}, \mathsf{st}_\mathsf{P}.\mathsf{at\text{-}succ} \leftarrow \bot$
11 : **return** $(\mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{at})$

$\mathsf{ARC}_{\mathsf{3M}}.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$

1 : $(\mathsf{st}_\mathsf{P}, \mathsf{hk}, \_, \mathsf{R}, \_, \_, \_, \mathsf{R}_{\mathsf{ack}}, \_) \leftarrow \mathsf{st}_\mathsf{P}$
2 : $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{num}, \mathsf{pt}) \leftarrow \mathsf{RC.Receive}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$
3 : **if** $\neg\mathsf{acc}$: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
4 : $h \leftarrow \mathsf{H.Eval}(\mathsf{hk}, (\mathsf{ad}, \mathsf{ct}))$
5 : **if** $\exists\, h' : (\mathsf{num}, h') \in \mathsf{R}_{\mathsf{ack}} \wedge h \neq h'$:
6 : $\quad$ **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \bot, \bot)$
7 : $\mathsf{R} \leftarrow \mathsf{R} \cup \{(\mathsf{num}, h)\}$
8 : $\mathsf{st}_\mathsf{P} \leftarrow (\mathsf{st}_\mathsf{P}, \mathsf{hk}, \_, \mathsf{R}, \_, \_, \_, \mathsf{R}_{\mathsf{ack}}, \_)$
9 : **return** $(\mathsf{acc}, \mathsf{st}_\mathsf{P}, \mathsf{num}, \mathsf{pt})$

$\mathsf{ARC}_{\mathsf{3M}}.\mathsf{AuthReceive}(\mathsf{st}_\mathsf{P}, \mathsf{at})$

1 : $(\_, \_, \mathsf{S}, \mathsf{R}, \_, \mathsf{role\text{-}at},$
2 : $\quad \mathsf{at\text{-}succ}, \mathsf{R}_{\mathsf{ack}}, \mathsf{R}_{\mathsf{at}}) \leftarrow \mathsf{st}_\mathsf{P}$
3 : $\mathsf{R}^{\overline{\mathsf{P}}} \leftarrow \emptyset$; $\mathsf{at\text{-}succ}^{\overline{\mathsf{P}}} \leftarrow \mathsf{true}$
4 : **if** $\mathsf{role\text{-}at} = \bot$:
5 : $\quad \mathsf{role\text{-}at} \leftarrow \mathsf{SYN\text{-}ACK}$; $(\mathsf{R}^{\overline{\mathsf{P}}}, \mathsf{num}^{\overline{\mathsf{P}}}) \leftarrow \mathsf{at}$
6 : **elseif** $\mathsf{role\text{-}at} = \mathsf{SYN}$:
7 : $\quad (\mathsf{R}^{\overline{\mathsf{P}}}, \mathsf{num}^{\overline{\mathsf{P}}}, \mathsf{at\text{-}succ}^{\overline{\mathsf{P}}}) \leftarrow \mathsf{at}$
8 : **else** // receive ACK case
9 : $\quad (\mathsf{num}^{\overline{\mathsf{P}}}, \mathsf{at\text{-}succ}^{\overline{\mathsf{P}}}) \leftarrow \mathsf{at}$
10 : $\mathsf{at\text{-}succ} \leftarrow (\mathsf{R}^{\overline{\mathsf{P}}} \stackrel{?}{\subseteq} \mathsf{S})$ // Boolean
11 : **if** $\mathsf{at\text{-}succ}^{\overline{\mathsf{P}}}$:
12 : $\quad \mathsf{R}_{\mathsf{ack}} \leftarrow \mathsf{R}_{\mathsf{ack}} \cup \mathsf{R}_{\mathsf{at}}$; $\mathsf{R} \leftarrow \mathsf{R} \setminus \mathsf{R}_{\mathsf{at}}$
13 : $\quad \mathsf{R}_{\mathsf{at}} \leftarrow \emptyset$
14 : **else** // failure
15 : $\quad$ **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \mathsf{num})$
16 : $\mathsf{st}_\mathsf{P} \leftarrow (\_, \_, \mathsf{S}, \mathsf{R}, \mathsf{num}, \mathsf{role\text{-}at},$
17 : $\quad \mathsf{at\text{-}succ}, \mathsf{R}_{\mathsf{ack}}, \mathsf{R}_{\mathsf{at}})$
18 : **return** $(\mathsf{at\text{-}succ}, \mathsf{st}_\mathsf{P}, \mathsf{num}^{\overline{\mathsf{P}}})$

Figure 4.18: Optimised 3M-UNF-secure ARC scheme $\mathsf{ARC}_{\mathsf{3M}}$ based on a RC scheme RC (Definition 33). $\mathsf{ARC}_{\mathsf{base}}$ refers to the unoptimised ARC defined in Figure 4.9. We assume $\mathsf{ARC}_{\mathsf{base}}.\mathsf{Send}$ updates local variable num. As before, the representation of R communicated can be optimised to contain only a single hash.

can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathrm{3m-unf}}_{\mathsf{ARC}_{\mathsf{3M}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{cr}}_{\mathsf{H}}(\mathscr{B}) \ .$$

Proof. We proceed similarly to the proof of Theorem 10. Without loss of generality, we analyze the authentication of P, who we assume to be the initiator, towards $\overline{\mathsf{P}}$. The adversary cannot call the SEND' and $\mathsf{RECEIVE}'$ oracles once the authentication process is started, therefore the sets S and R of both parties are fixed until the completion of the protocol.

To authenticate the set of received messages R^{P}, P first sends $\mathsf{at} \leftarrow (\mathsf{R}^{\mathsf{P}}, \mathsf{num})$ to $\overline{\mathsf{P}}$. To verify the authenticity of R^{P}, the party $\overline{\mathsf{P}}$ verifies whether $\mathsf{R}^{\mathsf{P}} \subseteq \mathsf{S}^{\overline{\mathsf{P}}}$. By the arguments of the proof for Theorem 10, this reduces to the collision-resistance of the hash function H. After receiving the first tag, $\overline{\mathsf{P}}$ is able to detect forgeries received by P but not by itself. This is taken into account in the 3M-UNF game (line 3 in $\mathsf{bad}\text{-}\overline{\mathsf{P}}$), which states that a forgery received by $\overline{\mathsf{P}}$ is valid only if it is not detected after receiving the second or third tag in the authentication process. Then, when receiving the second tag $(\mathsf{R}^{\overline{\mathsf{P}}}, \mathsf{num}^{\overline{\mathsf{P}}}, \mathsf{at\text{-}succ}^{\overline{\mathsf{P}}})$ from $\overline{\mathsf{P}}$, P is able to tell if itself received a forgery if $\mathsf{at\text{-}succ}^{\overline{\mathsf{P}}} = \mathsf{false}$.[4] By the same arguments as before, P can tell whether $\overline{\mathsf{P}}$ received a forgery by checking $\mathsf{R}^{\overline{\mathsf{P}}} \subseteq \mathsf{S}^{\mathsf{P}}$. Finally, upon receiving the third tag, $\overline{\mathsf{P}}$ can detect a forgery using $\mathsf{at\text{-}succ}^{\mathsf{P}}$.

The optimisation maintains 3M-UNF-security. Recall that the goal of the optimisation (lines 11-13 in Figure 4.18) is to reduce the size of the R set by storing authenticated messages in $\mathsf{R}_{\mathsf{ack}}$. To achieve this reduction, the party executing $\mathsf{AuthReceive}$ removes from R the set R_{at}, which is the set of received messages authenticated by the counterpart through the $\mathsf{at\text{-}succ}$ variable. Since by construction all the messages in R_{at} have been already authenticated by the counterpart, removing them from R does not remove unauthenticated messages from R. □

[4] In fact, in our formalism, unsuccessful $\mathsf{AuthReceive}$ calls are not modelled, so $\mathsf{at\text{-}succ}^{\overline{\mathsf{P}}}$ will always be true, but we argue this way to be more general.

Group Messaging Part II

5 Cryptographic Administration for Secure Group Messaging

In this chapter, we formalise administration in group messaging by extending *continuous group key agreement* to capture group administrators. An extended abstract corresponding to this work appeared at USENIX Security 2023, and was joint work with David Balbás and Serge Vaudenay [BCV23]. A full version of this work can be found on the Cryptology ePrint Archive [BCV22]. The author of this thesis was an advisor of David Balbás for his MSc thesis which contains some preliminary results [Bal21] on group administration that eventually led to this publication after significant revision.

5.1 Contribution

One of the major challenges in the design of group messaging protocols is the need to account for group *evolution* or *dynamics*, particularly to prevent membership changes from diminishing the confidentiality of sent messages. Since the list of group members may change at any point in time, complex key agreement protocols are required. Overall, securing group membership involves three main aspects: (1) *key updates*, ensuring that new members cannot read past messages and removed members cannot read future messages; (2) *membership consistency*, ensuring that all members faithfully know the list of members at any time; and (3) *securing control messages* (i.e., notifications for member addition and removal operations) from active adversaries and from the delivery service itself (i.e., ensuring authentication).

Many state-of-the-art protocols, including passively-secure continuous group key agreement (CGKA) protocols [ACDT20, KPPW$^+$21], Sender Keys (WhatsApp, Signal Messenger) [Wha20, BCG23b] and Matrix [ACDJ23], include cryptographic mechanisms for securing key updates, but provide weaker and sometimes even no guarantees for securing membership consistency and control messages. We identify membership consistency as both a correctness and a security property that is critical for confidentiality (otherwise, the sender of a message may not know the receivers) but is often ignored in the literature. Failing to secure control messages can also result in catastrophic attacks. Practical examples include the *burgle into a group attack* [RMS18], which exploits the lack of authentication of control messages to allow an

adversary with partial control over the central server to enter arbitrary group chats in Signal and WhatsApp. Recent attacks on the Matrix protocol [ACDJ23, ADJ24] make use of similar vulnerabilities, enabling the server to take over the control of a group.

In order to secure group membership, we observe that there is a strong trend in practice to distinguish between at least two types of users in a group: group administrators (admins) and standard users. In groups with administrators, all group changes are either performed or approved by the admins. Therefore, we address the problem of secure group management by developing a cryptographic framework for *group administration*.

Administration in Messaging Apps. Generally, an admin has all the capabilities of a standard user plus a set of administrative rights. In practice, admins are implemented at the application level via policies enforced either by the central server or users. Examples are the popular messaging apps Signal, Telegram and WhatsApp (as of 2024).

- In WhatsApp, only admins can add and remove users, create a group invite link, and govern the admin subgroup. All groups must have at least one admin, and in particular when the last admin leaves, a user is selected randomly as the new admin.
- In Telegram, the group creator can designate other admins with diverse sets of capabilities. Besides adding and removing users, admins can impose partial bans on any user's capabilities, such as sending or receiving messages, and can even restrict the content that users can send [Tel].
- In Signal Messenger, admins can specify whether all members or only admins can add and remove users from a group (in the latter case non-admins can request to add users) and create a group invite link.

Despite administration mechanisms being widely deployed, there is little mention of admins in the literature. Existing CGKA and group messaging approaches make no formal distinction between admin and non-admin users, which results in giving admin capabilities to all users.

Security Goals. There are four main security goals that our cryptographic administrators aim to achieve. In groups where no distinction is made between admins and standard members, our solutions can be extended to the whole group by treating all members as admins; these goals nonetheless still apply:

- Reduce trust on the delivery service, such that it has no control over group administration and membership.
- Mitigate the impact of insider attacks [KS05, AJM22] on protocol execution. Insider adversaries, or compromised group members, will not be able to gain control of a group unless they are administrators[1].

[1] Note that denial-of-service attacks from malicious non-admin insiders as in [ACJM20] are not necessarily

- Increase the resilience of implementations of messaging protocols, preventing pitfalls such as the *burgle into a group attack* [RMS18] or the recent attacks on Matrix [ACDJ23].
- Reduce concurrency issues, especially when the delivery service is not a central server [WKHB21], since only a reduced set of members are able to commit group changes.

Admin Capabilities. Let $G = \{ID_1, \ldots, ID_n\}$ be a group of users participating in messaging or continuous key exchange and $G^* \subseteq G$ be a non-empty subset of group administrators. Unlike regular group members, the administrators $ID \in G^*$ that we consider can: (1) add and remove members from the group, (2) approve/reject join and removal requests, (3) designate other administrators, (4) give up their admin status, (5) remove the admin status of other users. For performance and security, regular group members should be able to remove themselves and make key updates without admin approval.

These correspond to the common administration features among the solutions presented above. In the case of Telegram, their "fine-grained administration" is practical due to the lack of end-to-end encryption. By default, Telegram relies on a central server that decrypts all messages, which is incompatible with our schemes.

5.1.1 Summary

In this chapter, we cast group administration as a formal *cryptographic* problem. The complexity of secure messaging requires modular constructions and proofs of security; our main goal in this chapter is to provide these. Our core contributions are as follows.

1. We introduce the *administrated* CGKA (A-CGKA) primitive in Section 5.2 by extending the continuous group key agreement (CGKA) primitive.
2. We introduce a novel game-based correctness notion for both CGKA and A-CGKA in Section 5.2.3 that emphasises the role of *group dynamics* which we argue is centrally linked to group administration.
3. Extending existing CGKA key indistinguishability security notions, we introduce a game-based security notion (Section 5.2.4) which further aims to prevent even fully compromised non-admin users from modifying group membership.
4. We present two A-CGKA constructions, *individual admin signatures* (IAS) and *dynamic group signature* (DGS), each built on top of a CGKA protocol. Each approach provides different security and efficiency properties. In Section 5.3, we formalise both protocols in detail and prove their correctness and security. We analyse their performance in Section 5.4.1.

prevented; we also remark that this family of attacks does not affect confidentiality. This issue is discussed in later sections.

5. We propose an extension to MLS in Section 5.3.4 that provides efficient secure administration that we also implement and benchmark locally (Section 5.4.1).

5.1.2 Technical Overview

From CGKA to A-CGKA. Inspired by the MLS standard (and later draft versions in particular), CGKA has been increasingly formalised in the so-called *propose and commit* paradigm [ACJM20, AJM22, ACDT21a]. In CGKA, each user maintains a state which is input to and updated by local CGKA algorithms. Users in a given group can create proposal messages to propose to add or remove users, or to update their keying material for PCS reasons. Proposals are then combined by a member to form a *commit* message, which is ordered and distributed to all group members via a centralised *delivery service*. This is then *processed* by users which make the committed changes effective.

We extend CGKA to A-CGKA to support administration on the primitive level. We support additional proposal types, namely for adding and removing admins, as well as for admin key updates. In A-CGKA, only admins can make admin proposals, and moreover only admins can authorise all types of commit messages. That is, users only process group changes that have been attested by an admin (except for users leaving the group by themselves, which can always be processed).

Correctness. Our notion of A-CGKA correctness enforces that users that process the same sequence of commit messages for a given group derive consistent views of both the evolution of group members and admins, and also of the shared key. Our game explicitly checks that group membership can change only through processing *commit* messages. We also enforce that honest proposals have their intended effect when embedded in commit messages upon processing.

Early CGKA game-based correctness notions were limited to key consistency [ACDT20] or were not formally specified [KPPW$^+$21]. It is only recently that group evolution guarantees have been considered, notably in the latest version of Alwen et al. [AJM20, AJM22] which works in the UC and the monolithic security definition of Alwen et al. [ACDT21b]. We emphasise this because, given the length of our IAS construction and corresponding correctness proof, it is possible that subtle bugs concerning group evolution are hidden in existing group messaging constructions and implementations.[2] In particular, we spotted (resolved) inconsistencies while trying to prove our own protocols correct.

Security. Our security notion captures two core guarantees. Firstly, like previous work [ACDT20, KPPW$^+$21], we consider a key indistinguishability game where the adversary drives CGKA execution via oracles and may compromise parties. We prevent the adversary from winning the game trivially by restricting its behaviour using protocol-dependent clean-

[2] For example, in Figure 8 (SGM) and Figure 17 (RTreeKEM) in Alwen et al. [ACDT21b], commit processing is not well-defined for a user who is processing their own removal from the group.

ness predicates, similar to previous work on messaging and key exchange.

Secondly, differing from 'standard' CGKA, we require that the adversary is unable to forge an (admin) commit message that results in a change in group structure for the processing party, even if the adversary knows the group key. We model this by allowing the (semi-active) adversary to inject commit messages to particular parties which process the messages (albeit without updating their state). The adversary can adaptively expose the states of participants and make challenge queries. Security is ensured insofar as the adversary does not trivially compromise an administrator, so in particular they can compromise many non-admins. As in the case of key indistinguishability, we also specify a separate admin cleanness predicate to capture trivial attacks for this attack vector. Our security notion allows for FS and PCS guarantees with respect to the admin keying material.

Constructions. In this chapter, we provide two separate, modular constructions of A-CGKA from a CGKA that we describe in Section 5.3. We also introduce an extension of MLS that supports administration. The security of the authentication mechanisms in all our protocols matches the FS and PCS demands of modern group messaging.

In our first construction, individual admin signatures (IAS) (Section 5.3.1), admins keep track of their own signature key pair. Admin proposals and commits which change the group structure or admin structure or keys are signed using the committing admin's signature key. Admins update their signature keys via admin update proposals or by crafting commit messages.

Our second construction, dynamic group signature (DGS) (Section 5.3.2), relies on a secondary CGKA and authenticates a group of admins as a whole (possibly all group members). This highlights the fact that administration can be done without authenticating single users. Instead of maintaining individual signatures, admins instead execute within this CGKA and use the common secret to derive a signature key pair for each epoch. Non-admins keep track of the signature public key over time and verify that commits are signed using it.

Proofs. We formally prove that our protocols IAS and DGS are correct and secure with respect to our A-CGKA definitions. We prove IAS secure with respect to a sub-optimal admin cleanness predicate (somewhat weak forward security). We argue that the protocol and proof can be very easily modified to satisfy optimal security using forward-secure signatures [BM99] with no asymptotic overhead; this is discussed in Section 5.3.1. By contrast, in DGS, forward security is inherited from the admin CGKA.

MLS Extension. In Section 5.3.4, we embed the MLS protocol with A-CGKA functionality more organically by making use of MLS's credential infrastructure. We describe the main modifications needed and propose an extension of MLS that admits secure administration. Moreover, we implement and benchmark the efficiency of our MLS extension (and include a reference to the source code); we present our results in Section 5.4.1.

5.1.3 Additional Related Work

The TreeKEM protocol in MLS, initially proposed by Bhargavan et al. [BBR18], was inspired by Asynchronous Ratchet Trees [CCG+18]. Later, variants of TreeKEM arose like Tainted TreeKEM [KPPW+21], Insider-Secure TreeKEM [AJM22], Re-randomised TreeKEM [ACDT20], and Causal TreeKEM [Wei19]. MLS as a whole is studied by Brzuska et al. [BCK22] and Alwen et al. [ACDT21a].

CGKAs have been recently used to formally build full group messaging protocols [ACDT21a]. Besides TreeKEM, several CGKA variants have been proposed [ACJM20, AHKM22, WKHB21, AAN+22b]. Side works deal with multi-group security [CHK21], efficient key schedules for multiple groups [AAB+21], and concurrency [BDR20]. Separately, Rösler et al. [PRSS21] surveys group key exchange protocols. Group admins were considered by Rösler et al. [RMS18], although without a formal cryptographic approach, instead opting for property-based security notions described more informally akin to a symbolic model of security.

An alternative approach towards securing group membership was taken in the Signal Private Group System [CPZ20] which we discuss in Section 5.4.2.

5.2 (Administrated) Continuous Group Key Agreement

In this section, we introduce *continuous group key agreement* (CGKA) and then extend it to formalise our *administrated CGKA* (A-CGKA) primitive. We also introduce our correctness and security definitions for both CGKA and A-CGKA.

5.2.1 Continuous Group Key Agreement

The aim of the continuous group key agreement (CGKA) primitive [ACDT20] is to provide shared secrets (denoted by k) to dynamic groups of users over time. In CGKA, each group, labelled with a group identifier gid, is subject to additions (add), removals (rem), and user state refreshes/key updates (upd).

The definition of CGKA is introduced below, in the so-called *propose and commit* paradigm [AJM22, ACJM20], in which different operation proposals in a given group (for standard CGKA, including add, remove and key update operations) are collated into a commit message by a group member which is processed by users. The evolution of a CGKA in time is captured by *epochs*; a group member advances to a new epoch every time they successfully process a commit message, at which point there may be a change in the shared secret, in which case there may also be a change in the group structure from their perspective.

Note that the primitive is *stateful*: each user keeps their own state γ and calls each of the following algorithms locally which may update the state.

Definition 40. A continuous group key agreement (CGKA) scheme is a tuple of algorithms

CGKA = (Init, Create, Prop, Commit, Proc, Prop-Info, Props) such that:

- $\gamma \xleftarrow{\$} \mathsf{Init}(1^\lambda, ID)$: The initialisation function takes as inputs a security parameter 1^λ and an identity ID and outputs an initial state γ.
- $(\gamma', T) \xleftarrow{\$} \mathsf{Create}(\gamma, \mathsf{gid}, G)$: The group creation algorithm takes as inputs a state γ, a group identifier gid, and a list of group members $G = \{ID_1, \ldots, ID_n\}$ and outputs a new state γ' and a control (welcome) message T, where $T = \perp$ indicates failure.
- $(\gamma', P) \xleftarrow{\$} \mathsf{Prop}(\gamma, \mathsf{gid}, ID, \mathsf{type})$: The proposal algorithm takes as input a state, a group identifier, an ID, and a proposal type $\mathsf{type} \in \mathsf{types} = \{\mathsf{add}, \mathsf{rem}, \mathsf{upd}\}$, and outputs a new state γ' and a proposal message P, where $P = \perp$ indicates failure.
- $(\gamma', T, k) \xleftarrow{\$} \mathsf{Commit}(\gamma, \mathsf{gid}, \vec{P})$: The commit algorithm takes as inputs a state, a group identifier, and a vector of proposals $\vec{P}$, and outputs a new state γ', a control message T where $T = \perp$ indicates failure, and the (possibly new) group secret k.
- $(\gamma', \mathsf{acc}) \leftarrow \mathsf{Proc}(\gamma, T)$: The processing algorithm takes as inputs state and a control message T, and outputs a new state γ' and an acceptance bit $\mathsf{acc} \in \{\mathsf{true}, \mathsf{false}\}$, where $\mathsf{acc} = \mathsf{false}$ indicates failure.
- $(\mathsf{gid}, \mathsf{type}, ID, ID') \leftarrow \mathsf{Prop\text{-}Info}(\gamma, P)$: The proposal information algorithm takes as inputs a state and a proposal P, and outputs the group identifier of the proposal gid, its type type, the ID of the user *affected* by the proposal and the proposal creator ID'.
- $\vec{P} \leftarrow \mathsf{Props}(\gamma, T)$: The proposal extractor algorithm takes as inputs a state and control message T and outputs the vector of proposals $\vec{P}$ associated with T, where $\vec{P} = \perp$ indicates failure.

Finally, given ID's state γ and gid, the (possibly empty) set of group members in gid from ID's perspective is stored as $\gamma[\mathsf{gid}].G$, and the group secret k for gid is $\gamma[\mathsf{gid}].k$.

Protocol Execution. For simplicity, we assume that all users and groups are associated with a unique identifier ID and gid, respectively, which is, e.g., enforced by the PKI. Once every user has initialised their state using Init, a group is created when some party calls Create with some gid and a list of IDs as inputs. The Init algorithm can also serve to authenticate and register keys on a PKI when appropriate [ACDT20, KPPW+21, ACJM20]. In Section 5.3, we expand on authentication: for each protocol, we describe our assumptions on the PKI. The Create algorithm outputs a control message T that must be processed by prospective group members, including the group creator, to join the group gid.

In our formalism, any user can propose a member addition (add), member removal (rem) or key update (upd, only available for the caller) at any time. This is done via the Prop method, which outputs a proposal message P. Proposals encode the information needed to make a

change in the group structure or keying material, but the encoded changes are not immediately applied to the group. We emphasise that only the caller of Prop can use argument type = upd to propose to update (i.e., refresh) their keying material, in which case the input ID is ignored. Following Alwen et al. [ACDT21a], we define Prop-Info which outputs proposal attributes, rather than allowing for their direct access, to support possibly encrypted proposals (e.g., as in a `PrivateMessage` in the MLS standard [BBR+23]).

Proposed changes become effective once a user commits a (possibly empty) vector of proposals $\vec{P} = (P_1, \ldots, P_m)$ using Commit; we assume that proposals are suitably propagated in the group. The Commit algorithm outputs the new group key k and a control message T that contains the information needed by all current and incoming group members to process the changes. Typically, the Commit algorithm also updates the keying material of the caller. Control messages are processed via Proc, which updates the caller's state and outputs a bit acc indicating success or failure. We note that Proc and Props do not require a group identifier as input; this models the standard behaviour of a messaging protocol where, upon reception of a message, the user needs to determine which group the message corresponds to. In the event that a group member needs to parse the proposals in a commit message T without processing it, it can do so via the Props algorithm.

Example. Consider 5 parties $\{ID_1, \ldots, ID_5\}$ executing a CGKA protocol. After they each initialise their states as $\gamma_i \xleftarrow{\$} \mathsf{Init}(1^\lambda, ID_i)$, the following actions take place:

1. ID_1 calls $\mathsf{Create}(\gamma_1, \mathsf{gid}, \{ID_1, ID_2, ID_3, ID_4\})$, which updates γ_1 and outputs a control message T_0. At this stage, the group is still empty, $G = \emptyset$.

2. Each ID_i (including ID_1) processes the group creation as $\mathsf{Proc}(\gamma_i, T_0)$, which updates each state γ_i and outputs acc = true to each user. At this stage, $G = \{ID_1, ID_2, ID_3, ID_4\}$, and the group members share a common secret k_1.

3. Several users propose changes in the group:

   - ID_2 proposes to add ID_5 to the group by calling $(\gamma_2, P_1) \xleftarrow{\$} \mathsf{Prop}(\gamma_2, \mathsf{gid}, ID_5, \mathsf{add})$.
   - ID_3 wants to update its keying material and thus calls $(\gamma_3, P_2) \xleftarrow{\$} \mathsf{Prop}(\gamma_3, \mathsf{gid}, ID_3, \mathsf{upd})$.
   - ID_1 proposes to remove ID_4 from the group and calls $(\gamma_1, P_3) \xleftarrow{\$} \mathsf{Prop}(\gamma_1, \mathsf{gid}, ID_4, \mathsf{rem})$.

   This is shown in Figure 5.1. The group remains the same.

4. ID_2 collates all proposals in a commit message by calling $(\gamma_2, T_1, k_2) \xleftarrow{\$} \mathsf{Commit}(\gamma_2, \mathsf{gid}, (P_1, P_2, P_3))$. The group remains the same, since parties have not yet processed T_1.

5. All parties process T_1 by calling $\mathsf{Proc}(\gamma_i, T_1)$ and updating their states. Now, $G = \{ID_1, ID_2, ID_3, ID_5\}$ and these members share a new common secret k_2, which is not

known to ID_4. In addition, ID_3 (due to the update) and ID_2 (due to the commit) have refreshed their CGKA keying material for FS and PCS.

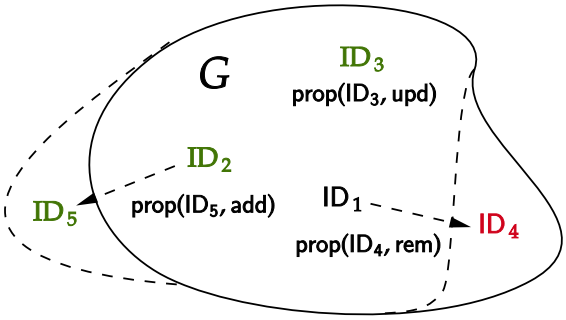

Figure 5.1: Diagram of a sample CGKA execution with 5 parties. Parties in green (ID_2, ID_3, ID_5) will update their keying material after ID_2 commits, while ID_4 will leave the group.

Commit Semantics. We assume that proposals input to Commit are processed in some deterministic, publicly-known, a priori determined order, that we call the *policy*. It is possible to extend the syntax of A-CGKA with a dedicated policy algorithm that defines this order [Bal21].

Alternative Definitions. In some CGKA definitions (especially older ones [ACDT20, KPPW+21]) and older MLS drafts, group changes are made effective immediately by processing proposals. To this end, the Commit and Prop algorithms are replaced by specific 'action' algorithms such as Add, Remove, Update. The *propose and commit paradigm* used in this chapter was introduced in draft 8 of MLS to allow for multiple operations to be applied at the same time to improve latency and concurrency support [BBR+23]. As mentioned in [KPPW+21], the older protocols can be written in the propose and commit paradigm, which is more flexible (and also better suited for group administrators as we will see).

A relevant difference between the definition in Alwen et al. [ACDT20] (and other game-based formulations such as in [KPPW+21]) and ours is that we work in the multi-group setting, and so we consider group identifiers of the form gid. Multi-group CGKAs have not been fully formalised in the literature, although multi-group security has been studied [CHK21], as well as efficient key schedules for multiple groups [AAB+21]. According to our definition, a user can be in many groups, identified by different values gid and interleave operations from each group arbitrarily. There are formulations of CGKA in the universal composability (UC) model [ACJM20, AJM22] which use group identifiers. In fact, composability guarantees in the UC model rely on the existence of unique, a priori established 'session identifiers' [Can01, KT11]; these can be established in practice via a central server or a distributed protocol [BLR04]. Note Cremers et al. [CHK21] considers cross-group security for messaging but does not treat CGKA as a primitive formally.

Finally, we note that there are other small differences in the literature. The semantics of Tainted TreeKEM [KPPW+21] enable users to speculatively execute operations; our syntax could be

modified to support this. In works on CGKA such as Alwen et al. [ACDT20], it is typical to make a distinction between standard commit and welcome messages. We implicitly incorporate this distinction in our constructions, but avoid it in the primitive syntax for simplicity. In some works [AJM22, ACJM20], additional algorithms such as GetKey are provided; we treat k as a state variable instead, which is output by Commit. As mentioned, it is also possible to conceive a policy algorithm as in [Bal21] that determines what the effect of a sequence of proposals should be (e.g., which proposals are prioritised, removing duplicates, etc.). Groups in some works [ACDT21a, AJM22], as well as the MLS standard itself, are initially of size 1: since initialising groups with several participants at once is common in practice, we believe it is of interest to model it. Different works formalise the role of the PKI to different degrees: Alwen et al. and Alwen et al. [ACDT21a, AJM22] consider an explicitly modelled PKI where users can choose their own possibly malicious keying material. Nevertheless, all works on CGKA to date formally assume that the PKI acts consistently for all users.

5.2.2 Administrated CGKA

An administrated continuous group key agreement (A-CGKA) is a CGKA where only a group G^* of *ID*'s, the so-called *group administrators*, can commit (and therefore make effective) changes to the group structure, such as adding and removing users. As with the group of users G in both CGKA and A-CGKA, the group of administrators G^* is dynamic.

Definition 41. An administrated continuous group key agreement (A-CGKA) scheme is a tuple of algorithms A-CGKA = (Init, Create, Prop, Commit, Proc, Prop-Info, Props) such that:

- Algorithms Init, Proc, Prop-Info, Props are defined as for a CGKA (Definition 40).
- In Prop and Prop-Info, types is redefined as types = {add, rem, upd, add-adm, rem-adm, upd-adm}.
- $(\gamma', T) \xleftarrow{\$} \mathsf{Create}(\gamma, \mathsf{gid}, G, G^*)$ additionally takes as input a group of admins G^*.
- $(\gamma', T, k) \xleftarrow{\$} \mathsf{Commit}(\gamma, \mathsf{gid}, \vec{P}, \mathsf{com\text{-}type})$ additionally takes as input a commit type $\mathsf{com\text{-}type} \in \mathsf{com\text{-}types} = \{\mathsf{std}, \mathsf{adm}, \mathsf{both}\}$.

Given *ID*'s state γ and gid, $\gamma[\mathsf{gid}].G$ and $\gamma[\mathsf{gid}].k$ are defined as in Definition 40, and $\gamma[\mathsf{gid}].G^*$ stores the set of admins in gid from *ID*'s perspective.

The execution of an A-CGKA is analogous to CGKA. Besides the introduction of the group of admins G^*, we introduce three additional proposal types add-adm, rem-adm, and upd-adm which concern administrative changes. Namely, an admin can propose to add another admin to the group of administrators, revoke the admin capabilities from a party, or update their administrative keying material, respectively. The commit type com-type specifies the scope of a commit operation, that is, whether it affects the general group (std), the administration of

the group (adm), or both at the same time (both). For the latter, a simple example is when an admin is both adding a member (group modification) and refreshing its admin keys (admin modification).

We note that the Create algorithm, enforced by our correctness and security notions to come, will require the condition $\emptyset \subset G^* \subseteq G$. Thus, the group administrators are always a subset of the group members. We take this approach following previous CGKAs [ACDT20, KPPW+21, ACJM20] and group messaging protocols [BBR+23, ACDT21a] where only group members can perform commits or make changes in the group. In these works and ours, it is (computationally) impossible for an external user to administrate a group, since external commits are not permitted.

Real-World Administrators. A-CGKA captures the core admin features of popular applications such as WhatsApp and Signal as mentioned in the introduction. We remark that the fact that non-admins are not allowed to make changes (except for leaving a group) is a desired consequence of our formulation of A-CGKA. A more fine-grained conceivable solution could be to allow admins to send a policy change proposal, to, e.g., modify the ability of all members to call Commit to add new users.

5.2.3 Correctness

Due to their similarity, we define the correctness of CGKA and A-CGKA together. Correctness of an (A)-CGKA scheme (A)-CGKA under the notion CORR is defined by game $\mathsf{CORR}_{\mathsf{(A)\text{-}CGKA},\mathsf{C}_{\mathsf{corr}}}$ played by adversary $\mathscr{A}$ in Figure 5.2. The main properties captured by the game are the following:

- *View consistency:* All users who transition to the same epoch (i.e., those which process the same sequence of commit messages) have the same group view (i.e., G, G^* and key k).
- *Message processing:* The group structure (G, G^*) and k can only be modified due to calls to Proc.
- *Forking states:* If the group is partitioned into subgroups that process different sequences of commit messages (thus leading to different group views), the game ensures that members in each partition have consistent views.
- *Multiple groups:* The adversary may create groups via CREATE on behalf of different users, and interact with different *ID*s in multiple groups.

Separately, we ensure that a user's state is not modified whenever a particular algorithm call fails. As observed for two-party messaging [BSJ+17], we require incorrect inputs to not affect the functionality of the protocol, and in particular to not cause a denial of service.

Game $\mathsf{CORR}_{\text{(A)-CGKA},\mathsf{C}_{\mathsf{corr}}}(\mathscr{A})$

1: **public** ep-view[·], ST[·], T[·] ← ⊥
2: **public** first-crt[·] ← ⊥
3: **public** prop-ctr, com-ctr ← 0 // msg counters
4: **public** ep[·] ← (−1, −1) // user epoch tracker
5: win ← 0
6: ST[ID] $\xleftarrow{\$}$ Init(1^λ, ID) ∀ ID
7: $\mathscr{A}^{\mathcal{O}}$
8: **require** $\mathsf{C}_{\mathsf{corr}}$ // optional predicate
9: **return** win // 1 if $\mathscr{A}$ is rewarded

Oracle CREATE(ID, gid, G, G^*)

1: (γ, T) $\xleftarrow{\$}$ Create(ST[ID], gid, G, G^*)
2: **if** $T = \bot$: **return**
3: **reward** $\neg(\emptyset \neq G^* \subseteq G)$
4: CheckSameGroupState(ST[ID], γ, gid)
5: T[gid, (−1, −1), com, ++com-ctr] ← T
6: ST[ID] ← γ

Oracle DELIVER(ID, gid, (t, c), c')

1: **require** ep[gid, ID] ∈ {(t, c), (−1, −1), ⊥}
2: T ← T[gid, (t, c), com, c'] // honest delivery
3: (γ, acc) ← Proc(ST[ID], T)
4: **if** ¬acc : **return** // failure
5: **reward** Props(ST[ID], T)
≠ T[gid, (t, c), vec, c']
6: **reward** $\neg(\emptyset \neq \gamma[\text{gid}].G^* \subseteq \gamma[\text{gid}].G)$
7: **if** $(t, c) = (-1, -1)$: // create msg
8: UniqueCreatePerGID(γ, gid, c')
9: **if** $ID \notin \gamma[\text{gid}].G$: // ID removed
10: ep[gid, ID] ← ⊥
11: **reward** γ[gid].k ≠ ⊥ : // key deleted
12: **else** : // ID in group
13: UpdateView(γ, gid, t, c')
14: **reward** $\gamma[\text{gid}].k \neq \mathsf{T}[\text{gid}, (t, c), \text{key}, c']$
15: ep[gid, ID] ← $(t + 1, c')$
16: ST[ID] ← γ

Oracle PROP(ID', gid, ID, type)

1: **require** type ∈ types
2: (γ, P) $\xleftarrow{\$}$ Prop(ST[ID'], gid, ID, type)
3: **if** $P = \bot$: **return** // failure
4: (gid*, type*, ID^*, ID'^*) ← Prop-Info(γ, P)
5: **reward** (gid*, type*, ID^*, ID'^*) ≠ (gid, type, ID, ID')
6: CheckSameGroupState(ST[ID'], γ, gid)
7: T[gid, ep[gid, ID'], prop, ++prop-ctr] ← P
8: ST[ID'] ← γ // upd. ST of proposer ID'

Oracle COMMIT(ID, gid, I, com-type)

// $I = (i_1, \ldots, i_k)$ for some k
1: **require** com-type ∈ com-types
2: **require** ep[gid, ID] ∉ {$(-1, -1)$, ⊥}
3: $\vec{P}$ ← $(T[\text{gid}, \text{ep}[\text{gid}, ID], \text{prop}, i])_{i \in I}$
4: (γ, T, k) $\xleftarrow{\$}$ Commit(ST[ID], gid, $\vec{P}$, com-type)
5: **if** $T = \bot$: **return** // failure
6: **reward** $ID \notin$ ST[ID][gid].G // no external comm.
7: CheckSameGroupState(ST[ID], γ, gid)
8: T[gid, ep[gid, ID], com, ++com-ctr] ← T
9: T[gid, ep[gid, ID], vec, com-ctr] ← Props(γ, T)
10: T[gid, ep[gid, ID], key, com-ctr] ← k
11: ST[ID] ← γ

UpdateView(γ, gid, t, c')

1: v ← ep-view[gid, $t + 1$, c']
2: **if** $v = \bot$: ep-view[gid, $t + 1$, c'] ← γ
3: **else** : CheckSameGroupState(v, γ, gid)

CheckSameGroupState(γ_1, γ_2, gid)

1: **reward** γ_1[gid].k ≠ γ_2[gid].k
2: **reward** $\gamma_1[\text{gid}].G \neq \gamma_2[\text{gid}].G$
3: **reward** $\gamma_1[\text{gid}].G^* \neq \gamma_2[\text{gid}].G^*$

UniqueCreatePerGID(γ, gid, c')

1: **if** first-crt[gid] ≠ ⊥ :
2: **require** c' = first-crt[gid]
3: **else** : first-crt[gid] ← c'

Figure 5.2: Correctness game for (A)-CGKA with respect to predicate $\mathsf{C}_{\mathsf{corr}}$ for $\mathcal{O}$ = {CREATE, DELIVER, PROP, COMMIT}. Highlighted code is executed only when considering an A-CGKA. Note that when **reward** P is true for predicate P, variable win is set to 1.

Overview. The game starts by setting up several public dictionaries. The main two are $\mathsf{ST}[\cdot]$, which is a dictionary indexed by *ID* which keeps the states of each of the users *ID* throughout the game; and $\mathsf{T}[\cdot]$, which keeps all control messages and proposals generated by the (A)-CGKA algorithms. A message T stored in T is indexed by the corresponding gid, epoch, message type (`prop`, `com` or `vec`, standing for proposal, commit or proposal vector, respectively), and a message counter prop-ctr or com-ctr; keys output by Commit for some gid and epoch are also stored in T and are marked with `key`. After initialisation, we let the adversary $\mathscr{A}$ interact with the oracles with respect to multiple groups. The variable win is set to 1 if one of the **reward** clauses is true, which leads to $\mathscr{A}$ winning given the (optional) predicate $\mathsf{C}_{\mathsf{corr}}$ is also true when $\mathscr{A}$ finishes executing.

Epochs. Control messages output by successful Create and Commit calls are labelled uniquely by the challenger. For correctness, an *epoch* is a pair (t,c), where t is an integer relative to a particular group and party which increments upon each successful Proc call while in the group, and c is the value of the global variable com-ctr when the corresponding control message was output. For a given group, each party's epoch value is stored in $\mathsf{ep}[\cdot]$ and initialised to $(-1,-1)$, and is set to $\perp$ when they leave a group. To this end, we model correctness in the presence of an adversary who maintains arbitrary network partitioning, so long as they provide a consistent view of messages to parties in each 'partition'.

Group Consistency. We enforce that, for each group member, that each group is only (possibly) updated upon a successful call to Proc (via CheckSameGroupState). For A-CGKA, we ensure that, for a group gid, $\emptyset \neq G^* \subseteq G$ must hold at all times. In Proc, we enforce that all users who transition to the same epoch have the same view of the group and set of admins when relevant (via UpdateView). To this end, the dictionary ep-view stores the state of the first party who transitions to a given epoch (t,c) for a gid, and the state of subsequent parties who transition to the same epoch are checked against the value stored in ep-view.

We also require that, even if there are multiple calls to Create, only one of them is processed by group members (i.e., states do not fork from the initial epoch). To achieve this, the variable first-crt tracks the commit number of the first successfully processed create message for a given gid. This check is made in UniqueCreatePerGID.

Key Partnering. Note that new epoch keys are derived upon successful Proc calls, and that a new key k (for group members) is always derived in this case. Correctness ensures that all users who transition to the same epoch derive the same key k. The consistency between the key output by Commit and the actual epoch keys is also verified. Moreover, whenever a user derives a key $k \neq \perp$, they must be a group member (line 11 of DELIVER).

Liveness. We enforce that some algorithms, such as Prop, always succeed on 'valid' input, since without such a check, an (A)-CGKA with algorithms that always fail is considered correct. One example of a liveness check is an equality check between proposals output by Props and the input proposals in COMMIT. Since the precise semantics of (A)-CGKA vary between applications, additional checks are delegated to a correctness predicate $\mathsf{C}_{\mathsf{corr}}$ which, in part,

parameterises the correctness game. In the previous example, $\mathsf{C}_{\mathsf{corr}}$ may depend on the protocol policy for the application of proposals. Without extra checks, the predicate should be set to $\mathsf{C}_{\mathsf{corr}} = \mathsf{true}$.

Definition 42 ($\mathsf{CORR}_{(\mathsf{A})\text{-}\mathsf{CGKA}}$)**.** Consider the correctness game CORR presented in Figure 5.2 A CGKA CGKA (resp. A-CGKA A-CGKA) is *correct* if, for all $\lambda \in \mathbb{N}$ and all computationally unbounded adversaries $\mathscr{A}$, we have $\Pr[\mathsf{CORR}_{(\mathsf{A})\text{-}\mathsf{CGKA},\mathsf{C}_{\mathsf{corr}}}(\mathscr{A}) \Rightarrow 1] = 0$.

Naturally, this notion can be relaxed to consider correctness that holds with high probability w.r.t. a computationally-bounded adversary (e.g., to capture decryption failures when using lattice-based cryptography as done in Chapter 3).

5.2.4 Security

A-CGKA is a primitive that extends the functionality of a CGKA, which allows a group to establish a sequence of group keys, to provide secure administration. Therefore, any A-CGKA construction must satisfy at least 'standard' CGKA security (key indistinguishability).

The main additional goal of A-CGKA over standard CGKA is to prevent unauthorised (standard) users from deciding on changes to a group, capturing the security of the group structure. Note that an A-CGKA in which the adversary fully controls a standard group member is not secure with respect to key indistinguishability, but should still prevent unauthorised group changes. We define (A)-CGKA security in Definition 43 after describing our notion below.

Overview. At its core, the security game in Figure 5.3 is a key indistinguishability game that captures the security of the common group secret for a single group (we implicitly assume a fixed gid), extending the game in [ACDT20]. The game considers a partially active adversary who can make forgery attempts and schedule messages but cannot totally control message delivery. Namely, the adversary can inject a control message to a specific party *ID*, but this message is not stored in the array T that keeps track of all honestly generated messages after Proc is called. The main consequence of this is that injected proposals cannot be included in commits generated by the challenger. Nevertheless, the adversary can make the challenger commit on vectors of proposals created via PROP.

Informally, the adversary can win the game if it plays a clean game where it either 1) correctly guesses the challenge bit by distinguishing between correct and uniformly sampled keys or (for A-CGKA) 2) manages to forge a message which, after being processed by a user, changes its view of (G, G^*). We assume that the dependency on the PKI is implicit in the game; we describe the PKI functionality we assume for each protocol as they are introduced. A detailed description follows.

Epochs. Messages output by successful Create, Commit, and Prop calls are uniquely labelled by the challenger via counters prop-ctr, com-ctr. Whenever such a call is made, the corresponding messages are stored in variable T with (incremented) last argument ++com-ctr.

Game $\mathsf{KIND}_{\mathsf{(A)\text{-}CGKA},\mathsf{C_{cgka}},\mathsf{C_{adm}},\mathsf{C_{forgery}}}(\mathscr{A})$

1: $b \xleftarrow{\$} \{0,1\}$
2: $\mathsf{K}[\cdot], \mathsf{ST}[\cdot] \leftarrow \bot$
3: **public** $\mathsf{T}[\cdot], \mathsf{G}[\cdot], \mathsf{ADM}[\cdot] \leftarrow \bot$
4: $\mathsf{prop\text{-}ctr}, \mathsf{com\text{-}ctr}, \mathsf{exp\text{-}ctr} \leftarrow 0$
5: $\mathsf{ep}[\cdot], \mathsf{exp}[\cdot] \leftarrow (-1,-1)$; $\mathsf{C}[\cdot] \leftarrow -1$
6: $\mathsf{chall}[\cdot], \mathsf{forged} \leftarrow \mathsf{false}$
7: $\mathsf{ST}[ID] \xleftarrow{\$} \mathsf{Init}(1^\lambda, ID)\ \forall\ ID$
8: $b' \xleftarrow{\$} \mathscr{A}^{\mathscr{O}}$
9: **if** $\neg \mathsf{C_{cgka}} \wedge \neg \mathsf{forged}$
10: $\quad b'' \xleftarrow{\$} \{0,1\}$; **return** b''
11: **else return** $\mathbb{1}[\![b = b' \vee \mathsf{forged}]\!]$

Oracle $\mathsf{CREATE}(ID, G, G^*)$

1: $(\gamma, T) \xleftarrow{\$} \mathsf{Create}(\mathsf{ST}[ID], G, G^*)$
2: **if** $T = \bot$ **return** // failure
3: $\mathsf{T}[(-1,-1), \mathtt{com}, \mathtt{++com\text{-}ctr}] \leftarrow (T, \mathsf{both})$
4: $\mathsf{ST}[ID] \leftarrow \gamma$

Oracle $\mathsf{PROP}(ID, ID', \mathsf{type})$

1: $(\gamma, P) \xleftarrow{\$} \mathsf{Prop}(\mathsf{ST}[ID], ID', \mathsf{type})$
2: $\mathsf{T}[\mathsf{ep}[ID], \mathtt{prop}, \mathtt{++prop\text{-}ctr}] \leftarrow P$
3: $\mathsf{ST}[ID] \leftarrow \gamma$

Oracle $\mathsf{COMMIT}(ID, (i_1, \ldots, i_k), \mathsf{com\text{-}type})$

1: $\vec{P} \leftarrow (\mathsf{T}[\mathsf{ep}[ID], \mathtt{prop}, i])_{i=(i_1,\ldots,i_k)}$
2: $(\gamma, T, k) \xleftarrow{\$} \mathsf{Commit}(\mathsf{ST}[ID], \vec{P}, \mathsf{com\text{-}type})$
3: **if** $T = \bot$ **return** // failure
4: $\mathsf{T}[\mathsf{ep}[ID], \mathtt{com}, \mathtt{++com\text{-}ctr}] \leftarrow (T, \mathsf{com\text{-}type})$
5: $\mathsf{T}[\mathsf{ep}[ID], \mathtt{vec}, \mathsf{com\text{-}ctr}] \leftarrow \mathsf{Props}(\mathsf{ST}[ID], T)$
6: $(t_s, t_a) \leftarrow \mathsf{ep}[ID]$
7: **if** $\mathsf{com\text{-}type} = \mathsf{adm}$ $t_s \leftarrow t_s - 1$
8: $\mathscr{K}[t_s + 1] \leftarrow k$; $\mathsf{ST}[ID] \leftarrow \gamma$

Oracle $\mathsf{EXPOSE}(ID)$

1: $\mathsf{exp}[ID, \mathtt{++exp\text{-}ctr}] \leftarrow \mathsf{ep}[ID]$
2: **return** $\mathsf{ST}[ID]$

Oracle $\mathsf{DELIVER}(ID, (t_s, t_a), c)$

1: **require** $\mathsf{ep}[ID] \in \{(t_s, t_a), (-1,-1)\}$
2: $(T, \mathsf{com\text{-}type}) \leftarrow \mathsf{T}[(t_s, t_a), \mathtt{com}, c]$ // honest deliv.
3: **if** $\mathsf{C}[(t_s, t_a)] \in \{c, -1\}$, $\mathsf{C}[(t_s, t_a)] \leftarrow c$
4: **else return** // bad commit for epoch
5: $(\gamma, \mathsf{acc}) \leftarrow \mathsf{Proc}(\mathsf{ST}[ID], T)$
6: **if** $\neg \mathsf{acc}$ **return** // failure
7: **if** $ID \notin \gamma.G$ // ID removed
8: $\quad \mathsf{ep}[ID] \leftarrow (-1,-1)$
9: **else** // ID in group, update dictionaries
10: $\quad \mathsf{ep}[ID] \leftarrow (t_s, t_a)$
11: $\quad$ **if** $\mathsf{com\text{-}type} \in \{\mathsf{std}, \mathsf{both}\}$
12: $\quad\quad \mathsf{K}[t_s + 1] \leftarrow \gamma.\mathsf{k}$
13: $\quad\quad \mathsf{G}[t_s + 1] \leftarrow \gamma.G$
14: $\quad\quad \mathsf{ep}[ID] \leftarrow \mathsf{ep}[ID] + (1,0)$
15: $\quad$ **if** $\mathsf{com\text{-}type} \in \{\mathsf{adm}, \mathsf{both}\}$
16: $\quad\quad \mathsf{ADM}[t_a + 1] \leftarrow \gamma.G^*$
17: $\quad\quad \mathsf{ep}[ID] \leftarrow \mathsf{ep}[ID] + (0,1)$
18: $\mathsf{ST}[ID] \leftarrow \gamma$

Oracle $\mathsf{CHAL}(t_s)$

1: **require** $(\mathsf{K}[t_s] \neq \bot) \wedge \neg \mathsf{chall}[t_s]$
2: $\mathsf{chall}[t_s] \leftarrow \mathsf{true}$
3: **if** $b = 0$ **return** $\mathsf{K}[t_s]$
4: **if** $b = 1$ **return** $r \xleftarrow{\$} \{0,1\}^\lambda$

Oracle $\mathsf{REVEAL}(t_s)$

1: **require** $(\mathsf{K}[t_s] \neq \bot) \wedge \neg \mathsf{chall}[t_s]$
2: $\mathsf{chall}[t_s] \leftarrow \mathsf{true}$
3: **return** $\mathsf{K}[t_s]$

Oracle $\mathsf{INJECT}(ID, m, t_a)$

1: **require** $\mathsf{C_{adm}} \wedge (\mathsf{ep}[ID] = (\cdot, t_a)) \wedge (t_a \neq -1)$
2: **require** $(m, \cdot) \notin \mathsf{T}$ // external forgery
3: $(\gamma, \bot) \leftarrow \mathsf{Proc}(\mathsf{ST}[ID], m)$
4: **if** $\mathsf{C_{forgery}}$
5: $\quad \mathsf{forged} \leftarrow \mathsf{true}$ // successful forgery
6: $\quad$ **return** b // adversary wins
7: **else return** $\bot$

Figure 5.3: Single-group key indistinguishability (KIND) security game for (A)-CGKA, parametrised by predicate $\mathsf{C_{cgka}}$ (resp. and predicates $\mathsf{C_{adm}}$ and $\mathsf{C_{forgery}}$ for A-CGKA) for $\mathscr{O} = \{\mathsf{CREATE}, \mathsf{PROP}, \mathsf{COMMIT}, \mathsf{CHAL}, \mathsf{DELIVER}, \mathsf{REVEAL}, \mathsf{EXPOSE}\}$ (resp. and INJECT). Highlighted code is executed only when considering an A-CGKA.

The evolution of the group after parties process such control messages is modelled using epochs (differing from the epochs considered for correctness), which are each represented as an integer t_s (for CGKA) or a pair of integers (t_s, t_a) (for A-CGKA). The *standard epoch* t_s represents the time between two successive key evolutions, where a different key should be derived in each t_s. The *administrative epoch* t_a represents the time between two changes in the group administration (i.e., between two sequences of simultaneous admin updates, adds and/or removals).

For CGKA, epochs (t_s) advance every time a commit is processed. For A-CGKA, t_s advances if the commit type $\mathsf{com\text{-}type} \in \{\mathsf{std}, \mathsf{both}\}$ and t_a advances if $\mathsf{com\text{-}type} \in \{\mathsf{adm}, \mathsf{both}\}$. Group members can be in different epochs, captured by the variable $\mathsf{ep}[ID]$ which stores the current standard/admin epoch pair for a given group member *ID*. If a participant *ID* is not in the group, then $\mathsf{ep}[ID] = -1$ (CGKA) or $\mathsf{ep}[ID] = (-1,-1)$ (A-CGKA) holds.

Challenges. At any point in the game, the adversary can challenge with respect to a standard epoch t_s by calling CHAL. In a challenge, the adversary is given the group key $\mathcal{K}[t_s]$ if the challenger's bit is $b = 0$, and a random string $r \xleftarrow{\$} \{0,1\}^\lambda$ if $b = 1$. The adversary must try to determine the value of b by outputting a guess b' of b. A given execution is considered valid when either the *standard cleanness predicate* $\mathsf{C}_{\mathsf{cgka}}$ is true or, for A-CGKA, the adversary makes a forgery and the *admin cleanness predicate* is true. Cleanness ensures that no trivial attacks on the game are possible; we elaborate on this below.

Exposure Mechanisms. In order to capture group key ratcheting (for FS and PCS), the adversary has two mechanisms to obtain secret group material: it can *expose* a user *ID* using EXPOSE and *reveal* the group secret k using REVEAL. An exposure leaks the entire current state of *ID* stored in $\mathsf{ST}[ID]$. We keep track of the specific epochs in which each *ID* was exposed using the $\mathsf{exp}[\cdot]$ variable. On the other hand, a reveal leaks the group key to the adversary on a specified epoch t_s, in which case $\mathsf{chall}[t_s]$ is set to true to prevent the adversary from challenging on t_s (conversely, the reveal fails when $\mathsf{chall}[t_s] = \mathsf{true}$).

Injections. For A-CGKA, the adversary can also win the game by successfully *injecting* a forged commit. An injection can be attempted by calling $\mathsf{INJECT}(ID, m, t_a)$, given that *ID* is in admin epoch t_a, where *ID* is the target group member and m is the forged message. Note we require $t_a \neq -1$ since the adversary could otherwise trivially invite a new user into a new group that it controls. Forgeries can only be attempted if the *administrative predicate* $\mathsf{C}_{\mathsf{adm}}$ is not violated. As discussed below, $\mathsf{C}_{\mathsf{adm}}$ captures administration security by excluding trivial attacks. A trivial scenario excluded by any predicate is when the adversaryadversary has exposed an administrator immediately before a forgery attempt. The adversary wins the game if the forgery is accepted by any group member $ID \in G$ and if the *forgery predicate* $\mathsf{C}_{\mathsf{forgery}}$ that we define below is true.

CGKA Cleanness Predicate. The security game in Figure 5.3 is parametrised by three cleanness predicates, $\mathsf{C}_{\mathsf{cgka}}$, $\mathsf{C}_{\mathsf{adm}}$ and $\mathsf{C}_{\mathsf{forgery}}$. The first predicate $\mathsf{C}_{\mathsf{cgka}}$ follows approaches like [BRV20, ACDT20] to parametrise the security of the common (A)-CGKA key. Namely, this pred-

icate excludes trivial attacks on indistinguishability, i.e., those that break security unavoidably such as exposing a user and issuing a challenge before its keying material has been updated. Further, it captures the exact security of the protocol (with respect to key indistinguishability), which in our case comprises forward security and post-compromise security after updates. If an independent CGKA is used to construct an A-CGKA, the predicate $\mathsf{C}_{\mathsf{cgka}}$ may mostly depend on the security of the CGKA. An example, as we show in later sections, is our second construction DGS.

A more fine-grained characterisation of this predicate is to write it as $\mathsf{C}_{\mathsf{cgka}} = \mathsf{C}_{\mathsf{cgka\text{-}opt}} \wedge \mathsf{C}_{\mathsf{cgka\text{-}add}}$, where $\mathsf{C}_{\mathsf{cgka\text{-}opt}}$ is an *optimal*, generic cleanness predicate that excludes only unavoidable trivial attacks, and $\mathsf{C}_{\mathsf{cgka\text{-}add}}$ is an additional cleanness predicate that depends on the scheme and excludes other attacks. We define $\mathsf{C}_{\mathsf{cgka\text{-}opt}}$ in a similar way to the safe predicate in [ACDT20]. Namely, we exclude the following cases: (1) the group secret in challenge epoch t_s^* was already challenged or revealed, and (2) a group member ID whose state was exposed in standard epoch $t_{\mathsf{exp}} \leq t_s^*$ did not update their keys (i.e., processed their own commit, or processed a commit in which they were the subject of an add, remove, or update proposal) or was not removed before the challenge epoch t_s^*. The optimal cleanness predicate is given in Figure 5.4 for an adversary that makes oracle queries $q_1, \ldots, q_n$ in the game.

$\boxed{\mathsf{C}_{\mathsf{cgka\text{-}opt}}:}\ \forall\ (i, ID, \mathsf{ctr} \in (0, \mathsf{exp\text{-}ctr}]) : q_i = \mathsf{CHAL}(t_i^*),$
$\big(ID \notin \mathsf{G}[t_i^*]\big) \vee$
$\big(\exists (t_i, c) : (\mathsf{tExp}(ID, \mathsf{ctr}).t_s < t_i \leq t_i^*) \wedge \mathsf{hasUpd}_{\mathsf{std}}(ID, \mathsf{T}[(t_i, \cdot), \mathtt{com}, c], \mathsf{T}[(t_i, \cdot), \mathtt{vec}, c]) \wedge (\mathsf{C}[(t_i, \cdot)] = c)\big)$
$\vee \big(t_i^* < \mathsf{tExp}(ID, \mathsf{ctr}).t_s\big).$

Figure 5.4: Optimal CGKA predicate where the adversary makes oracle queries $q_1, \ldots, q_n$.

The predicate is the logical disjunction of three clauses: for every exposure, adversarial challenge, and party ID, we require that either (1) ID was not a group member at the challenge time; (2) the challenge epoch precedes the exposure (forward security); or (3) ID updated between the exposure and the (subsequent) challenge (post-compromise security). To avoid cluttering the predicate, our game already enforces that only one challenge or reveal can be performed per epoch (which is optimal for our game).

We have used the following auxiliary functions. The function tExp is such that $\mathsf{tExp}(ID, \mathsf{ctr}) = \mathsf{exp}[ID, \mathsf{ctr}]$ if $\exists k : q_k = \mathsf{EXPOSE}(ID)$, and -1 (for CGKA) or $(-1, -1)$ (for A-CGKA) otherwise. Given $\vec{P}$, the function $\mathsf{hasUpd}_{\mathsf{std}}(ID, (T, \mathsf{com\text{-}type}), \vec{P})$ (sans com-type for CGKA) outputs true if either: (i) ID has processed a commit of his own, where $\mathsf{com\text{-}type} \in \{\mathsf{std}, \mathsf{both}\}$, or (ii) ID is a user affected by an add, update, or removal proposal in $\vec{P}$ that had an effect.

Admin Cleanness Predicate. The second predicate $\mathsf{C}_{\mathsf{adm}}$ models administration security. Intuitively, this predicate should be more permissive in some aspects than $\mathsf{C}_{\mathsf{cgka}}$, since a forgery attempt should be permitted even if the adversary knows the state of a (standard) group member. Following the approach above, we can decompose $\mathsf{C}_{\mathsf{adm}}$ as $\mathsf{C}_{\mathsf{adm}} = \mathsf{C}_{\mathsf{adm\text{-}opt}} \wedge \mathsf{C}_{\mathsf{adm\text{-}add}}$.

$\mathsf{C}_{\text{adm-opt}}$ is symmetric to $\mathsf{C}_{\text{cgka-opt}}$ and excludes the following family of attacks: the adversary attempts a forgery on a member ID' at an administrative epoch t_a^* while having exposed the state of an administrator $ID \in G^*$ at an administrative epoch $t_{\mathsf{exp}} \leq t_a^*$, such that ID has not updated at some point between them. The predicate is optimal, as any attack that it excludes must occur while an administrator is directly under state exposure. In the game itself, we also require that ID' is in the challenge epoch specified by the adversary, i.e., $\mathsf{ep}[ID'] = (\cdot, t_a^*)$. Notice that this predicate is unrelated to the common group secret and standard epochs t_s, and only relates to administration dynamics.

$\boxed{\mathsf{C}_{\text{adm-opt}}:}\ \forall\ (i, ID, ID', \mathsf{ctr} \in (0, \text{exp-ctr}]) : q_i = \mathsf{INJECT}(ID', \cdot, t_i^*),$
$(ID \notin \mathsf{ADM}[t_i^*]) \vee$
$\big(\exists (t_i, c) : (\mathsf{tExp}(ID, \mathsf{ctr}).t_a < t_i \leq t_i^*) \wedge \mathsf{hasUpd}_{\mathsf{adm}}(ID, \mathsf{T}[(\cdot, t_i), \mathtt{com}, c], \mathsf{T}[(\cdot, t_i), \mathtt{vec}, c]) \wedge \mathsf{C}[(\cdot, t_i)] = c\big)$
$\vee\ \big(t_i^* < \mathsf{tExp}(ID, \mathsf{ctr}).t_a\big).$

Figure 5.5: Optimal administrative predicate where the adversary makes oracle queries $q_1, \ldots, q_n$.

The optimal administrative predicate $\mathsf{C}_{\text{adm-opt}}$ is captured in Figure 5.5. In the expression, the function $\mathsf{hasUpd}_{\mathsf{adm}}$ is defined as in $\mathsf{hasUpd}_{\mathsf{std}}$, except it is defined with respect to com-type $\in$ {adm, both} (rather than com-type $\in$ {std, both}).

A-CGKA Forgery Predicate. For A-CGKA, we define security under active attacks performed using the INJECT oracle with respect to a predicate $\mathsf{C}_{\mathsf{forgery}}$. The predicate we describe captures the fact that if admins have not been compromised, then non-admins can only make group changes for self-removes, i.e., when non-admins want to remove themselves. Moreover, we require that self-removes cannot be forged themselves by parties that are not corrupted. If there are no self-remove operations, then the predicate reduces to the fact that non-admins cannot cause changes in the group.

The predicate $\mathsf{C}_{\mathsf{forgery}}$ is defined as follows with respect to variables in INJECT and the KIND game in general. Suppose m is input to INJECT. Let $\vec{P} = \mathsf{Props}(\mathsf{ST}[ID], m)$. Consider $\vec{P}' = \{P \in \vec{P} : P' \in \mathsf{T}[\mathsf{ep}[ID], \mathtt{prop}, \cdot] \wedge \mathsf{Prop\text{-}Info}(\mathsf{ST}[ID], P) = \mathsf{Prop\text{-}Info}(\mathsf{ST}[ID], P')\}$.[3] Let $H = \{ID : (\mathsf{gid}, \mathsf{rem}, ID, ID) = \mathsf{Prop\text{-}Info}(\mathsf{ST}[ID], P) \wedge (P \in \vec{P}')\}$ and $H^* = \{ID : (\mathsf{gid}, \text{rem-adm}, ID, ID) = \mathsf{Prop\text{-}Info}(\mathsf{ST}[ID], P) \wedge (P \in \vec{P}')\}$. Then $\mathsf{C}_{\mathsf{forgery}}$ is true if and only if $(\mathsf{ST}[ID].G \setminus H, \mathsf{ST}[ID].G^* \setminus H^*) \neq (\gamma.G, \gamma.G^*)$. If there are no self-removes, i.e., $H = H^* = \emptyset$, this simplifies to the predicate $(\mathsf{ST}[ID].G, \mathsf{ST}[ID].G^*) \neq (\gamma.G, \gamma.G^*)$; let $\mathsf{C}_{\mathsf{forgery}}{}^*$ be this simplified predicate.

Definition 43 (KIND)**.** A CGKA CGKA (resp. A-CGKA A-CGKA) is KIND w.r.t. predicate $\mathsf{C}_{\mathsf{cgka}}$ (resp. and $\mathsf{C}_{\mathsf{adm}}, \mathsf{C}_{\mathsf{forgery}}$) if, for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\text{kind}}_{\text{(A)-CGKA}}(\mathscr{A}) := \left| \Pr[\mathsf{KIND}_{\text{(A)-CGKA}, \mathsf{C}_{\mathsf{cgka}}, (\mathsf{C}_{\mathsf{adm}}, \mathsf{C}_{\mathsf{forgery}})}(\mathscr{A}) \Rightarrow 1] - \frac{1}{2} \right| = \mathsf{negl}\,,$$

[3] The effect of the equality check with respect to Prop-Info is that a dishonest proposal P' that has the same semantics as an honest proposal P will not be considered a 'forgery' by $\mathsf{C}_{\mathsf{forgery}}$.

where game KIND is defined in Figure 5.3.

Limitations. Our security definition, which is based on Alwen et al.'s unauthenticated CGKA notion [ACDT20], does not allow arbitrary message injections to participants. Thus, attacks on robustness are not captured by our security model. In particular, so long as non-admins are allowed to make commits, our A-CGKA schemes will only provide as much security as the underlying core CGKA: using MLS's TreeKEM, for example, a malicious non-admin can deny service by sending a malformed commit message that can be successfully processed only by some of the users. This can be fixed at the expense of using NIZKs within TreeKEM [ACJM20, DDF21]. In any case, we note that confidentiality is not compromised under this family of attacks, as their main consequence is to "disconnect" users from the protocol (in particular, new users cannot be added).

If only admins are allowed to commit, then our schemes (to be introduced) are safe against this attack vector for some non-strongly robust variants of TreeKEM, such as the one used in MLS [BBR+23]. Standard users can still attain FS and PCS guarantees, and in particular PCS when their update proposals are committed.

Among the broader family of group key agreement protocols, where long-lived sessions and PCS are not always considered, modelling fully active adversaries is common [PRSS21]. We also do not explicitly model authentication (we implicitly assume an incorruptible PKI) and randomness manipulation, and we do not explicitly model parties who do not delete their state as instructed by the algorithm and are then exposed (via a *no-deletion* oracle or similar [ACDT21a]). We leave these for future work.

Multi-group security can be captured rather easily. The main difference (besides increased notation complexity introduced by the gids as in Figure 5.2) appears in the state exposure oracle: exposing the state of a party implies a security loss in all groups that the party is a member of simultaneously. The feature is not included as our security proofs are in the single-group setting.

Our INJECT oracle does not allow the adversary to inject group creation messages. Of course, the adversary can always make a new group with whatever users it chooses. It is nonetheless possible to extend our security notion to allow for injections, such that the adversary can only create a group for 'valid' users, i.e., those who have registered their keys with the PKI (which would be checked when a user processes a welcome message).

5.3 A-CGKA Constructions

In this section, we first present two different constructions for A-CGKA: *individual admin signatures* (IAS) and *dynamic group signature* (DGS). In the first construction, we construct A-CGKA on top of a CGKA by adding additional administration mechanisms based on authentication via signatures. In the second construction, we use two independent but synchronised

CGKAs. We selected these two approaches due to their simplicity, efficiency and flexibility to adapt to different underlying CGKAs.

A first attempt of A-CGKA is to simply require group members to keep a list of administrators over time. Whenever an admin wants to make a commit, it can simply check whether the admin-changing proposals have been made by administrators, then commit them, and the other users will verify the admin condition upon processing. This approach is functional, but not secure in our model due to a lack of admin authentication. An adversary can easily forge a commit message and impersonate an admin unless this message is authenticated (for example signed). Many notions of CGKA security [ACDT20, KPPW+21] do not necessarily imply such a level of authentication.

One partial fix is to require admins to sign using a key derived from a long-term identity key. Then, security cannot be recovered if the admin is compromised once, resulting in the adversary winning the A-CGKA game too. Our constructions provide FS and PCS for the relevant admin authentication mechanisms in order to circumvent this problem.

In the following two subsections, all implemented A-CGKA algorithms, including Init, are stateful as if executed by the same party and, as written, *do not explicitly return the updated local state*. Instead, they modify the state during runtime. In the event of algorithm failure, the state is not modified and appropriate failure values are output.

5.3.1 Individual Admin Signatures

In our first construction, *individual admin signatures* (IAS), we build a generic and modular administration mechanism on top of an arbitrary CGKA protocol (denoted by CGKA). Each group administrator $ID \in G^*$ maintains their own signature key pair $(\mathsf{ssk}, \mathsf{spk})$ that is updated over time. Each key pair is independent from the keys used in CGKA, which is mostly used as a black-box. Group members keep track of the list of admins G^* which is (possibly) updated upon processing each control message. Proposed changes to the group and to the administration are signed using an admin's keying material.

The IAS construction is presented in Figures 5.6, 5.7 and 5.8. The first figure describes the A-CGKA algorithms, and the second and third describe helper functions and auxiliary methods. We note that the algorithms defined in Figure 5.6 are incomplete without the helper functions; therefore, the construction spans all three figures.

States. We represent the state of a participant by the symbol γ, which is in part a dictionary of states indexed by group identifiers, i.e., $\gamma[\mathsf{gid}]$. Users further maintain a common state via $\gamma.\mathsf{s0}$ encoding the underlying CGKA state, security parameter 1^λ in $\gamma.1^\lambda$ and the user's ID in $\gamma.ME$. For each group gid, users keep a separate variable $\gamma[\mathsf{gid}].\mathsf{adminList}$ that encodes the group administrators and two administration-related signature key pairs. The state also tracks the group members as $\gamma[\mathsf{gid}].G = \gamma[\mathsf{gid}].\mathsf{s0}.G$ and the CGKA key as $\gamma[\mathsf{gid}].\mathsf{k} = \gamma[\mathsf{gid}].\mathsf{s0}.\mathsf{k}$ from the underlying CGKA, as well as the admins as $\gamma[\mathsf{gid}].G^* = \{ID : \gamma[\mathsf{gid}].\mathsf{adminList}[ID] \neq \bot\}$.

Description

In our functions in Figures 5.6, 5.7 and 5.8, we often omit the group identifier of the state to simplify presentation. We assume that γ refers to $\gamma[\mathsf{gid}]$ whenever gid is a subject of the algorithm, such as when it is a parameter of the function, and sometimes omit gid when it is clear from context. We note that our scheme nevertheless supports multiple groups.

Randomness. In our construction, we make randomness used by protocol algorithms explicit, including sampled randomness $r_0 \in \{0,1\}^\lambda$ as input. Namely, for the input randomness r_0 used in any randomised method, we apply a PRF $(r_1,\ldots,r_k) \leftarrow H_k(r_0,\gamma)$ that combines the entropy of r_0 and the state γ. We do this to reduce the impact of randomness leakage and manipulation attacks [BRV20]: without prior knowledge of γ (and assuming it has a sufficient entropy), an adversary that reads or manipulates r_0 will not be able to derive a corresponding r_i value. This is an additional feature that aims to maintain certain security properties in stronger adversarial models than considered in this chapter, and does not interfere with the rest of the protocol.

PKI. IAS assumes a basic, incorruptible PKI functionality where all parties are authenticated with the PKI. The PKI provides a fresh signature public key spk for which only the party *ID* can retrieve the corresponding secret key ssk. This functionality is used in for two main reasons:

1. When the group of administrators expands; namely, when a party ID' crafts a group gid or makes an admin add proposal[4]; and
2. When a non-admin user wishes to remove themselves from gid (a 'self-remove').

For these purposes, we define a getSpk algorithm, which on input (ID, ID', gid) for subject *ID* and caller ID' outputs spk relevant to the context the call is made in. We also assume a method of the form $\mathsf{getSsk}(\mathsf{spk}, ID, \mathsf{gid})$ that returns the ssk associated to spk when called by *ID* given they uploaded it. During protocol execution, parties upload signature key pairs $(\mathsf{ssk}, \mathsf{spk})$ to the PKI via an abstract $\mathsf{registerKeys}(ID)$ method in initialisation and via $\mathsf{registerKey}(ID)$ during the two aforementioned scenarios.[5] Formally, the adversary is only exposed to getSpk and $\mathsf{registerKey}$; we assume the other functions are called as needed in the security game though.

Initialisation. Before the creation of a group, a participant starts by calling the init method, which initialises the state γ. In turn, Init calls CGKA.Init from the underlying CGKA to initialise its state $\gamma.\mathsf{s0}$. $(\mathsf{ssk}, \mathsf{spk})$ and $(\mathsf{ssk}', \mathsf{spk}')$ are two signature key pairs for group administration. The first pair is the valid admin signature key pair using during protocol execution, while the second pair stores updated keys after a commit or a key update operation is performed by the participant but before it is processed (i.e., acts as a temporary variable). After successfully processing a commit message, the second key pair replaces the first.

[4] Since admin proposals are all signed, and at proposal time it is not clear if a commit will contain an admin add proposal, admins always upload their updated signature key to the PKI at proposal time. For simplicity, signature keys are also always uploaded to the PKI when sampled at commit time also.

[5] This abstraction is made to reduce notational complexity.

IAS.Init($1^\lambda, ID; r_0$)

$(r_1, r_2) \leftarrow H_2(r_0, \gamma)$; γ.s0 $\leftarrow$ CGKA.Init($1^\lambda, ID; r_1$)
γ.ME $\leftarrow ID$; $\gamma.1^\lambda \leftarrow 1^\lambda$
$\gamma[\cdot]$.adminList$[\cdot] \leftarrow \perp$ // maps ID to spk
$\gamma[\cdot]$.ssk, $\gamma[\cdot]$.spk $\leftarrow \perp$ // active admin key pair
$\gamma[\cdot]$.ssk$'$, $\gamma[\cdot]$.spk$' \leftarrow \perp$ // temporary key pair
registerKeys(ID) // gen/upload keys to PKI

IAS.Prop(gid, ID, type; r_0)

$P \leftarrow \perp$; $(r_1, r_2, r_3, r_4) \leftarrow H_4(r_0, \gamma)$
if type = $*$-adm
  // Note if type = upd-adm, keys are updated
  require γ.ME $\in \gamma.G^*$
  $P \leftarrow$ makeAdminProp(gid, type, $ID; r_1, r_2$)
else $(\gamma$.s0$, P) \leftarrow$ CGKA.Prop(γ.s0, gid, ID, type; r_1)
if (type = rem) $\wedge$ ($ID = ME$) $\wedge$ ($ID \notin \gamma.G^*$)
  (ssk, spk) $\leftarrow$ KeyGen($\gamma.1^\lambda; r_3$); registerKey(ID)
  $P \leftarrow (P$, Sign(ssk$', P; r_4$))
return P

IAS.Commit(gid, $\vec{P}$, com-type; r_0)

require γ.ME $\in \gamma$.s0.$G \wedge$ com-type $\in$ {adm, std, both}
$(r_1, \ldots, r_4) \leftarrow H_4(r_0, \gamma)$
$(\vec{P}_0, \vec{P}_A, \Sigma,$ admReq$) \leftarrow$ propCleaner(gid, $\vec{P}$)
require verifyPropSigs($\vec{P}_0, \Sigma, \vec{P}_A$)
if admReq $\vee$ (com-type $\in$ {adm, both})
  require γ.ME $\in \gamma.G^*$
  if com-type $\in$ {adm, both} : $C_A \leftarrow \vec{P}_A$
  if com-type $\in$ {std, both}
    $(C_0, W_0,$ adminList$, k) \leftarrow$ c-Std(gid, $\vec{P}_0, \vec{P}_A; r_1$)
    require $C_0 \neq \perp$
  // Generate new key pair and sign new spk
  $(\gamma$.ssk$', \gamma$.spk$') \leftarrow$ KeyGen($\gamma.1^\lambda; r_2$); registerKey(ID)
  $T_C \leftarrow$ ('comm', γ.ME, $C_0, C_A, \perp, \gamma$.spk$')$
  if $W_0 \neq \perp$ // share updated admin list
    $T_W \leftarrow$ ('wel', γ.ME, W_0, adminList)
  else $T_W \leftarrow \perp$
  $\sigma_T \leftarrow$ Sign(γ.ssk, (gid, $T_C, T_W); r_4$)
else // only self-removes - no admin sig
  $(C_0, \perp, \perp, k) \leftarrow$ c-Std(gid, $\vec{P}_0, \perp; r_3$)
  $T_C \leftarrow$ ('comm', γ.ME, $C_0, \perp, \Sigma, \perp$)
  $T_W \leftarrow \perp$; $\sigma_T \leftarrow \perp$
return $((\text{gid}, T_C, T_W, \sigma_T), k)$

IAS.Create(gid, $G, G^*; r_0$)

require $(\gamma$.ME $\in G^*) \wedge (G^* \subseteq G)$
$(r_1, r_2) \leftarrow H_2(r_0, \gamma)$
$(\gamma$.s0$, W_0) \leftarrow$ CGKA.Create(γ.s0, gid, $G; r_1$)
if $W_0 = \perp$ **return** $\perp$
adminList$[\cdot] \leftarrow \perp$ // this is not γ.adminList
for $ID \in G^*$:
  adminList$[ID] \leftarrow$ getSpk(ID, γ.ME)
γ.spk$' \leftarrow$ adminList[ME]
γ.ssk$' \leftarrow$ getSsk(γ.spk$'$, ME)
$T_W \leftarrow$ ('wel', γ.ME, W_0, adminList)
return (gid, $\perp, T_W$, Sign(γ.ssk$'$, (gid, $\perp, T_W); r_2$))

IAS.Proc(T)

(gid, $T_C, T_W, \sigma_T) \leftarrow T$; acc $\leftarrow$ false
if (γ.ME $\notin \gamma$[gid].s0.G) $\wedge$ ($T_W \neq \perp$)
  require T_W.msg-type = 'wel'
  acc $\leftarrow$ p-Wel(gid, T_W, σ_T) // welcome helper
else if (γ.ME $\in \gamma$[gid].s0.G) $\wedge$ ($T_C \neq \perp$)
  (msg-type, $ID, C_0, \cdot, \Sigma, \cdot) \leftarrow T_C$
  require msg-type = 'comm'
  for $\sigma : (P, ID', \sigma) \in \Sigma$:
    if $\neg$Vrfy(getSpk(ID', ME), σ, P)
      $\vee$ ($ID' \in \gamma$[gid].G^*) **return** false
  if $\sigma_C = \perp$ // no sign - check only self-removes
    $(\gamma'$, acc$) \leftarrow$ CGKA.Proc(γ[gid].s0$, C_0$)
    $SR \leftarrow \{ID' : (\cdot, ID') \in \Sigma\}$
    if $\neg$acc $\vee \gamma'$[gid].s0.$G \cup SR \neq \gamma$[gid].s0.G
      return false
    γ[gid].s0 $\leftarrow \gamma'$; **return** true
  if $\neg[(ID \in \gamma$[gid].$G^*) \wedge$
  (Vrfy(γ[gid].adminList$[ID]$, (gid, T_C, T_W), σ_T))]
    return false // verification failed
  acc $\leftarrow$ p-Comm(gid, T_C) // admin commit helper
return acc

IAS.Prop-Info(P)

if P is of the form $(P, \sigma) : (P, \sigma) \leftarrow P$ // self-removes
if P is a CGKA proposal
  $(P$.gid, P.type, P.ID, P.ID$') \leftarrow$ CGKA.Prop-Info(γ.s0$, P$)
else if P is an admin proposal
  $(P$.gid, P.type, P.ID, P.ID$', \perp, \perp) \leftarrow P$
return (P.gid, P.type, P.ID, P.ID$')$

Figure 5.6: Individual admin signatures (IAS) A-CGKA built from a CGKA CGKA, a signature scheme Sig, and PRFs $H_n : \mathscr{R} \times \text{ST} \to \mathscr{R}^n$ for $n \leq 4$, randomness space $\mathscr{R}$ and state space ST. We let γ[gid].$G = \gamma$[gid].s0.G, γ[gid].$G^* = \{ID : \gamma$[gid].adminList$[ID] \neq \perp\}$, and γ[gid].k = γ[gid].s0.k.

valid(P)

// Predicate checks validity of admin proposal
(P.gid, P.type, P.ID, P.ID′) ← Prop-Info(P)
S_1 := (P.gid = gid) // correct group
S_2 := ($P.ID \in \gamma$[gid].G) // ID member
S_3 := ($P.ID' \in \gamma$[gid].G^*) // ID' admin
C_1 := (P.type = rem-adm)
S_4 := ($P.ID \in \gamma$[gid].G^*) // ID admin
C_2 := (P.type = add-adm)
return $S_1 \wedge S_2 \wedge S_3 \wedge (\neg C_1 \vee S_4) \wedge \neg(C_2 \wedge S_4)$

makeAdminProp(gid, type, $ID; r_1, r_2$)

$P_0 \leftarrow \bot$
if type = add-adm
  $\text{spk}_{\text{pki}} \leftarrow$ getSpk(ID, γ.ME)
  $P_0 \leftarrow$ (gid, type, ID, γ.ME, spk_{pki})
else if type = rem-adm
  $P_0 \leftarrow$ (gid, type, ID, γ.ME, $\bot$)
else if type = upd-adm
  if $(\gamma.\text{ssk}', \gamma.\text{spk}') \neq (\bot, \bot)$
    return $\bot$ // only one update per epoch
  $(\gamma.\text{ssk}', \gamma.\text{spk}') \leftarrow \text{KeyGen}(\gamma.1^\lambda; r_1)$; registerKey($ID$)
  $P_0 \leftarrow$ (gid, type, γ.ME, γ.ME, $\gamma.\text{spk}'$)
else return $\bot$
return $(P_0, \text{Sign}(\gamma.\text{ssk}, P_0; r_2))$

c-Std(gid, $\vec{P}_0, \vec{P}_A; r_1$)

$(\gamma.\text{s0}, (C_0, W_0), k) \leftarrow \text{CGKA.Commit}(\gamma.\text{s0}, \text{gid}, \vec{P}_0; r_1)$
if $W_0 \neq \bot$ // list for new users only
  adminList′ ← updAL(γ.adminList, $\vec{P}_A$)
  return $(C_0, W_0, \text{adminList}', k)$
else return $(C_0, \bot, \bot, k)$

verifyPropSigs($\vec{P}_0, \Sigma, \vec{P}_A$)

for $(P, ID, \sigma) \in \Sigma$ // non-admin self-removes
  spk ← getSpk($ID, \gamma.ME$)
  if $\neg\text{Vrfy}(\text{spk}, P, \sigma) \vee P \notin \vec{P}_0 \vee ID \in \gamma.G^*$
    return false
for $(P, \sigma_P) \in \vec{P}_A$: // admin props
  $(\bot, \bot, \bot, ID') \leftarrow$ Prop-Info(P)
  $\text{spk}_P \leftarrow \gamma$.adminList[$ID'$]
  if $\neg(\text{Vrfy}(\text{spk}_P, P, \sigma_P) \wedge \text{valid}(P))$
    return false
return true

propCleaner(gid, $\vec{P}$)

admReq ← false; $\vec{P}_0, \vec{P}_A, \Sigma \leftarrow []$
for $P \in \vec{P}$:
  (gid, P.type, P.ID, P.ID′) ← Prop-Info(P)
  if (P.type = $*$-adm) $\wedge$ valid(P)
    $\vec{P}_A \leftarrow [\vec{P}_A, P]$
    admReq ← true
  else // $\vec{P}_0$ is handled by CGKA
    if P.type = rem $\wedge$
    $(P.ID = P.ID') \wedge (P.ID \notin \gamma$[gid].$G^*)$
      $(P', \sigma) \leftarrow P$
      $\vec{P}_0 \leftarrow [\vec{P}_0, P']$; $\Sigma \leftarrow [\Sigma, (P, P.ID, \sigma)]$
    else if P.type $\in$ {add, rem}
      admReq ← true
    else $\vec{P}_0 \leftarrow [\vec{P}_0, P]$
    // Admin rem from $G \implies$ rem from G^*
    if $(P.\text{type} = \text{rem}) \wedge (P.ID \in \gamma$[gid].$G^*)$
      $P' \leftarrow$ makeAdminProp(gid, rem, $ID; \bot$)
      $\vec{P}_A \leftarrow [\vec{P}_A, P']$
$(\vec{P}_0, \vec{P}_A) \leftarrow$ enforcePolicy($\vec{P}_0, \vec{P}_A$)
return $(\vec{P}_0, \vec{P}_A, \Sigma,$ admReq)

p-Wel(gid, T_W, σ)

$(\bot, ID, W_0, \text{adminList}) \leftarrow T_W$
$(\gamma$[gid].s0, acc) ← CGKA.Proc(γ.s0, W_0)
acc ← acc $\wedge$ Vrfy(getSpk($\gamma.ME, ID$), (gid, $\bot, T_W$), σ)
if acc
  γ[gid].adminList ← adminList
  if adminList[ME] $\neq \bot$
    γ.spk ← adminList[ME]
    γ.ssk ← getSsk(spk, ME)
return acc

updAL(adminList, $\vec{P}_A$)

for $P \in \vec{P}_A$
  (gid, type, $ID, \bot$, spk, $\bot$) ← P
  if type $\in$ {add-adm, upd-adm}
    if (type = add-adm) $\wedge$ $(ID = \gamma.ME)$
      $\gamma.\text{spk}, \gamma.\text{ssk} \leftarrow (\text{spk}, \text{getSsk}(\text{spk}, ID))$
    adminList[ID] ← spk
  if type = rem-adm
    adminList[ID] ← $\bot$
    if $(ID = \gamma.ME)$
      $(\gamma.\text{ssk}, \gamma.\text{spk}) \leftarrow (\bot, \bot)$
return adminList

Figure 5.7: Helper functions for IAS in Figure 5.6 (part I).

IAS.Props(T)

// Supports non-welcome control messages
1 : $(T_C, T_W, \sigma_T) \leftarrow T$
2 : $\vec{P}_0 \leftarrow$ CGKA.Props($T_C.C_0$); $\vec{P}_A \leftarrow T_C.C_A$
3 : **return** $\vec{P}_0 || \vec{P}_A$

enforcePolicy($\vec{P}_0, \vec{P}_A$)

// This method can be extended to other policies
1 : numAdmins $\leftarrow |G^*|$
2 : **for** $P \in [\vec{P}_0, \vec{P}_A]$
3 : (gid, P.type, P.ID, P.ID′) ← Prop-Info(P)
4 : **if** type = rem // if duplicates, removal prevails
5 : Delete any other P' s.t. $P.ID = P'.ID$ except for rem-adm proposals
6 : **else if** type = rem-adm
7 : Delete any other admin P' s.t. $P.ID = P'.ID$
8 : numAdmins−
9 : **else if** type = add-adm, numAdmins++
10 : **require** numAdmins ≥ 1 // ensures $\emptyset \neq G^* \subseteq G$
11 : **return** $(\vec{P}_0, \vec{P}_A)$

p-Comm(gid, T_C)

1 : $(\perp, ID, C_0, C_A, \Sigma, \text{spk}) \leftarrow T_C$
// Check signatures in proposals
2 : **if** $C_A \neq \perp$ // $C_A = \vec{P}_A$
3 : **if** ¬verifyPropSigs($\perp, \perp, C_A$) **return** false
// Apply commit
4 : **if** $C_0 \neq \perp$
5 : $(\gamma', \text{acc}) \leftarrow$ CGKA.Proc(γ.s0, C_0)
6 : **if** acc = false **return** false
7 : **if** γ.ME $\notin \gamma'.G$ // user removed
8 : γ[gid] $\leftarrow \perp$ // reinitialise state (only for gid)
9 : **else** γ[gid].s0 $\leftarrow \gamma'$
// Set temporary updated keys
10 : **if** $(ID = \gamma.\text{ME}) \vee (\exists P \in C_A : P.ID = \gamma.\text{ME})$
11 : $(\gamma.\text{ssk}, \gamma.\text{spk}) \leftarrow (\gamma.\text{ssk}', \gamma.\text{spk}')$
12 : $\gamma.\text{ssk}', \gamma.\text{spk}' \leftarrow \perp$
13 : γ.adminList ← updAL(γ.adminList, C_A)
14 : γ.adminList[ID] ← spk // committer's key
15 : **return** true

Figure 5.8: Helper functions for IAS in Figure 5.6 (part II).

Group Creation. The Create algorithm creates the group gid from the list of members G, the admin list from G^*, and outputs a (signed) control message T for the new members in G. The adminList variable maps identifiers $ID \in G^*$ to signature public keys spk_{ID}. The public signature keys are obtained via getSpk and each admin's private key can be retrieved from the PKI via getSsk while they are processing T, the control message that adds them to the group. The group creator directly stores such key pair as $(\gamma.\text{ssk}', \gamma.\text{spk}')$.

Proposals. Any group member can use Prop to create a proposal of a non-admin type; the algorithm calls CGKA.Prop in this case. Administrative proposals are restricted to admins and crafted by makeAdminProp, which includes an administrative signature in the proposal. The signature is included to prevent an (insider) adversary from forging the sender of the proposal in an attempt to impersonate an admin.[6] Proposal creation does not have any effect on the state other than the storage of temporary keys for proposals with type type = upd-adm. In the case of an add-adm proposal to promote ID to admin status, the proposer γ.ME retrieves a public signature key spk of ID from the PKI using getSpk. In the case of a rem proposal where $ID = ME$ and $ID \notin \gamma$[gid].G^* (i.e., a non-admin self-remove), the caller samples a new signature key pair, registers the public key spk with the PKI and signs their proposal.

The Prop-Info method simply retrieves the main information of a proposal. As mentioned in

[6] Note in MLS that all proposals are in any case signed.

the previous section, it could be adapted to support CGKAs and A-CGKAs where proposals are encrypted[7] (under some key derived from the group key, for instance).

Commits. The Commit algorithm can only be called by group administrators (except for the special case in which only key updates and self-removes are proposed, in which cases standard users can commit), and performs the following actions:

1. Clean the input vector of proposals $\vec{P}$, ensuring that they are well-formed. This is done via the propCleaner algorithm, which in turn calls the enforcePolicy method. For security reasons, we adopt the main features of the MLS policy (removing duplicates and prioritising removals) in our construction [BBR+23], but extensions to this policy can be implemented. In addition, we verify the legitimacy of the admin proposals and the fact that self-remove proposals are correctly signed via verifyPropSigs. Then, the predicate valid(P) verifies that the gid matches, that added users (respectively admins) do not belong to G (resp. G^*), that removed users do belong to G (resp. G^*), and that the proposer is an admin. Finally, we ensure that all users removed from G are also removed from the adminList.

2. Carry out the administrative and the standard commits and produce an administrative commit message C_A (which is the clean admin proposal vector), a standard CGKA commit C_0, and an updated adminList. We split the CGKA commit in two components C_0 and W_0 as is usual in the literature [ACDT20, AJM22, KPPW+21, ACJM20]. If the CGKA does not allow for this, it is easy to modify the protocol without compromising security[8].

3. Generate a new administrative signature key pair $(\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}')$ stored as the temporary key pair.

4. Produce the final control message T which includes the new spk'. The message T is again split into two components: A first component T_W (for welcome) includes all the required information for incoming A-CGKA members, including the new list of admins. A second component T_C (for commit) contains the updating information for group members. Both components are signed together using the committer's current $\gamma.\mathsf{ssk}$.

The Props method, given a commit, retrieves the list of proposals that it implements; this simply calls the underlying CGKA.Props algorithm and combines the output with the list of admin updates directly contained in a well-formed commit message.

Processing Control Messages. The Proc method takes a control message T as input and updates the state accordingly. The algorithm returns an acceptance bit acc which is true if

[7] Some precaution must be taken with respect to security when proposals are encrypted under the CGKA key, as the adversary gains access to multiple additional ciphertexts which can result in a security loss.

[8] This division is made for clarity, but it may be used to improve efficiency too. Namely, the welcome part W_0 of a commit message does not need to be processed by existing group members, so in principle C_0 can be sent only to these. In A-CGKAs such as IAS, this can also be applied if signatures are handled carefully.

the processing succeeds, in which case the state is updated. Otherwise, the state remains the same. During an execution of Proc, some checks must pass before the state is updated. For newly added users, p-Wel verifies the message signature on the adminList, attempts to process the message via the underlying CGKA, and updates the state given this succeeds. For group members, p-Com verifies the administrator signature and the signatures in the admin proposals. The state is updated if all verification succeeds; a removed user deletes their state, and temporary keys are updated if necessary. The case in which the control message T is not signed is handled by Proc directly by verifying that no changes to the group structure are made except possibly for valid signed self-removals.

Features

We first note that the IAS protocol can be built over any (unauthenticated or authenticated) CGKA. Since signatures are often already present in CGKAs such as [AJM22], the extension from CGKA to A-CGKA can be more direct (and thus incur less overhead) than presented here. This also holds true for any group messaging scheme, such as the administrated MLS extension we describe in Section 5.3.4.

Commit and Propose Policies. Our construction allows standard users to perform a commit if there are no changes in the group structure or administration. This is an optional design choice that does not affect security in our model (and could be reflected in a correctness predicate), although, as previously discussed, adversarial group members may deny service if the underlying CGKA is not robust. We also enforce that standard users cannot propose administrative changes (even if these could be later ignored by admins), and similarly can be allowed as required by an application.

Security Mechanisms. The security of the group administration is provided by the admin signatures; an adversary should not be able to commit changes to the group unless it compromises the state of one of the group administrators. The update mechanism provides optimal post-compromise security in our model.

On the other hand, administrative actions are undeniable and traceable both by group members and by the message delivery service. Separately, additional protections (i.e. checking members are registered on the PKI) are needed to ensure that parties are not invited to fake groups where the list of group administrators is forged.

On Optimal Forward Security. Note that, as defined, our construction does not satisfy forward security with respect to injection queries even if the underlying CGKA provides optimal forward security. Concretely, suppose that *ID* makes their last update in epoch 3, and then their state is exposed in epoch 5. Then *ID* can trivially forge commit messages for parties that are in epochs 3 and 4 since their keying material has not been updated. A similar forward security issue is present in the MLS standard affecting confidentiality [ACDT20]. Optimal security can be straightforwardly achieved by replacing regular signatures with *forward-secure*

signatures [BM99]. Forward-secure signatures allow signers to non-interactively update their secret keys and provide forward security given state exposure. In IAS, it suffices to use forward-secure signatures such that whenever an epoch passes and an admin has not sampled a new signature key, they invoke the signature scheme's secret key update function, where new signature keys are otherwise derived as in the construction. We note that forward-secure signatures involve an overhead that may be undesirable in some cases, and also they are not used in current protocols (signatures are already used in MLS's CGKA, for instance). In Theorem 17, we characterise the exact security of IAS using standard primitives via our sub-optimal predicate. In this way, the security of both alternatives is fully characterised.

Correctness and Security

We proceed to proving correctness and security.

Theorem 16. Let CGKA be a correct CGKA (Definition 42) and Sig be a 1-correct signature scheme. Then, the IAS protocol (Figures 5.6, 5.7 and 5.8) is correct with respect to Definition 42.

Proof. We want to prove that no adversary $\mathscr{A}$ can win $\mathsf{CORR}_{\mathsf{A\text{-}CGKA},\mathsf{C}_{\mathsf{corr}}}$ (where we set $\mathsf{C}_{\mathsf{corr}} = \mathsf{true}$) played with respect to IAS (Figures 5.6, 5.7 and 5.8) given that CGKA is correct. We analyze the different game oracles separately and sketch parts derived by direct inspection or based on CGKA correctness.

For PROP, $\mathscr{A}$ can win the game in PROP if either Prop-Info incorrectly interprets the proposal or if the Prop call changes the view of the group. In the first case, correctness follows from the correctness of CGKA.Prop-Info if the proposal is standard, and by inspection of IAS otherwise. In the second case, the group view is never changed by Prop (as makeAdminProp only updates $\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}'$ in case of an admin proposal, and CGKA.Prop is correct in the case of a standard proposal).

CREATE and COMMIT can be proven correct by inspection in a similar way.

For DELIVER, we examine each **reward** clause. Note that in line 2 of DELIVER, T is either a commit message created by Create or by Commit.

We start with the clause $(\emptyset \neq \gamma[\mathsf{gid}].G^* \subseteq \gamma[\mathsf{gid}].G)$. If T is a create message, then line 1 of Create and the fact that variable adminList is populated ensures that this condition is fulfilled for T. Upon processing, and after a correct PKI retrieval, p-Wel overwrites $\gamma.\mathsf{adminList}$ and $\gamma.\mathsf{s0}$ (via CGKA.Proc), so the condition holds. If T is a (standard) commit made by a non-admin user – that is, one without a signature – then there are no changes to the group as checked explicitly by the Proc algorithm from line 10. Otherwise, if T an admin commit, then it is processed by p-Comm. We can distinguish some further cases depending on the proposals contained in T:

- If T contains proposals of types upd, upd-adm, add only, then the condition is trivially met.
- If T additionally contains rem proposals, for every removed ID a corresponding rem-adm proposal is generated by an honest admin in propCleaner, hence $G^* \subseteq G$.
- If T additionally contains add-adm proposals, the valid(P) predicate checks that the added admins are already group members (via S_2), hence $G^* \subseteq G$.
- If T additionally contains rem-adm proposals, the final check in enforcePolicy ensures that $G \neq \emptyset$.
- Any other combination of several contradicting proposals affecting the same ID is handled by enforcePolicy, which prioritises removals (while preserving admin removals for the same user) which performs a final check on the size of G^*.

We conclude that, for any possible combination of proposals, the condition is always met provided that acc = true with respect to T.

The next cases are the **reward** Props(ST[ID], T) $\neq$ T[gid, (t,c), vec, c'] and the **reward** γ[gid].k $\neq \perp$ condition given Proc outputs γ such that $ID \notin \gamma$[gid].G. It is straightforward to see that both conditions are met by correctness of CGKA.Props.

The check by UpdateView rewards the adversary if two users processing the same commit message (on epoch (t,c)) differ in their group view. We show correctness by induction. Suppose ID_1 and ID_2 process the same commit message T. If they are in epoch $(-1,-1)$ and process a create message, correctness is easily seen. For the inductive step, we assume that their group views were equal in (t,c), and we want to show that they remain equal after moving to epoch $(t+1,c')$. The commit is handled by p-Comm, and the only parts that can change for different users are the **if** condition in line 10 and updAL. The behaviour of both sections of code varies only on the modification of γ's signature keys, but not on the group structure. Hence, by the correctness of CGKA, we conclude that ID_1 and ID_2 end up having consistent views.

The edge case in which a user is just added to the group is handled by p-Wel, and follows from CGKA correctness and the fact that γ.adminList $\leftarrow$ adminList is executed where adminList is directly provided in T.

Finally, the check **reward** γ[gid].$k \neq$ T[gid, (t,c), key, c'] follows from the correctness of the CGKA.Commit algorithm which outputs the new group key k. □

Recall that IAS, which uses a (regular) digital signature scheme, does not provide optimal forward security. Therefore, we prove security with respect to a sub-optimal admin cleanness predicate $\mathsf{C}_{\mathsf{adm}}$ where $\mathsf{C}_{\mathsf{adm}} = \mathsf{C}_{\mathsf{adm\text{-}opt}} \wedge \mathsf{C}_{\mathsf{adm\text{-}add}}$ and $\mathsf{C}_{\mathsf{adm\text{-}opt}}$ is defined in Section 5.2.4.

Theorem 17. Consider KIND CGKA CGKA with respect to cleanness predicate $\mathsf{C}_{\mathsf{cgka}}$, SUF-CMA signature scheme Sig and PRF H_4. Then, for A-CGKA IAS, we have, with respect

to predicates $\mathsf{C}_{\mathsf{cgka}}, \mathsf{C}_{\mathsf{adm}}, \mathsf{C}_{\mathsf{forgery}}$, that for every efficient adversary $\mathscr{A}$ that makes at most q oracle queries, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathrm{kind}}_{\mathsf{IAS}}(\mathscr{A}) \leq q \cdot \mathsf{Adv}^{\mathrm{prf}}_{H_4}(\mathscr{B}) + \mathsf{Adv}^{\mathrm{kind}}_{\mathsf{CGKA}}(\mathscr{B}) + q^2 \cdot \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{B}) .$$

Proof idea. We first bound an adversary $\mathscr{A}$'s advantage in distinguishing between the $\mathsf{KIND}_{\mathsf{A\text{-}CGKA}}$ game and a game G_1 which replaces calls to hash functions H_i by uniformly sampling the output (modelling each H_i in IAS as a PRF). Then, we divide $\mathscr{A}$'s behaviour in G_1 into two events based on whether they successfully query the INJECT oracle (event E_1) or not (event E_2). Given E_1, we reduce security via a number of SUF-CMA adversaries. Otherwise, we reduce directly with $\mathsf{KIND}_{\mathsf{CGKA}}$ adversary, at which point the claim follows.

We start with the definition of the predicate $\mathsf{C}_{\mathsf{adm}}$ predicate (Figure 5.9) that we prove IAS secure with respect to. After proving security, we discuss how one can (easily) extend the proofs and ensure optimal security using forward-secure signatures.

$$\begin{array}{|l|l|}\hline \mathsf{C}_{\mathsf{adm}}: & \forall\ (i, ID, ID', \mathsf{ctr} \in (0, \mathsf{exp\text{-}ctr}]) : q_i = \mathsf{INJECT}(ID', \cdot, t_i^*), \\ & \big(ID \notin \mathsf{ADM}[t_i^*]\big) \vee \\ & \big(\exists (t_i, c) : (\mathsf{tExp}(ID, \mathsf{ctr}).t_a < t_i \leq t_i^*) \wedge \\ & \mathsf{hasUpd}_{\mathsf{adm}}(ID, \mathsf{T}[(\cdot, t_i), \mathtt{com}, c], \mathsf{T}[(\cdot, t_i), \mathtt{vec}, c]) \wedge \\ & (\mathsf{C}[(\cdot, t_i)] = c)\big). \\ \hline \end{array}$$

Figure 5.9: Sub-optimal administrative predicate where the adversary makes oracle queries $q_1, \ldots, q_n$.

Note that it differs from the optimal predicate (Figure 5.5) only by the lack of $(t_i^* < \mathsf{tExp}(ID, \mathsf{ctr}).t_a)$ condition, so it holds that $\mathsf{C}_{\mathsf{adm}} \wedge \mathsf{C}_{\mathsf{adm\text{-}opt}} = \mathsf{C}_{\mathsf{adm}}$. In particular, the forward security guarantees are weaker since, e.g., if a party updates their admin key in admin epoch 3 then if they are exposed in epoch 5 then the adversary can make a trivial forgery in the construction (and thus it is considered a trivial attack by $\mathsf{C}_{\mathsf{adm}}$).

Towards proving security, we prove the following lemma.

Lemma 16. Let $\mathscr{A}$ be a $\mathsf{KIND}^{\mathscr{A}}_{\mathsf{A\text{-}CGKA}, \mathsf{C}_{\mathsf{cgka}}, \mathsf{C}_{\mathsf{adm}}, \mathsf{C}_{\mathsf{forgery}}}$ adversary playing with respect to IAS. Consider any query $\mathscr{A}$ makes of the form $\mathsf{INJECT}(ID, m, t_a)$ which outputs $v \neq \bot$. Then there is an efficient algorithm that, given the inputs/outputs of oracle queries $\mathscr{A}$ makes, parses $m = (\mathsf{gid}, T_C, T_W, \sigma_T)$ and derives pk such that $\mathsf{Vrfy}(\mathsf{pk}, \sigma_T, (\mathsf{gid}, T_C, T_W))) = 1$.

Proof. Consider a given query $\mathsf{INJECT}(ID, m, t_a)$ made by $\mathscr{A}$ that outputs $v \neq \bot$. Given $v \neq \bot$ and by definition of INJECT, a call $(\gamma, \mathsf{true}) \leftarrow \mathsf{Proc}(\mathsf{ST}[ID], m)$ was previously made by the challenger such that $\mathsf{C}_{\mathsf{forgery}} = \mathsf{true}$. Note $\mathscr{A}$ cannot register (malicious) signature keys with the PKI, and keys are assumed to be bound to the context that they are used in, e.g., for self-removals. In addition, any non-admin commit comprising of group changes that only consists of self-removes will not result in the adversary winning by construction of IAS,

and if a self-remove is created then even if another 'dishonest' self-remove can be created for that party, $\mathsf{C}_{\mathsf{forgery}}$ will consider it equivalent to the 'original' self-remove. Thus, self-removes cannot cause $\mathsf{C}_{\mathsf{forgery}}$ to be true. Now, INJECT disallows $t_a \neq -1$, which is the case if and only if $ID \notin G$, and that unsigned control messages cannot change the group structure. Thus, we only need to consider control messages that are input to p-Comm in Proc and result in output acc = true when $\sigma_T \neq \perp$. To reach p-Comm, the Proc call must be such that $\mathsf{Vrfy}(\gamma[\mathsf{gid}].\mathsf{adminList}[ID], (\mathsf{gid}, T_C, T_W), \sigma_T) = 1$. Since $\gamma[\mathsf{gid}].\mathsf{adminList}[ID]$ must have been previously sent in some commit or welcome message by construction of IAS, it follows that $\mathsf{pk} = \gamma[\mathsf{gid}].\mathsf{adminList}[ID]$ is efficiently computable. □

We prove IAS secure below.

Proof. Let G_0 be the $\mathsf{KIND}^{\mathscr{A}}_{\mathsf{IAS},\mathsf{C}_{\mathsf{cgka}},\mathsf{C}_{\mathsf{adm}},\mathsf{C}_{\mathsf{forgery}}}$ game. Let G_1 be as in G_0, except that all calls of the form $(r_1,\ldots,r_i) \leftarrow H_i(r_0,\gamma)$ are replaced with calls of the form $(r_1,\ldots,r_i) \xleftarrow{\$} R^i$ where R is the space of random coins used by each (A)-CGKA algorithm. Note that in IAS we always have $i \leq 4$; we assume for simplicity that each H_i for $i \leq 4$ is implemented by calling H_4 and then truncating the output as needed.

Let $G_{0,0} = G_0$. Let $G_{0,j}$ be G_0 except that the first j calls of the form $H_i(r_0,\gamma)$ that adversary $\mathscr{A}$ makes are replaced as above, and the rest remain unchanged. Note that since every oracle query that $\mathscr{A}$ makes results in at most one call to a function of the form $H_i(\cdot,\cdot)$, $G_{0,k} = G_1$ for some $k \leq q$.

Consider $G_{0,j-1}$ and $G_{0,j}$ where $j \geq 1$; suppose these games are played by adversary $\mathscr{A}$. Let $\mathscr{A}'$ be a PRF adversary (keyed with the first argument). $\mathscr{A}'$ simulates directly except when $\mathscr{A}$ makes their oracle query which leads to the j-th call to a function of the form H_i. Upon this call, $\mathscr{A}$ simulates this call by calling $\mathsf{EVAL}(\gamma)$ for the input γ, and truncates the response $(r_1,\ldots,r_4)$ to $(r_1,\ldots,r_i)$ when necessary before continuing execution (i.e., its simulation). Clearly $\mathscr{A}'$ perfectly simulates $G_{0,j-1}$ when $\mathscr{A}'$'s challenger's bit is 1. When $\mathscr{A}'$'s challenger's bit is 0, note that the output of $\mathscr{A}'$'s EVAL call, namely $(r_1,\ldots,r_4)$, is of the form $F(\gamma)$ for a uniformly random function $F : \mathsf{ST} \to R^4$, where ST is the A-CGKA state space. Since F is randomly chosen, this output is distributed identically to $(r_1,\ldots,r_4)$ where each r_i is uniformly sampled from R. It follows that $\mathscr{A}'$ perfectly simulates $G_{0,j}$ when its challenge bit is 0. By combining the sequence of game hops, it follows that

$$\left|\Pr[G_0(\mathscr{A}) \Rightarrow 1] - \frac{1}{2}\right| \leq \left|\Pr[G_1(\mathscr{A}) \Rightarrow 1] - \frac{1}{2}\right| + q \cdot \mathsf{Adv}^{\mathrm{prf}}_{H_4}(\mathscr{A}').$$

In the following, we will simulate for a G_1 adversary via either a SUF-CMA or CGKA adversary. Without hopping from G_0 to G_1, the simulation would not have been identical when e.g. using the SUF-CMA SIGN oracle. Since we have transitioned to G_1, which uses randomness normally, there are no issues regarding simulation and randomness.

Let $\mathscr{A}$ be a G_1 adversary. Let E_1 be the event that $\mathscr{A}$ makes a query to INJECT such that INJECT outputs value $v \neq \perp$ (i.e., the challenge bit). Let E_2 be the event that this does not occur; clearly $\Pr[E_1] + \Pr[E_2] = 1$. We consider each event separately. Without loss of generality, we restrict our simulations given E_1 to the case where $\mathsf{C}_{\mathsf{adm}}$ is true as otherwise the adversary cannot win, and similarly given E_2 to the case where $\mathsf{C}_{\mathsf{cgka}}$ is true. It suffices to observe that these two predicates are efficiently computable such that the adversaries we construct below can abort during unclean executions.

$\underline{E_1}$: By construction of IAS and Lemma 16, note, given that INJECT outputs $v \neq \perp$, that a signature forgery has occurred where the signature keying material is sampled due to an oracle call. Given E_1, we need to determine which input values ID and t_a are used by $\mathscr{A}$ on the first INJECT call which outputs $v \neq \perp$. By construction of IAS, this can happen as a result of a query to CREATE, PROP or COMMIT. However, $\mathscr{A}$ may make at most q injection attempts with respect to this key pair, each of which may be a winning one. Thus, the reduction has to guess both 1) the INJECT query which first outputs $v \neq \perp$ and 2) the query q_i which generates the signature key pair corresponding to this injection.

Let $E_{1,i,j}$ be the event that $\mathscr{A}$ makes query q_i which leads to the generation of signature key pair $(\mathsf{ssk}, \mathsf{spk})$ such that query q_j is the first query to INJECT resulting in output $v \neq \perp$[9]. Note that IAS is such that each oracle query q_i results in at most one new signature key pair being sampled by the challenger. By the union bound, we have:

$$\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_1] \leq \sum_{i,j \in \{1,\ldots,q\}} \Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_{1,i,j}].$$

Suppose $E_{1,i,j}$ holds. Let $\mathscr{A}'$ be an SUF-CMA adversary that simulates for G_1 adversary $\mathscr{A}$. We aim to show that $\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_{1,i,j}] \leq \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{A}')$.

$\mathscr{A}'$ simulates as follows. $\mathscr{A}'$ simulates variable initialisation as in G_1 except that it lasily simulates Init calls (ensuring it remains polynomially-bounded). $\mathscr{A}'$ simulates the first $i-1$ of $\mathscr{A}$'s oracle queries locally, i.e., simulates all relevant behaviour resulting in corresponding state and game variables being set and updated. This includes queries of the form getSsk and getSpk which $\mathscr{A}'$ simulates via signature scheme Sig.

Consider $\mathscr{A}'$'s i-th oracle query q_i. Let $(\mathsf{ssk}^*, \mathsf{spk}^*)$ denote the signature key pair sampled by the SUF-CMA challenger; recall that SUF-CMA adversary $\mathscr{A}'$ has access to oracle SIGN. $\mathscr{A}'$ simulates as follows:

- If q_i is to CREATE, then $(\mathsf{ssk}^*, \mathsf{spk}^*)$ plays the role of $(\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}')$. Namely, $\mathscr{A}'$ sets the output of $\mathsf{getSpk}(ID, \gamma.ME)$ to spk^* and that of $\mathsf{getSsk}(\gamma.\mathsf{spk}', ME)$ to ssk^* after embedding spk^* in $\mathsf{adminList}[ME]$ at line 7 of Create. $\mathscr{A}'$ also calls $\mathsf{SIGN}((\mathsf{gid}, \perp, T_W))$ which

[9] Note that we only consider the first such query since the simulation will end after this point.

outputs σ. $\mathscr{A}'$ otherwise simulates and returns the result to $\mathscr{A}$ (which includes σ).

- If q_i is to PROP with input type = upd-adm, then $(\mathsf{ssk}^*, \mathsf{spk}^*)$ plays the role of $(\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}')$ (line 10 of makeAdminProp). Similarly to above, $\mathscr{A}'$ embeds spk^* in P_0 (line 11 of makeAdminProp), and then simulates the rest of the oracle call.
- Otherwise, q_i is to COMMIT. Note we assume $E_{1,i,j}$ holds. Thus, the branch at line 5 in Commit must be executed. As before, $(\mathsf{ssk}^*, \mathsf{spk}^*)$ plays the role of $(\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}')$; $\mathscr{A}'$ simulates the remainder of the call.

All other oracle queries, except to INJECT, are simulated locally by $\mathscr{A}'$ except when signatures with respect to spk^* are required, in which case SUF-CMA oracle SIGN is used, or when spk^* is to be embedded in a message. Note that at most q SIGN queries are made by $\mathscr{A}'$ since each oracle query $\mathscr{A}$ makes produces at most one signature (which may or may not require SIGN to simulate).

Note by construction of IAS that each signature key pair is sampled by a single party and is never revealed/embedded in another message, and that each key pair is uniformly and independently sampled. Thus, the simulation is valid, ignoring EXPOSE queries, since the challenge key pair is independent of all other keying material and challenge secret key ssk^* is never revealed. Regarding EXPOSE queries, since $\mathsf{C}_{\mathsf{adm}}$ is true (since we assume E_1), $\mathscr{A}$ will never query EXPOSE with respect to the challenge key pair, a query which would otherwise not be able to be simulated.

For $\mathscr{A}$'s query $q_{j'} = \mathsf{INJECT}(ID)$ such that $j' \neq j$, $\mathscr{A}'$ simulates by returning $\perp$ to $\mathscr{A}$. When $\mathscr{A}$ makes query $q_j = \mathsf{INJECT}(ID, m, t_a)$, $\mathscr{A}'$ inspects $m = (\mathsf{gid}, T_C, T_W, \sigma_T)$ and returns the message/signature pair $((\mathsf{gid}, T_C, T_W), \sigma_T)$ to $\mathscr{A}$ which, by Lemma 16, exists. These two steps are valid by definition of $E_{1,i,j}$ and that $j' < j$ in the first step since we only simulate up to query j.

It thus follows that the simulation is perfect. Noting that $\mathscr{A}$ wins at most as often as $\mathscr{A}'$ (since e.g., $\mathscr{A}$ may come up with a forgery (m, σ) nevertheless rejected by Proc), we have:

$$\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_{1,i,j}] \leq \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{A}').$$

$\underline{E_2}$: Note first that given E_2, we can deduce that every query to INJECT will have output $\perp$. Let $\mathscr{A}'$ be a $\mathsf{KIND}^{\mathscr{A}'}_{\mathsf{CGKA},\mathsf{C}_{\mathsf{cgka}}}$ adversary. $\mathscr{A}'$ simulates for $\mathsf{KIND}^{\mathscr{A}}_{\text{A-CGKA},\mathsf{C}_{\mathsf{cgka}},\mathsf{C}_{\mathsf{adm}},\mathsf{C}_{\mathsf{forgery}}}$ adversary $\mathscr{A}$ as follows. When $\mathscr{A}$ makes an oracle query, $\mathscr{A}'$ executes all code in IAS except for that which makes use of CGKA algorithms, which are processed via $\mathscr{A}'$'s oracles; $\mathscr{A}'$ then returns each response to $\mathscr{A}$. One case we must deal with is when Proc is called at line 11; the call may succeed but the caller may ignore the state update, undoing the changes made; this happens given the group state changes as a result of the call. Note that $\mathscr{A}'$ can determine whether this case occurs or not using the fact that commit messages are honestly delivered by construction of the KIND game and by correctness which ensures that honestly-generated and delivered

commit messages are accepted. In particular, $\mathscr{A}'$ can use the policy to determine whether or not the call would update the group state, and only call its DELIVER when the group state is not changed. Finally, $\mathscr{A}'$ outputs the same bit as $\mathscr{A}$.

To see that the simulation is perfect, first note that CGKA algorithms are used as black boxes in IAS. Moreover, except in the case dealt with above, CGKA state variables $\mathsf{s}0$ for each *ID* are not used except as input to CGKA algorithms, after which they are immediately overwritten, exactly as done by the CGKA KIND challenger given correctness and in particular the fact that failing algorithm calls do not update the state. Thus, it suffices to simulate CGKA code using $\mathscr{A}'$'s oracles. Thus:

$$\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_2] = \Pr[\mathsf{KIND}_{\mathsf{CGKA},\mathsf{C}_{\mathsf{cgka}}}(\mathscr{A}') \Rightarrow 1]$$

We then have:

$$\begin{aligned}
\left|\Pr[G_1(\mathscr{A}) \Rightarrow 1] - \frac{1}{2}\right| &= \left|\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge (\vee_{i,j} E_{1,i,j} \vee E_2)] - \frac{1}{2}\right| \\
&\leq \left|\sum_{i,j} \Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_{1,i,j}] + \Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_2] - \frac{1}{2}\right| \\
&\leq \left|\sum_{i,j} \Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_{1,i,j}]\right| + \left|\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_2] - \frac{1}{2}\right| \\
&\leq q^2 \cdot \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{A}') + \mathsf{Adv}^{\mathrm{kind}}_{\mathsf{CGKA}}(\mathscr{A}') \,,
\end{aligned}$$

where the second and third lines follow from the union bound and triangle inequality, respectively. The result follows by combining this with the game hop earlier. □

Optimal Forward Security. As described above, we can achieve optimal forward security, and thus optimal admin security (i.e., security with respect to $\mathsf{C}_{\mathsf{adm\text{-}opt}}$) by replacing signatures with forward-secure signatures [BM99, MMM02]. The logic of the security reduction is very similar to that presented with regular signatures above. The main difference is that forward-secure signature calls are replaced by oracle calls, including possibly key exposure calls.

5.3.2 Dynamic Group Signature

In our second construction, *dynamic group signature* (DGS), the group administrators agree on a *common* signature key pair that they use for signing administrative messages on an underlying CGKA. To agree on a secret and generate a common key pair, they run a separate CGKA. As opposed to IAS, group administrators may now be opaque to group members if the concrete CGKA which is used allows it. The reason is that they authenticate admin messages using an admin signature key that is shared among all admins. Notably, group members do not need to keep track of an administrator list; admins implicitly track this via their CGKA.

5.3.3 Description

Protocol. DGS is introduced in Figures 5.10 and 5.11. In the algorithm, we refer to the primary (or standard) CGKA as CGKA, and to the administrative CGKA as CGKA^*. The first CGKA allows group members to agree on a common secret and group over time as in IAS, whereas the second exists only for administrative purposes (i.e., admins deriving a common signature key). Note that CGKA^* need not be implemented in the same way as the primary CGKA. This can be exploited by a protocol designer either for performance reasons or if, for instance, stronger FS and PCS guarantees are desired for the administrative CGKA. For simplicity of exposition, DGS as written does not support self-signed removal operations that non-admins can commit directly, but we note that the technique to implement them is identical to IAS.

States. Each party stores $\gamma.\mathsf{s0}$, corresponding to the primary CGKA, as well as $\gamma.\mathsf{sA}$, corresponding to CGKA^*, which are used for each group they consider. For a given gid, let gid^* be another group identifier such that for all $\mathsf{gid}_1 \neq \mathsf{gid}_2$, $\mathsf{gid}_1 \neq \mathsf{gid}_2 \neq \mathsf{gid}_1^* \neq \mathsf{gid}_2^*$. We assume for DGS that gid is used by the main CGKA and gid^* by the admin CGKA. Besides these fields, the state includes the administrative public key $\gamma[\mathsf{gid}].\mathsf{spk}$ known by all group members (and can be a public group parameter, known for instance by a central server) to enable verification. The state variables are now $\gamma[\mathsf{gid}].G = \gamma.\mathsf{s0}[\mathsf{gid}].G$, $\gamma[\mathsf{gid}].G^* = \gamma.\mathsf{sA}[\mathsf{gid}^*].G$, and $\gamma[\mathsf{gid}].\mathsf{k} = \gamma.\mathsf{s0}[\mathsf{gid}].\mathsf{k}$.

PKI. As in IAS, we assume an incorruptible PKI functionality. Similar to IAS, DGS relies on abstract helpers $\mathsf{registerKey}(\mathsf{gid})$ and $\mathsf{getSpk}(\mathsf{gid})$, both of which now only take an identifier gid as input. We assume that admins register their admin signature public keys whenever they are updated or created, which either happens at group creation time or during a Commit call; nonetheless we only require that each user calls $\mathsf{getSpk}(\mathsf{gid})$ when they join group gid. Authentication could be implemented while ensuring k-anonymity such that a member authenticates his group membership but not his identity; such a feature cannot be provided by IAS without modification.

Initialisation. The Init procedure calls the $\mathsf{CGKA.Init}$ and $\mathsf{CGKA}^*.\mathsf{Init}$ algorithms to initialise $\gamma.\mathsf{s0}$ and $\gamma.\mathsf{sA}$, respectively, and sets $\gamma[\cdot].\mathsf{spk}, \gamma[\cdot].\mathsf{ssk} \leftarrow \perp$.

Group Creation. The Create algorithm creates a group for the two separate CGKAs by calling the two corresponding Create methods. These calls output new states $\mathsf{s0}$ and sA, which overwrite the stored states, as well as control messages W_0 and W_A, which are collated into a *create* control message $T = T_{CR}$. We assume that an initial group signature public key is sampled and uploaded to the PKI by the caller of Create.

Proposals. The Prop algorithm generates a proposal message P by using $\mathsf{CGKA.Prop}$ the input type is standard and $\mathsf{CGKA}^*.\mathsf{Prop}$ when it is administrative (i.e., of the form $*\text{-}\mathsf{adm}$). As in IAS, a validity check on the caller ID' and the target ID of the proposal is made using the $\mathsf{valid}(P)$ predicate (from IAS). Administrative proposals are signed with $\gamma.\mathsf{spk}$; this is done to protect against insider adversaries that may re-send previously crafted administrative proposals (i.e.,

DGS.Init(1^λ, ID)

$\gamma.\mathsf{s0} \xleftarrow{\$} \mathsf{CGKA.Init}(1^\lambda, ID)$
$\gamma.\mathsf{sA} \xleftarrow{\$} \mathsf{CGKA}^*.\mathsf{Init}(1^\lambda, ID)$
$\gamma.\mathsf{ME} \leftarrow ID$; $\gamma.1^\lambda \leftarrow 1^\lambda$
$\gamma[\cdot].\mathsf{spk}, \gamma[\cdot].\mathsf{ssk} \leftarrow \bot$

DGS.Create(gid, $G, G^*; r_0$)

require $(\gamma.\mathsf{ME} \in G^*) \wedge (G^* \subseteq G)$
$(r_1, r_2, r_3, r_4) \leftarrow H_4(r_0, \gamma)$
$(W_0, \gamma.\mathsf{s0}) \leftarrow \mathsf{CGKA.Create}(\gamma.\mathsf{s0}, \mathsf{gid}, G; r_1)$
$(W_A, \gamma.\mathsf{sA}) \leftarrow \mathsf{CGKA}^*.\mathsf{Create}(\gamma.\mathsf{sA}, \mathsf{gid}^*, G^*; r_2)$
$T_{CR} \leftarrow$ ('create', W_0, W_A)
$(\gamma[\mathsf{gid}].\mathsf{spk}, \gamma[\mathsf{gid}].\mathsf{ssk}) \leftarrow \mathsf{Gen}(\gamma.1^\lambda; r_3)$; registerKey(gid)
$\sigma_T \leftarrow \mathsf{Sign}(\gamma.\mathsf{ssk}, (\mathsf{gid}, T_{CR}); r_4)$
return $(\mathsf{gid}, T_{CR}, \bot, \bot, \sigma_T)$

DGS.Prop(gid, ID, type; r_0)

$(r_1, r_2) \leftarrow H_2(r_0, \gamma)$
if type = $*$-adm
  require $\gamma.\mathsf{ME} \in \gamma.\mathsf{sA}.G$
  $(\gamma.\mathsf{sA}, P_0) \leftarrow \mathsf{CGKA}^*.\mathsf{Prop}(\gamma.\mathsf{sA}, \mathsf{gid}^*, ID, \mathsf{type}; r_1)$
  $P \leftarrow (P_0, \mathsf{Sign}(\gamma.\mathsf{ssk}, P_0; r_2))$
else
  $(\gamma.\mathsf{s0}, P) \leftarrow \mathsf{CGKA.Prop}(\gamma.\mathsf{s0}, \mathsf{gid}, ID, \mathsf{type}; r_1)$
return P

DGS.Prop-Info(P)

$(\mathsf{gid}, \mathsf{type}, \mathsf{ID}, \mathsf{ID}') \leftarrow \mathsf{CGKA.Prop\text{-}Info}(\gamma.\mathsf{s0}, P)$
if $\gamma.\mathsf{sA}[\mathsf{gid}] \neq \bot$ // Must be an admin proposal
  type $\leftarrow$ type||-adm
return $(\mathsf{gid}, \mathsf{type}, \mathsf{ID}, \mathsf{ID}')$

DGS.Props(T)

// Supports non-welcome control messages
$(\mathsf{gid}, T_{CR}, T_W, T_C, \sigma_T) \leftarrow T$
$\vec{P}_0 \leftarrow \mathsf{CGKA.Props}(T_C.C_0)$; $\vec{P}_A \leftarrow \mathsf{CGKA}^*.\mathsf{Props}(T_C.C_A)$
return $\vec{P}_0 || \vec{P}_A$

DGS.Commit(gid, $\vec{P}$, com-type; r_0)

require $\gamma.\mathsf{ME} \in \gamma.G$
require com-type $\in$ {adm, std, both}
$(r_1, r_2, r_3) \leftarrow H_3(r_0, \gamma)$
$C_0, C_A, W_0, W_A, k \leftarrow \bot$
$(\vec{P}_0, \vec{P}_A, \mathsf{admReq}) \leftarrow \mathsf{propCleaner}(\mathsf{gid}, \vec{P})$
if admReq $\vee$ (com-type $\in$ {adm, both})
  require $\gamma.\mathsf{ME} \in \gamma.G^*$
  $(\mathsf{ssk}, \mathsf{spk}) \leftarrow \mathsf{getSigKey}(\gamma.\mathsf{sA}.k)$ // old keys
  if com-type $\in$ {adm, both} // update spk
    $(\mathsf{spk}, C_A, W_A) \leftarrow \mathsf{c\text{-}Adm}(\mathsf{gid}, \vec{P}_A; r_1)$
  if com-type $\in$ {std, both}
    $(C_0, W_0) \leftarrow \mathsf{c\text{-}Std}(\mathsf{gid}, \vec{P}_0, k; r_2)$
  $T_C \leftarrow$ ('comm', C_0, C_A, W_A, spk)
  $T_W \leftarrow$ ('wel', W_0, spk)
  $\sigma_T \leftarrow \mathsf{Sign}(\mathsf{ssk}, (\mathsf{gid}, T_C, T_W); r_3)$
else // can be done by non-admins
  $(C_0, \bot, k) \leftarrow \mathsf{c\text{-}Std}(\vec{P}_0; r_1)$
  $T_C \leftarrow (\mathsf{gid}, \text{'comm'}, C_0, \bot, \bot)$
  $T_W, \sigma_T \leftarrow \bot$
if $k = \bot$ $k \leftarrow \gamma.\mathsf{s0}.k$
return $((\bot, T_C, T_W, \sigma_T), k)$

DGS.Proc(T)

$(\mathsf{gid}, T_{CR}, T_W, T_C, \sigma_T) \leftarrow T$; acc $\leftarrow$ false
if $T_{CR} \neq \bot \wedge T_W = T_C = \bot$
  if $\gamma.\mathsf{ME} \in \gamma[\mathsf{gid}].\mathsf{s0}.G$ **return** false
  acc $\leftarrow \mathsf{p\text{-}Create}(\mathsf{gid}, T_{CR}, \sigma_T)$
else if $(\gamma.\mathsf{ME} \notin \gamma[\mathsf{gid}].G) \wedge (T_W \neq \bot)$
  acc $\leftarrow \mathsf{p\text{-}Wel}(\mathsf{gid}, T_C, T_W, \sigma_T)$
else if $(\gamma.\mathsf{ME} \in \gamma[\mathsf{gid}].G) \wedge (T_C \neq \bot)$
  acc $\leftarrow \mathsf{p\text{-}Comm}(\mathsf{gid}, T_C, T_W, \sigma_T)$
return acc

Figure 5.10: Dynamic group signature (DGS) construction of an A-CGKA, built from two (possibly different) CGKAs CGKA and CGKA*, a signature scheme Sig, and PRFs $H_n : \mathscr{R} \times \mathsf{ST} \to \mathscr{R}^n$ for $n \leq 4$, randomness space $\mathscr{R}$ and state space ST. We let $\gamma[\mathsf{gid}].G = \gamma[\mathsf{gid}].\mathsf{s0}.G$, $\gamma[\mathsf{gid}].G^* = \gamma[\mathsf{gid}].\mathsf{sA}.G$, and $\gamma[\mathsf{gid}].\mathsf{k} = \gamma[\mathsf{gid}].\mathsf{s0}.\mathsf{k}$.

c-Adm(gid, $\vec{P}_A$; r_1)

```
1: for P ∈ P⃗_A
2:   if P.type = add-adm
3:     require P.ID ∈ γ.G
4:   (γ.sA, (C_A, W_A), k) ←
         CGKA*.Commit(γ.sA, gid*, P⃗_A; r_1)
5: if C_A = ⊥  return ⊥
6: (ssk, spk) ← getSigKey(k)
7: return (spk, C_A, W_A)
```

c-Std(gid, $\vec{P}_0$; r_2)

```
1: (γ.s0, C_0, W_0, k) ← CGKA.Commit(γ.s0, gid, P⃗_0; r_2)
2: return (C_0, W_0, k)
```

propCleaner(gid, $\vec{P}$)

```
1: admReq ← false;  P⃗_0, P⃗_A ← []
2: for P ∈ P⃗
3:   (gid', type, ID, ID') ← prop-info(P)
4:   if gid' = gid* ∧ P.type = *-adm ∧ IAS.valid(P)
5:     P⃗_A ← [P⃗_A, P]
6:     admReq ← true
7:   else if gid' = gid
8:     P⃗_0 ← [P⃗_0, P]
9:     if type ∈ {add, rem}
10:      admReq ← true
       // admin rem from G ⟹ rem also from G*
11:    if (type = rem) ∧ (ID ∈ γ.G*)
12:      P' ← CGKA*.Prop(γ.sA, gid*, ID, rem)
13:      P⃗_A ← [P⃗_A, P']
14: (P⃗_0, P⃗_A) ← enforcePolicy(P⃗_0, P⃗_A)
15: return (P⃗_0, P⃗_A, admReq)
```

p-Create(gid, T_{CR}, σ_T)

```
1: (msg-type, W_0, W_A) ← T_CR
2: require msg-type = 'create'
3: (γ_0, acc) ← CGKA.Proc(γ.s0, W_0)
4: (γ_A, ⊥) ← CGKA*.Proc(γ.sA, W_A)
5: if γ_A[gid].G = ⊥ ∨ (∅ ≠ γ_A[gid].G ⊆ γ_0[gid].G)
6:   (γ.s0, γ.sA) ← (γ_0, γ_A)
7: else return false
8: return acc ∧ Vrfy(getSpk(gid), (gid, T_CR), σ_T)
```

p-Wel(gid, T_C, T_W, σ_T)

```
1: (msg-type, W_0, spk) ← T_W
2: require msg-type = 'wel' ∧ getSpk(gid) = spk
3: if ¬Vrfy(spk, (gid, T_C, T_W), σ_T)  return false
4: (γ', acc) ← CGKA.Proc(γ.s0, W_0)
5: if ¬acc  return false
6: γ[gid].s0 ← γ'
7: γ[gid].spk ← spk
8: return true
```

p-Comm(gid, T_C, T_W, σ_T)

```
1: (msg-type, C_0, C_A, W_A, spk) ← T_C
2: require msg-type = 'comm'
3: γ' ← γ.sA
4: if σ_T = ⊥  // no sig ⇒ check no changes to G
5:   (γ', acc) ← CGKA.Proc(γ.s0, C_0)
6:   if ¬acc ∨ (γ'[gid].G ≠ γ[gid].G)
7:     return false
8:   γ[gid].s0 ← γ'
9:   return true
10: else if ¬Vrfy(γ[gid].spk, (gid, T_C, T_W), σ_T)
11:   return false
12: if γ.ME ∈ γ.G*
13:   (γ', acc) ← CGKA*.Proc(γ.sA, C_A)
14:   if ¬acc  return false
15: else if W_A ≠ ⊥
16:   (γ', ⊥) ← CGKA*.Proc(γ.sA, W_A)
17: if C_0 ≠ ⊥
18:   (γ†, acc†) ← CGKA.Proc(γ.s0, C_0)
19:   if ¬acc†  return false
20:   γ[gid].s0 ← γ†
21:   if γ.ME ∉ γ†.G  // removed user
22:     γ[gid] ← ⊥
23:     return true
24: γ[gid].sA ← γ'; γ[gid].spk ← spk
25: return true
```

getSigKey(r)

```
1: (ssk, spk) ← KeyGen(1^λ; H_ro(r))
2: return (ssk, spk)
```

enforcePolicy($\vec{P}_0, \vec{P}_A$)

```
1: return (P⃗_0, P⃗_A) ← IAS.enforcePolicy(P⃗_0, P⃗_A)
```

Figure 5.11: Helper functions for DGS in Figure 5.10 w.r.t. random oracle $H_{\mathsf{ro}} : \{0,1\}^\lambda \to \{0,1\}^\lambda$.

those that are legitimate but correspond to a previous epoch), or even create new ones if these are sent in plaintext. The Prop-Info algorithm is fully based on the respective Prop-Info algorithms (which in principle may not necessarily output fourth value ID' if anonymous proposals are allowed); we assume Props is likewise inherited from the underlying CGKAs.

Commits. For a given gid, administrative changes are committed via CGKA*.Commit (which outputs (C_A, W_A)) and standard group changes via CGKA.Commit (outputting (C_0, W_0) as usual). Note that the CGKA* secret $\mathsf{k}_{\mathsf{adm}}$ associated with the CGKA*.Commit call is used to generate each admin signature key pair $(\mathsf{ssk}', \mathsf{spk}')$.

The new admin key spk' is included in the final A-CGKA commit message so that (non-admin) group members can process it. In order to prove the authenticity of the commit (and of spk'), the committer signs the whole commit message including C_A, W_A, C_0, W_0 and spk' with the old admin signing key $\gamma.\mathsf{ssk}$. In addition, the committer must verify all proposal signatures in advance.

As before, a commit can be split into a welcome message T_W for newly added users, and a commit message T_C for group members. These are signed jointly in our construction to simplify the security proof, but may also be signed separately. In T_C, we also include the welcome messages to CGKA*, since they must always be addressed to current group members (i.e., of G). In a given Commit call, one or both of CGKA and CGKA* may be updated, and in particular CGKA but not CGKA* (for a non-admin commit) and vice-versa (for an admin-only commit).

Processing Control Messages. The Proc method takes a control message T, determines the type of message (create, welcome, or commit) and the gid, and updates the state only if processing succeeds ($\mathsf{acc} = \mathsf{true}$). Newly added users verify the admin signature, process the welcome message using CGKA.Proc and store the new public admin key (spk', provided in T) in $\gamma.\mathsf{spk}$.

Group members verify the administrator signature (if the commit requires administrative rights) using $\gamma.\mathsf{spk}$. Then, depending on the commit type at least one of CGKA and CGKA* are updated via the corresponding CGKA Proc algorithm. Given CGKA* or both CGKAs are updated, the updated admin key is set as $\gamma.\mathsf{spk} \leftarrow \mathsf{spk}'$. In case T contains a create message T_{CR}, both CGKAs process the respective welcome messages contained in T_{CR} separately.

Features

DGS allows the use of two distinct and independent CGKA protocols that authenticate admins as a group, providing some notable features that differ from IAS. One can also imagine a 'hybrid' approach where users run IAS except that some IAS keys are maintained and updated via their own CGKA and/or a hierarchy of CGKAs is employed.

Minimal Information Reveal. As opposed to IAS, the set of group administrators can be opaque to the central server and to the rest of the group (whenever the underlying CGKAs preserve the anonymity of group members with respect to external parties); this is reflected in the way DGS uses a PKI.

Limitations. A drawback of DGS is that enforcing different "levels of administration", for which IAS can be easily extended, is not straightforward. Nevertheless, one can still implement minor policies such as muting users at an application level (as done in practice). We also note that admins may not have a reliable view of the set of admins if CGKA^* is susceptible to insider attacks that violate robustness[10]. If these attacks are relevant, one can deploy heavier protocols such as the P-Act-Rob protocol from Alwen et al. [ACJM20]. A third limitation is that admins cannot give up their admin status immediately; they must send a self rem-adm proposal, erase their admin state, and wait for another admin to commit. This occurs generally in CGKA when a member leaves a group; to minimise this delay, the MLS standard requires users to commit immediately upon receipt of a valid proposal (in this case a removal). This problem could nevertheless be solved using the same approach as for self-removes in IAS, i.e., allowing admin self-removes without an admin signature.

Security Mechanisms. We note the conceptual simplicity of achieving PCS and FS for the group administration keys (in the adversarial model for CGKA^*) given the existence of secure CGKA schemes in the literature, since both properties are ensured by CGKA^* itself. Update mechanisms are largely simplified due to a single admin key being used. Delegation and revocation of admin keys are also straightforward.

Correctness and Security

Theorem 18. Let CGKA and CGKA^* be correct CGKAs, and Sig be a 1-correct signature scheme. Then, the DGS protocol (Figures 5.10 and 5.11) is correct with respect to Definition 42.

Proof. We prove that no adversary $\mathscr{A}$ can win $\mathsf{CORR}_{\mathsf{A\text{-}CGKA},\mathsf{C}_{\mathsf{corr}}}$ (where we set $\mathsf{C}_{\mathsf{corr}} = \mathsf{true}$) played with respect to DGS (Figures 5.10 and 5.11) given that $\mathsf{CGKA}, \mathsf{CGKA}^*$ are correct. We omit some details which are analogous to IAS' correctness proof (note that IAS and DGS are designed such that they share sections of their code).

For PROP, the correctness of Prop-Info follows from the correctness of $\mathsf{CGKA}.\mathsf{Prop\text{-}Info}, \mathsf{CGKA}^*.\mathsf{Prop\text{-}Info}$ by assumption. Also, the adversary cannot win after the group membership check as Prop only modifies the state by calling $\mathsf{CGKA}.\mathsf{Prop}$ and $\mathsf{CGKA}^*.\mathsf{Prop}$; which are correct by assumption.

The group membership check must always pass in CREATE and COMMIT for identical reasons.

[10] This scenario is out of the scope of our security model where admins are fully trusted.

For DELIVER, we examine each **reward** clause as done previously with IAS. As before, note that in line 2 of DELIVER, T is a commit message created either by Create or by Commit. Also, it is easy to see that the CGKA.Props check in PROP holds.

The clause $(\emptyset \neq \gamma[\mathsf{gid}].G^* \subseteq \gamma[\mathsf{gid}].G)$ is met upon generation of any create message T (produced by CREATE) by construction (line 1 of Create). When any message is processed by p-Create, the condition is enforced again.

If T is a (standard) commit made by a non-admin user – that is, one without a signature – then there are no changes to the group in the commit as checked explicitly by the auxiliary c-Std upon commit. The message must only contain a T_C which is processed by p-Comm, which again enforces this condition.

If T an admin commit, then it is processed by p-Comm or by p-Wel. In both cases, commit messages are processed by the underlying CGKA, CGKA* methods. Therefore, correctness depends on the Commit algorithm. The case distinction follows the exact same logic as in IAS, since the algorithms propCleaner and enforcePolicy, and the predicate valid(P) enforce the same conditions.

The next case is the **reward** $\gamma[\mathsf{gid}].\mathsf{k} \neq \bot$ for a removed (or non-member) ID, which is enforced in DGS by p-Comm and by the correctness of the CGKAs.

For the last check by UpdateView, one can proceed by induction as in IAS. The main difference is that the admin update is not done manually as in IAS (i.e., modifying adminList), but rather by the underlying Proc algorithms of CGKA*, which yields the result easily. The last check for the consistency of the derived group key also follows as in IAS. We omit the details. □

Theorem 19. Consider KIND CGKA CGKA with respect to cleanness predicate $\mathsf{C}_{\mathsf{cgka}}$, KIND CGKA CGKA* with respect to $\mathsf{C}_{\mathsf{cgka}*}$, SUF-CMA signature scheme Sig and PRF H_4. Then, for A-CGKA DGS, we have, with respect to cleanness predicates $\mathsf{C}_{\mathsf{cgka}}, \mathsf{C}_{\mathsf{adm}}, \mathsf{C}_{\mathsf{forgery}} = \mathsf{C}_{\mathsf{forgery}}{}^*$, that for every efficient adversary $\mathscr{A}$ that makes at most q oracle queries, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathrm{kind}}_{\mathsf{DGS}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{kind}}_{\mathsf{CGKA}}(\mathscr{B}) + q \cdot \left(\mathsf{Adv}^{\mathrm{prf}}_{H_4}(\mathscr{B}) + \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{B}) + q_{\mathsf{ro}} \cdot \mathsf{Adv}^{\mathrm{kind}}_{\mathsf{CGKA}^*}(\mathscr{B}) + 2^{-\lambda}\right),$$

where $\mathsf{C}_{\mathsf{adm}}$ is a function of $\mathsf{C}_{\mathsf{cgka}*}$ defined below and $\mathsf{C}_{\mathsf{forgery}}{}^*$ is defined in Section 5.2.4.

Proof idea. We first describe $\mathsf{C}_{\mathsf{adm}}$. Intuitively, $\mathsf{C}_{\mathsf{adm}}$ ensures that the set of safe oracle queries for DGS adversary $\mathscr{A}$ given inject queries of the form $q_i = \mathsf{INJECT}(ID, m, t_a)$ are those that are safe for CGKA* adversary $\mathscr{A}'$ under essentially the same queries, replacing $\mathsf{INJECT}(\cdot, \cdot, t_a)$ queries with queries of the form $\mathsf{CHAL}(t_a)$. To prove security, we use a similar game-hopping argument as in IAS. We first replace H_i calls using the PRF assumption. We then consider E_1 (a successful injection is made) and E_2 (otherwise) as in IAS. Given E_2, we can simulate directly via the CGKA adversary. Given E_1, we simulate differently depending on whether $\mathscr{A}$ makes a query to random oracle H_{ro} with a correct CGKA* key before the successful injection

or not. Here, if $\mathscr{A}$ is successful, we reduce security to the CGKA* adversary by intercepting the relevant random oracle query and guessing the correct bit using information from CHAL. Otherwise, we can simulate via an EUF-CMA adversary as in IAS since the signature key is now uniform from the adversary's perspective.

Predicate $\mathsf{C}_{\mathsf{adm}}$. $\mathsf{C}_{\mathsf{adm}}$ is tailored to DGS and is a function of the underlying CGKA* predicate $\mathsf{C}_{\mathsf{cgka}*}$. Intuitively, $\mathsf{C}_{\mathsf{adm}}$ ensures that the set of safe oracle queries for DGS adversary $\mathscr{A}$ given inject queries of the form $q_i = \mathsf{INJECT}(ID, m, t_a)$ are those that are safe for CGKA* adversary $\mathscr{A}'$ (i.e., the predicate $\mathsf{C}_{\mathsf{cgka}*}$) under roughly the same queries, replacing at most one $\mathsf{INJECT}(\cdot,\cdot,t_a)$ queries with a $\mathsf{CHAL}(t_a)$ query. For example, noting the symmetry between predicates $\mathsf{C}_{\mathsf{adm\text{-}opt}}$ and $\mathsf{C}_{\mathsf{cgka\text{-}opt}}$, if CGKA* is secure with respect to CGKA predicate $\mathsf{C}_{\mathsf{cgka\text{-}opt}}$, then DGS is secure with respect to admin predicate $\mathsf{C}_{\mathsf{adm\text{-}opt}}$.

We define $\mathsf{C}_{\mathsf{adm}}$ more formally. Let $Q = (q_i)_I$ be the ordered sequence of oracle queries made by the DGS adversary $\mathscr{A}$. To define $\mathsf{C}_{\mathsf{adm}}$, we construct an ordered sequence of queries Q^* that are made by the CGKA* adversary $\mathscr{A}'$ in the security proof below by replacing, inserting and/or deleting queries in-order. Let $\ell \in [1, q_{inj}] \cup \{\perp\}$ where q_{inj} is the number of INJECT queries made by $\mathscr{A}$. To this end, consider each $q_i \in Q$ and, for each ℓ, define q_i^* to be either a single query or a sequence of queries in Q^* as follows:

- $q_i = \mathsf{CREATE}(ID, G, G^*)$: Set $q_i^* = \mathsf{CREATE}(ID, G^*)$.
- $q_i = \mathsf{PROP}(ID, ID', \mathsf{type})$: Set $q_i^* = \perp$ if $\mathsf{type} \neq *\text{-adm}$ and $q_i^* = \mathsf{PROP}(ID, ID', \mathsf{type}^*)$ otherwise where $\mathsf{type} = \mathsf{type}^*\text{-adm}$.
- $q_i = \mathsf{COMMIT}(ID, (i_1, \ldots, i_k), \mathsf{com\text{-}type})$: If the condition $(\mathsf{admReq} \vee \ldots)$ at line 6 of Commit is false, com-type = std or ID is not currently an admin, set $q_i^* = \perp$. Otherwise, let $\{ID_1, \ldots, ID_j\}$ be the (possibly empty) set of parties for which CGKA* rem proposals are introduced by propCleaner (line 12). Let $\vec{P}_A$ be the value input to CGKA*.Commit at line 4 of c-Adm (or $\vec{P}_A = \perp$ if the line is not reached), and $(i_1, \ldots, i_k)$ the corresponding proposal indices in the CGKA* KIND game. Let q' be the REVEAL query that reveals the key k output by Commit in the COMMIT call if the corresponding signature key pair is not used for the first successful INJECT query, and $\perp$ otherwise. Then, set q_i^* to the sequence $(\mathsf{PROP}(ID, ID_1, \mathsf{rem}), \ldots, \mathsf{PROP}(ID, ID_j, \mathsf{rem}), \mathsf{COMMIT}(ID, (i_1, \ldots, i_k)), q')$.
- $q_i = \mathsf{CHAL}(t_s)$: Set $q_i^* = \perp$.
- $q_i = \mathsf{DELIVER}(ID, (t_s, t_a), c)$: Let $(T, \mathsf{com\text{-}type}) = \mathsf{T}[(t_s, t_a), \mathtt{com}, c]$ for DGS KIND game variable T. If Proc is called by the game, $ID \in G$ holds, $T_c \neq \perp$ holds and either $ID \in G^*$ holds or W_A (contained in T) is $\neq \perp$, set $q_i^* = \mathsf{DELIVER}(ID, t_a, c^*)$, where $c^* \leq c$ is the number of times CGKA*.Commit was called after c queries to COMMIT. Otherwise, set $q_i^* = \perp$.
- $q_i = \mathsf{REVEAL}(t_s)$: Set $q_i^* = \perp$.
- $q_i = \mathsf{EXPOSE}(ID)$: Set $q_i^* = q_i$.

- $q_i = \mathsf{INJECT}(ID, m, t_a)$: Set $q_i^* = \perp$ if q_i^* is the j-th query to INJECT where $j \neq \ell$ (possibly $\perp$) and $\mathsf{CHAL}(t_a)$ otherwise ($\ell \neq \perp$).

Then, $\mathsf{C}_{\mathsf{adm}}$ is defined to be true if $\mathsf{C}_{\mathsf{cgka}^*}$ is true for runs where the first successful INJECT query is the ℓ-th query to INJECT (where the $\perp$-th query denotes no successful injection).

We prove DGS secure below.

Proof. Following the proof of Theorem 17 we let G_0 be the $\mathsf{KIND}^{\mathscr{A}}_{\mathsf{A\text{-}CGKA},\mathsf{C}_{\mathsf{cgka}},\mathsf{C}_{\mathsf{adm}},\mathsf{C}_{\mathsf{forgery}}}$ game, and G_1 be as in G_0 except that all calls of the form $(r_1,\ldots,r_i) \leftarrow H_i(r_0,\gamma)$ are replaced with calls of the form $(r_1,\ldots,r_i) \xleftarrow{\$} R^i$. We transition between G_0 and G_1 exactly as in Theorem 17. That is, we define hybrids $G_{0,j}$ where $G_{0,0} = G_0$, $G_{0,j} = G_1$ when $j \geq q$ and $G_{0,j}$ differs from G_0 for appropriate $0 < j < q$ by replacing the first j calls to functions of the form H_i with uniformly sampled values by the challenger. As before, we have $\left|\Pr[G_{0,j-1}(\mathscr{A}) \Rightarrow 1] - \Pr[G_{0,j}(\mathscr{A}) \Rightarrow 1]\right| \leq \epsilon_F$ for each $j \geq 1$, where ϵ_F is the advantage of PRF adversary $\mathscr{A}'$, which implies also that

$$\left|\Pr[G_0(\mathscr{A}) \Rightarrow 1] - \frac{1}{2}\right| \leq \left|\Pr[G_1(\mathscr{A}) \Rightarrow 1] - \frac{1}{2}\right| + q \cdot \mathsf{Adv}^{\mathrm{prf}}_{H_4}(\mathscr{A}').$$

Let E_1 be the event that $\mathscr{A}$ queries INJECT such that the oracle does *not* output $\perp$ (i.e., it outputs the challenge bit). Let E_2 be the event that this does not occur; clearly $\Pr[E_1] + \Pr[E_2] = 1$. To prove security, we will reduce to the security of the primary CGKA, i.e., CGKA, given E_2, and to CGKA^* and signature security given E_1.

We first consider the simpler E_2 case where no successful injection is made (and thus INJECT calls can be easily simulated); let $\mathscr{A}'$ be a KIND adversary w.r.t. CGKA simulating for G_1 adversary $\mathscr{A}$ given E_2. $\mathscr{A}'$ simulates as follows. For each oracle query, $\mathscr{A}'$ simulates relevant CGKA^* calls locally unless stated otherwise. In particular, $\mathscr{A}'$ simulates $\mathsf{Init}(1^\lambda, ID)$ for ID only as needed (i.e., lazily). Then:

- $\mathsf{CREATE}(ID, G, G^*)$: $\mathscr{A}'$ calls $\mathsf{CREATE}(ID, G)$ if needed and otherwise locally simulates.
- $\mathsf{PROP}(ID, ID', \mathsf{type})$: $\mathscr{A}'$ simulates locally if type is of the form $*$-adm and simulates via $\mathsf{PROP}(ID, ID', \mathsf{type})$ otherwise.
- $\mathsf{COMMIT}(ID, (i_1,\ldots,i_k), \mathsf{com\text{-}type})$: Since CGKA.Commit is called after CGKA^*.Commit, $\mathscr{A}'$ can simulate CGKA^* calls locally and call $\mathsf{COMMIT}(ID, J = (j_1,\ldots,j_{k'}))$, where J corresponds to the set of relevant CGKA proposal indices derived from $(i_1,\ldots,i_k)$, to simulate CGKA.Commit calls.
- $\mathsf{DELIVER}(ID, (t_s, t_a), c)$: $\mathscr{A}'$ simulates the relevant CGKA.Proc call (there is at most one such call made by construction of DGS Proc) via $\mathsf{DELIVER}(ID, t_s, c')$ where c' is the index of the relevant CGKA control message. $\mathscr{A}'$ simulates locally otherwise.
- $\mathsf{REVEAL}(t_s)$ and $\mathsf{CHAL}(t_s)$: $\mathscr{A}'$ simulates directly via their respective oracles.

- EXPOSE(*ID*): $\mathscr{A}'$ calls EXPOSE(*ID*) and simulates the rest of the call locally.

- INJECT: By definition of E_2, INJECT always returns $\perp$, and since INJECT does not modify the state, $\mathscr{A}'$ simply outputs $\perp$ upon each INJECT call.

By construction, DGS inherits the (normal) CGKA cleanness predicate $\mathsf{C}_{\mathsf{cgka}}$ from CGKA, and so the simulation is perfect and it follows thus that:

$$\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_2] \leq \mathsf{Adv}^{\mathrm{kind}}_{\mathsf{CGKA}}(\mathscr{A}').$$

Now, consider adversary $\mathscr{A}$ playing G_1 given E_1 occurs, i.e., $\mathscr{A}$ makes a successful INJECT call. Note that there are at most q different CGKA* epochs during a given execution and consequently at most q different CGKA* signature key pairs computed by correct parties (since each signature key pair is derived from a given CGKA* epoch secret).

Given E_1, let F_1 be the event that G_1 adversary $\mathscr{A}$ calls random oracle H_{ro} with input r that corresponds to the signature key pair used in the successful INJECT query (guaranteed to exist by Lemma 16) *before* the injection is made. That is, r is such that $(\mathsf{ssk}, \mathsf{spk}) \leftarrow \mathsf{Gen}(1^\lambda; H_{\mathsf{ro}}(r))$ is called by the challenger at some point. Let F_2 be the complementary event (i.e., either such a H_{ro} query is made after the successful injection or not at all). We consider the case with F_1 (by simulating via the CGKA* KIND game) and F_2 (via the SUF-CMA game) separately.

Consider F_1. Let $F_{i,j}$ be the event that the aforementioned $H_{\mathsf{ro}}(r)$ query is the i-th query to H_{ro} and is with respect to the j-th CGKA* key pair sampled by oracle queries during the game's execution; clearly at most $q \cdot q_{\mathsf{ro}}$ such events occur. Let $\mathscr{A}'$ be a CGKA* adversary who simulates for G_1 adversary $\mathscr{A}$ as follows given $F_{i,j} \wedge E_1$. We assume KIND is such that $\mathscr{A}'$ can reveal secrets k output by Commit from oracle COMMIT. Note that our predicate $\mathsf{C}_{\mathsf{adm}}$ is designed for this simulation.

$\mathscr{A}'$ simulates by calling relevant CGKA* oracles and simulating locally otherwise. When $\mathscr{A}_i$ calls getSigKey while simulating Commit calls, H_{ro} is queried with input r; $\mathscr{A}_i$ lazily samples in this case. When $\mathscr{A}$ queries INJECT, $\mathscr{A}'$ simply returns $\perp$. Note that COMMIT invokes Commit but does not output the key k that Commit outputs. When $\mathscr{A}'$ first derives key k from COMMIT for the x-th CGKA* key pair, for $x \neq j$, $\mathscr{A}$ reveals secret k output by Commit; $\mathscr{A}$ can thus simulate DGS algorithm Commit perfectly. When $\mathscr{A}'$ makes their i-th query to H_{ro} with input r, $\mathscr{A}'$ makes a CHAL query for the corresponding epoch, which outputs k'; $\mathscr{A}_i$ finishes simulating and returns bit 0 if and only if $k' = r$. The simulation is perfect, and when $b = 0$ $\mathscr{A}_i$ wins iff $\mathscr{A}$ wins, since $\mathscr{A}_i$ outputs 0 only if they derive the correct key or signature public key in the simulation, and when $b = 0$ the challenge oracle outputs the correct key. The $b = 1$ case is similar except in the case that $b = 1$ and the r sampled by the game is the same as the real key (which happens with probability $\frac{1}{2^\lambda}$); it follows that

$$\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_1 \wedge F_1] \leq q \cdot q_{\mathsf{ro}} \cdot \Pr[\mathsf{KIND}_{\mathsf{CGKA}^*, \mathsf{C}_{\mathsf{cgka}*}}(\mathscr{A}') \Rightarrow 1] + \frac{q}{2^\lambda}.$$

We consider F_2. Let F'_i be the event, for $1 \leq i \leq q$, that a successful injection is made which uses the i-th CGKA^* key pair sampled by oracle queries during the game's execution. Note that such a key pair must exist and that a signature forgery, by algorithm construction, is a necessary but not sufficient condition to make a INJECT query with a non-bottom response. Consider $\mathscr{A}_i$ who simulates as follows. $\mathscr{A}_i$ simulates all queries locally except that $\mathscr{A}_i$ embeds his challenge key in the i-th such CGKA^* key pair in relevant control messages and uses the SIGN oracle to produce signatures as necessary. Note that the safety predicate is such that, conditioned on F'_i, that an EXPOSE query that leaks the corresponding signature key is disallowed. In addition, $\mathscr{A}$ does not make any H_{ro} query that would lead to a trivial exposure by definition of F_2, and thus the distribution of the challenge key pair is uniform (i.e., correct). It follows that the simulation is perfect and that:

$$\Pr[G_1(\mathscr{A}) \Rightarrow 1 \wedge E_1 \wedge F_2] \leq q \cdot \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{A}')$$

The proof is completed by combining the sequence of game hops considered hitherto. □

5.3.4 Integrating A-CGKA into MLS

Some group messaging protocols already authenticate group members via signatures and public-key infrastructure. The MLS specification [BBR+23] relies on credentials, which are essentially public signature keys for each protocol user that are certified by a PKI; these keys authenticate messages originating from that user[11]. Therefore, it is possible to extend the CGKA used in MLS to an A-CGKA in a more efficient way than using a compiled A-CGKA construction resembling IAS. We note that, in practice, it is feasible to support secure administration in MLS via an *MLS extension*, a feature that enables additional proposal types and actions in the protocol [BBR+23]. Constructing such an extension is almost straightforward and we identify three main necessary changes:

- Credentials are not necessarily refreshed in MLS, meaning that admins (and users in general) whose state is compromised at some point lack forward security and post-compromise security on their authentication keys (unless they proactively update them). Our solution is to enforce an IAS-like credential update mechanism for admin signature keys (providing post-compromise security) which may be invoked without updating the core CGKA secret.
- Group members need to keep track of the administrators (for an IAS-like extension). To this end, we introduce new admin proposal types and enforce that admin proposals are signed, alongside corresponding update policies and modifications to Commit and Proc.
- As in IAS, admins register their keys with the PKI as they are updated over time.

[11] Signatures play an important role in MLS: *"...group members can verify a message originated from a particular member of the group. This is guaranteed by a digital signature on each message from the sender's signature key. The signature keys held by group members are critical to the security of MLS against active attacks..."* [BBR+23].

Modifications. We propose an extension of the main algorithms of the MLS protocol (in particular, of the CGKA-related Prop, Commit and Proc) in Figure 5.12, that we also benchmark in Section 5.4.1. Our goal is to show how an IAS-like protocol can be easily achieved with relatively low overhead. We follow Alwen et al. [ACDT21a] (in particular, Figure 8 in the full version [ACDT21b]), as this is the most comprehensive formalisation of MLS in the literature at the time of writing; therefore, we also work in the single-group setting and omit gids. We omit the Send and Receive algorithms as these are used to send application messages only. We note also that their Create method supports only one initial participant.

Protocol Details. The main modifications are to (1) the admins' credentials, which are regularly updated via upd-adm proposals and admin commits; and (2) in the introduction of the three additional proposal types from A-CGKA. For brevity, we omit several parts of the protocol, such as some sanity checks (e.g., some **require** predicates), functionality that we do not need to modify and details on a higher level than CGKA (like the use of a MAC by MLS). Also for simplicity, we extend the CGKA state γ to include the state variables used in IAS. Following Alwen et al. [ACDT21a], we split the processing algorithms in two – one for commit messages, and one for welcome messages – that a committer produces for each incoming user separately.

Overall, the overhead with respect to (bare-bones) MLS is minimal; we essentially only need to support the new types of proposals and to refresh admin credentials for admin updates. Most of the protocol logic relates to updating signatures and the adminList. Note that proposals are always signed in MLS so signing within makeAdminProp can be foregone. We also support self-removal proposals that can be committed by standard users.

Correctness and Security. We leave it open to formally propose and prove correctness and security for an appropriate MLS extension; we sketch here how it could be done. The modelling of messaging and MLS in particular of Alwen et al. [ACDT21a] is more complex than ours. In particular, they consider CGKA as a sub-primitive that is used to build secure group messaging (SGM) alongside several other primitives. Thus, one could re-define SGM to account for new proposal types and administration as we have done for A-CGKA, including admin correctness guarantees and security upon injections from non-admins. Since admin proposals are tightly-coupled with protocol flow, proposing a model 'on top' of theirs to provide admin security seems difficult, although modular guarantees would be ideal. Proving correctness and security then boils down to similar case analysis and reductions to theirs and ours.

5.4 Evaluation and Discussion

In this section, we describe our implementation and corresponding experimental results for our MLS extension proposed in Section 5.3.4. We then discuss how our modelling in this chapter compares to the related work.

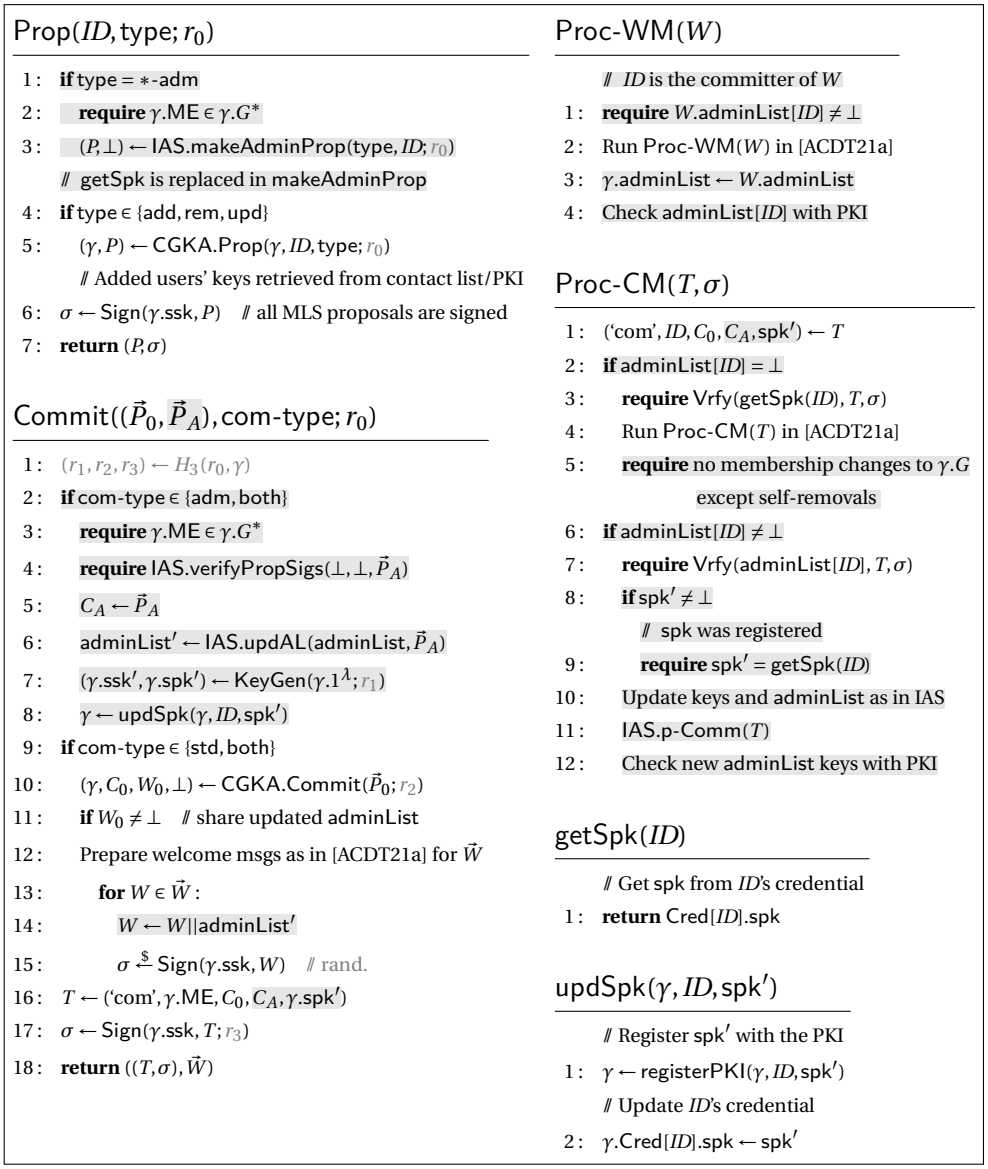

Figure 5.12: Construction of an MLS extension that supports group administrators, effectively turning the CGKA in MLS into an A-CGKA. Highlighted lines correspond to our main modifications in the original secure group messaging (SGM) construction of Alwen et al. [ACDT21a]. Cred[·] denotes a dictionary that stores the credentials of all ID's. We also use the abstract function registerPKI for standard PKI functionality of registering signature keys and getSpk for fetching keys as in IAS. WM (resp. CM) denotes 'welcome message' (resp. 'commit message). Some technical details are omitted.

5.4.1 Benchmarks and Performance

We implemented the protocol in Section 5.3.4 to obtain a realistic estimate of the overhead of securely administrating a real-world messaging protocol. We modified an open-source

implementation of MLS in Go[12] and compare the running times of MLS (which also performs e.g. parent hashing and non-admin proposal signing), with the running times of administrated MLS in different scenarios. In particular, we analyze the Commit and Proc algorithms in Figure 5.12, where the latter includes Proc-CM and also processing proposals when relevant (done separately in the implementation). We ran our benchmarks on a laptop with a 4-core 11th Gen Intel i5-1135G7 processor and 16 GB of RAM using Go's `testing` package[13]. Core cryptographic operations were implemented as HPKE [BBLW22] with ciphersuite `DHKEM(P-256, HKDF-SHA256)`, `HKDF-SHA256`, `AES-128-GCM` (in particular using ECDSA for signatures) from Go standard libraries. We measured the time taken for a *single group member* to perform the relevant operation. For each data point, we took the average over 100 iterations that randomised the group members and admins performing group operations, as performance can be affected by their position in MLS's TreeKEM.

Our results are displayed in Figures 5.13 and 5.14, where we show the running time of the Commit (Figure 5.13) and Proc (Figure 5.14) algorithms in different realistic scenarios. We run experiments where relevant, i.e., when there are no admin operations, using the original implementation as a baseline, as well as using our modified implementation, to demonstrate that the additional admin logic we introduce does not noticeably affect performance.

On the one hand, we present the running time of both algorithms for varying group size $|G|$ with a fixed member/admin ratio $|G|/|G^*| = 4$. In the event of updating users, there are $t = |G^*|$ users updating and/or $t/2$ admins doing an admin-update. On the other hand, we benchmark our algorithms for fixed group size $|G| = 64$, $|G^*| = 16$, while varying the number of updates t (and/or $t/2$ updating admins) in a commit.

For both cases, we compare the committing and processing times of (1) standard commits (com-type = std, omitted in fixed group size benchmarks), (2) standard commits with t update proposals, (3) standard and admin commits (com-type = both) with $t/2$ admin-update proposals but no standard-update proposals, and (4) standard and admin commits with t update and $t/2$ admin-update proposals.

Communication Overhead. In both the baseline and our implementations, proposals used 364 to 366 bytes, and admin proposals used 364 to 368 bytes (all proposals being signed). Commit message sizes in both implementations vary proportionally with the size of the group and the number of proposals. In the *baseline MLS implementation*, a typical Commit for $|G| = 8$ and $|G| = 128$ with $t = 2$ and $t = 32$ update proposals uses 1.49 KB and 17.11 KB respectively. In *our implementation*, corresponding commits use 1.56 KB and 17.17 KB respectively. If $t/2$ admin updates are added (1 and 16, respectively), commits require 1.60 KB and 17.65 KB. In general, commits in our implementation, even with admin proposals, incur only a small amount of overhead (tens of bytes) over the baseline implementation when fixing the number of proposals.

[12] The original source code is available at https://github.com/cisco/go-mls.
[13] https://pkg.go.dev/testing

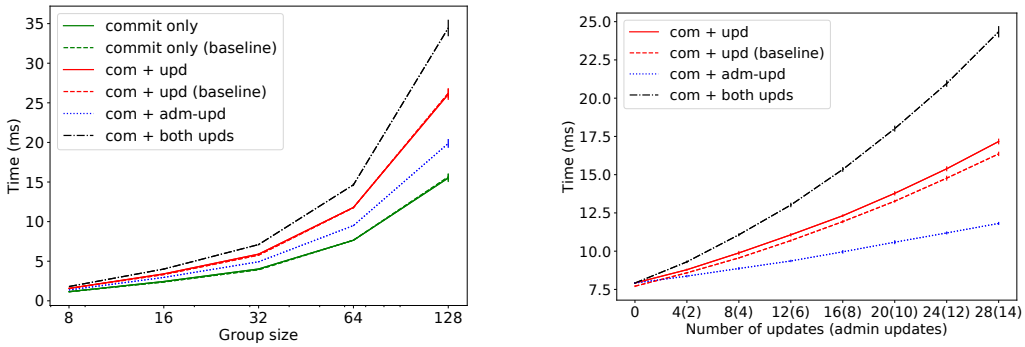

Figure 5.13: Benchmark of the Commit algorithm in the following scenarios: (1) standard commits (com-type = std, omitted right), (2) standard commits with t update proposals, (3) standard and admin commits (com-type = both) with $t/2$ admin-update proposals but no standard-update proposals, and (4) standard and admin commits with t update and $t/2$ admin-update proposals. Original MLS is displayed as baseline. *Left:* running time with respect to group size $|G|$ on constant member/admin $|G|/|G^*| = 4$ ratio and constant number of updates $t = |G^*|$ ($t/2$ admin updates). *Right:* running time with respect to the number of updating users t (and $t/2$ admin updates), for fixed $|G| = 64$.

Protocols. The results above show that the additional cost (for users) of running a securely-administrated MLS is minimal. Figure 5.13 shows that the Commit algorithm involves less than a 20% overhead when up to $|G|/8$ members carry out admin updates simultaneously (note that admin updates also involve standard updates). Figure 5.14 shows that the processing time of admin and standard updates is very similar, and increases linearly in the number of updates.

Separately, we analyze the overhead of IAS and DGS for group members, both for number of operations and for message size. We note that this assumes that IAS and DGS are implemented modularly and not integrated with an existing CGKA as before. In IAS, admins generate a signature key pair and sign every time they carry out a commit or a proposal, and verify a small amount t of signatures (typically $t \leq |G^*|$) in admin proposals before a commit. If we denote the cost (time/length) of a message signature or verification by s, and the cost (time/length) of a signature key pair generation by k, we obtain the values[14] in Table 5.1. Note that $s = \mathcal{O}(\lambda)$ and $k = \mathcal{O}(\lambda)$ (i.e., are constant) for security parameter λ.

The overhead of DGS depends heavily on the cost of the admin CGKA CGKA^* (an optimistic estimation can be $\mathcal{O}(\log m)$ [ACDT20, KPPW+21, ACJM20] but can be $\mathcal{O}(m)$ in the worst case). CGKA operations only affect administrators. Note that a DGS admin-only commit is not sent to standard members (only the signed new admin key has to). Hence, DGS is very efficient for standard users.

[14] We ignore the size of user identifiers *ID*s as we assume they are small and are each a constant or even independent function of the security parameter. In any case, the overhead from identifiers should be similar in existing application-level protocols.

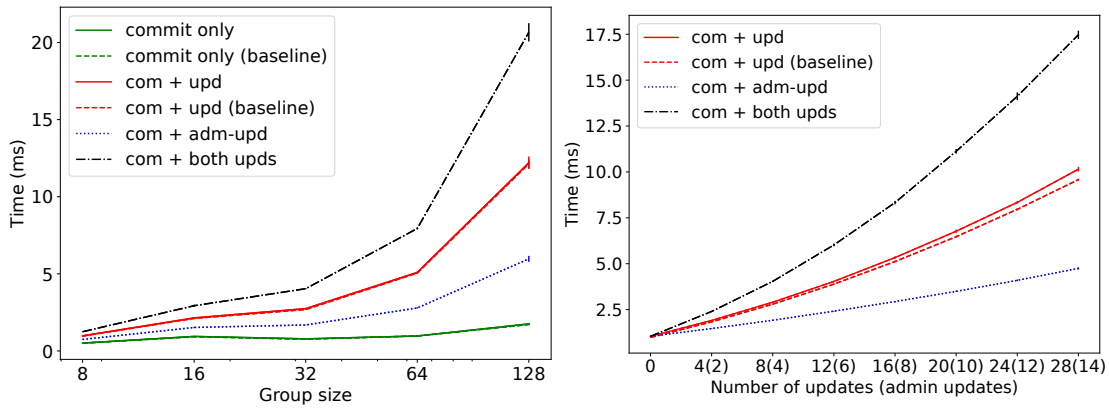

Figure 5.14: Benchmark of the Proc algorithm when processing a commit message. The different scenarios are those of Figure 5.13.

	Length (Adm)	Length (All)	Time (Adm)	Time (All)
IAS	$ts+tk$	$ts+tk$	$\mathcal{O}(ts+k)$	$\mathcal{O}(ts)$
DGS	$C+s+k$	$s+k$	$\mathcal{O}(C+s+k)$	$\mathcal{O}(s+k)$

Table 5.1: Additional cost of IAS and DGS with respect to a plain CGKA (per group) where t is the number of admin proposals and C (for DGS only) refers to the cost of running CGKA^*.

The ratio of additional messages sent, which is application-specific, is hard to estimate. Admin-only commits and admin modifications are expected to be less frequent than standard operations. The number of update proposals (although individually very cheap) is expected to be at most linear in $|G|$.

Forward-secure signatures (for optimally-secure IAS) can be instantiated with essentially constant amortised overhead in space and time relative to a regular signature scheme, while supporting unbounded secret key updates [MMM02].

We conclude that IAS presents a generally affordable overhead for all users, while DGS introduces basically no cost for standard users and is more costly for administrators if $|G^*|$ is relatively large.

Admins in TreeKEM Variants. The Tainted TreeKEM protocol provides efficiency advantages if only a subset of users carry out tree-changing operations (adds and removals) [KPPW+21]. Tainted TreeKEM, however, is not formalised in the propose-and-commit-paradigm, which complicates the efficiency comparison; such an analysis thus remains open. When standard users are allowed to commit updates, the tree blanking issue with MLS TreeKEM is not worsened by administrators, hence efficiency should not decrease either.

5.4.2 Modelling in Related Work

CGKAs and MLS. The CGKA abstraction has deviated from MLS and has become an object of study of its own [AHKM22, BDG+22, AAN+22b], but in general still inherits important limitations from MLS. Among them, CGKAs rely on the availability of a well-behaved PKI, generally require total ordering on control messages (recent work relaxes this to causal ordering [AMT23]), and fail to capture messaging solutions that deviate from group key agreement such as Signal and WhatsApp.

In MLS, there exists a strong architectural separation between the Delivery Service (DS, usually a central server) and a so-called Authentication Service (AS) whose design is left to the infrastructure designers [BBR+23]. Following this separation, the Delivery Service is often modelled adversarially in CGKAs whereas the AS is abstracted as a PKI [ACDT20, KPPW+21, ACJM20] as in this chapter. A compromised AS allows for the corruption of user credentials, resulting in trivial user impersonation.

PKI. Both IAS and DGS rely on a PKI that we assume is incorruptible. In IAS, parties use the PKI to verify the identity of administrators and self-removing users. In DGS, incoming users use it to retrieve the current group-wide admin signature public key. As the PKI is only used to establish an initial root of trust among parties, i.e., forward and post-compromise security are ensured for existing group members without additional PKI calls, our modelling is consistent with the separation between delivery and authentication discussed above. Note that group administration aims to remove the trust in the DS (the server) but is still vulnerable to a corrupt AS. Previous CGKA work follows similar PKI abstractions [ACDT20, KPPW+21], or ignores the AS [CHK21]. That is, in all group messaging works we are aware of, the PKI *always* behaves consistently and correctly for all users. Two partial exceptions are Alwen et al. [AJM22] and Alwen et al. [ACDT21a], where malicious keys can be registered, although security (inevitably) degrades strongly for such users. By abstracting away the AS, our schemes are compatible with diverse authentication solutions such as out-of-band verification.

Signal Private Groups. In Chase et al. [CPZ20] and as deployed in Signal, a central server manages the membership of a group whilst hiding the set of group members from non-members (modulo metadata leaked to a network adversary). The main goals of this solution are to achieve user privacy and act as a single source of truth for the membership of a group. We believe that this approach could be extended to support secure administration; an advantage is that users no longer have to track group membership individually as in (A)-CGKA, which prevents consistency issues when users do not apply the same sequence of group updates locally. We note that Signal Private Groups however does not fully protect from server and network attacks as our A-CGKA constructions do: for example, it is possible for the server to re-add removed users. In addition, the system has not been analysed in composition with an underlying group messaging protocol (pairwise Signal) where concurrency issues can arise.

6 WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs

In this chapter, we formally model and analyse the group messaging protocol Sender Keys used by WhatsApp and Signal, and propose some extensions and improvements. An extended abstract corresponding to this work appeared at ASIACRYPT 2023, and was joint work with David Balbás and Phillip Gajland [BCG23b]. A preliminary version of this work appeared at RECSI 2022 [BCG22]. A full version of this work can be found on the Cryptology ePrint Archive [BCG23a].

6.1 Contribution

In the context of group messaging, Signal [M+16][1] and later WhatsApp [Wha20] have adopted the so-called *Sender Keys* protocol [Mar14], which has enjoyed widespread adoption for numerous years. Besides these, other popular solutions such as Matrix [ADJ24] and Session [Jef20] implement variants of this protocol. In Sender Keys, messages are encrypted using a user-specific symmetric key (which is then hashed forward) and then authenticated with a signature. Additionally, parties rely on secure two-party channels (instantiated in practice with the Double Ratchet) to share key material between them. Looking ahead, two-party channels will be central to determine the security attained by any instantiation of Sender Keys.

As mentioned in Chapter 5, a baseline for secure group messaging has been recently established by the IETF Messaging Layer Security (MLS) [BBR+23] standard, a joint effort between academia and industry[2]. The protocol provides sub-linear complexity for group operations (adding/removing members and updating key material). Academic works have also explored so-called continuous group key agreement (CGKA) [BBR18, ACDT20, KPPW+21, ACJM20, ACDT21a], although these are only a component of a fully-fledged group messaging protocol. So far, in terms of complete messaging protocols, only the modular construction

[1] Contrary to the folklore understanding that the Signal Messenger uses the pairwise channels approach for group messaging in small groups, Signal currently uses Sender Keys whenever possible. We refer to Section 6.5.4 for details.

[2] Recent academic works and ongoing discussions in mailing lists have identified and addressed several security issues that emerged during the standardisation of MLS [ACDT20, AJM22, IETF23].

from [ACDT21a] building on CGKA (which includes MLS), DCGKA [WKHB21] in the decentralised setting and very recently Matrix [ADJ24] have been formalised to date.

Despite being the most complete and well-studied protocol to-date in the literature, MLS and CGKAs in general still have some drawbacks. While some exhibit sub-linear performance in specific executions (and this class of executions is not well-characterised in the literature), their performance can degrade to linear in general, which is unavoidable at least when using off-the-shelf cryptographic primitives [BDG+22]. Moreover, they tend to be complex, increasing their attack surface and making them more susceptible to design and implementation bugs. Finally, given the standardisation of MLS only occurred recently, MLS is yet to be widely deployed.

Hence, Sender Keys and similar approaches to group messaging remain an essential and practical alternative with different security / performance trade-offs. Firstly, Sender Keys stands out for its relative simplicity, which reduces its potential attack surface, making the protocol less susceptible to vulnerabilities in both its design and implementation. Secondly, Sender Keys offers good performance in small to moderate-sized groups, as demonstrated by its successful adoption for groups of up to 1024 parties in WhatsApp and Signal [Wha20, M+16]. While the main group operations (adding and removing users) respectively have $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ total communication complexity for groups of size n, concrete efficiency suffices in practice. Thirdly, Sender Keys offers forward-secure confidentiality and robust support for concurrent and out-of-order application message exchange.

Surprisingly, despite having the widest adoption and an open source implementation of its core cryptographic operations [M+16], Sender Keys has not been formally studied in the literature, prompting the following natural question:

Can we formalise the Sender Keys protocol in a meaningful security model?

To answer this question we start by introducing a new cryptographic primitive, along with a security model, to capture a broad class of group messaging protocols that do not necessarily employ CGKA [ACDT20] at their core. Our framework provides native support for group messaging protocols that utilise secure two-party communication channels under the hood, for which we introduce a clean level of abstraction. This novel framework proves instrumental in our analysis, as existing literature predominately focuses on CGKA-oriented models that do not suit Sender Keys and similar protocols.

Subsequently, we present a detailed description of the core Sender Keys protocol within our framework and provide a security proof validating the soundness of the protocol. In our analysis, we observe that Sender Keys presents several deficiencies that, despite not being easily exploitable flaws, prevent several desirable and fundamental security notions from being met. These include forward security under message injections, resilience against injections impacting group membership changes[3], and fast recovery from state compromise. These

[3]Note that Signal uses a dedicated private group management solution in practice [CPZ20] that we do not

findings call into question the widespread use of the term "secure messaging" by commercial messaging solutions, motivating the need for more detailed discussion about the nuances around these protocols.

In this regard we propose an improved version of Sender Keys, that we call Sender Keys+, where we only employ readily available cryptographic primitives that have minimal impact on efficiency[4]. This addresses the following pertinent question:

How can we improve the security of Sender Keys whilst preserving its practical efficiency?

Overall, we believe that the formalisation and establishment of a provably secure variant of Sender Keys, such as the Sender Keys+ protocol proposed in this chapter, can serve as a valuable foundation for future implementations of the protocol.

6.1.1 Summary

In summary, the main scientific contributions of this chapter are the following:

- We introduce a new cryptographic primitive that we call *Group Messenger* (GM). We establish a modular security model for GM designed to capture messaging protocols like Sender Keys that are not necessarily based on group key agreement. It accounts for an active adversary capable of controlling the network and adaptively learning the secret states of different parties.
- We develop a general framework for composing two-party channels with group messaging protocols that use them. Our approach parameterises the security of the Group Messenger primitive based on the underlying two-party channels, presenting a novel perspective that, to the best of our knowledge, has not been explored previously.
- We formally describe Sender Keys, based on an analysis of Signal's source code [M+16], WhatsApp's security white paper [Wha20], and the `yowsup` library [Gal21].
- We prove the security of Sender Keys in our model and describe several shortcomings. These force us to restrict the capabilities of the adversary substantially for the proof to be carried out.
- We propose security fixes and improvements, several of which result in the improved protocol Sender Keys+. In particular, we secure group membership changes, improve the forward security of the protocol, and introduce an efficient key update mechanism. We also formalise the additional security guarantees in our model.

capture and is less affected by this attack vector than WhatsApp [RMS18]; we refer to Section 6.5.4 for further details.

[4] Our approach veers away from a theoretically systematic exploration to determine the "optimal" security for a Sender Keys-like protocol, as this would require non-standard primitives that considerably degrade performance [BRV20, ACJM20].

6.1.2 Technical Overview

Security in Group Messaging. Besides standard notions such as confidentiality, authenticity, and integrity of sent messages, we again consider forward security (FS) and post-compromise security in this chapter. Additionally, protocols must secure group membership updates, namely removed members must not be able to read messages sent after their removal, and newly added members must not (by default) be able to read past messages.

Most of the different formalisations of security in the literature model an adversarial Delivery Service (DS), the entity responsible for delivering messages between parties over the network. The adversary (modelling the DS) can act as an eavesdropper with extended capabilities, e.g., that can schedule messages to be consistently delivered by users, as in Alwen et al. [ACDT20], as a semi-active adversary that can schedule messages arbitrarily [KPPW+21], or as an active adversary that can inject messages to different degrees [ACJM20, BCV23]. In many protocols, including Sender Keys and MLS, the DS relies mainly on some centralised infrastructure (the *central server* hereafter).

Sender Keys. In a Sender Keys group $\boldsymbol{G}$, every user $ID \in \boldsymbol{G}$ owns a so-called *sender key* which is shared with all group members. A sender key is a tuple $\mathsf{SK} = (\mathsf{spk}, \mathsf{ck})$, where spk is a public signature key (with a private counterpart ssk), and ck is a symmetric *chain key*. Every time ID sends a message m to the group, ID encrypts m using a *message key* mk that is deterministically derived (via a key derivation function H_1) from its chain key ck and erased immediately after being used. Upon message reception, group members derive mk to decrypt the corresponding ciphertext, which can also be delivered out-of-order as we discuss in later sections. Messages are authenticated by appending the sender's signature to them. In Figure 6.1, we provide a high-level depiction of what takes place in a three-member group $\boldsymbol{G} = \{A, B, C\}$ when A sends a message that parties B and C receive.

Informally, forward security is provided by using a fresh message key for every message: every time a message is sent, the chain key is symmetrically ratcheted, i.e., hashed forward using a key derivation function H_2. The protocol, that we describe further in Section 6.4, also requires that there exist confidential and authenticated communication channels between each pair of group members. These are used for sharing sender keys when parties are added or removed from the group, or when some party updates their key material. For example, in the event that some ID leaves the group, members erase their own sender key and start over. This mechanism provides a form of PCS when a user is removed as the key material is refreshed.

Modelling Two-Party Channels. Formally capturing the security of two-party channels is central to our analysis of Sender Keys since fresh sender keys are sent over these channels. Two-party channels that are not regularly used can undermine security. For example, if a group member ID's state is compromised, there is no guarantee that fresh keys sent by other members (via two-party channels) are not leaked, since ID's two-party channels may not yet have healed yet. Moreover, two-party channels can take more than one round trip to heal when using the Double Ratchet, as is the case for WhatsApp and the Signal Messenger [ACD19].

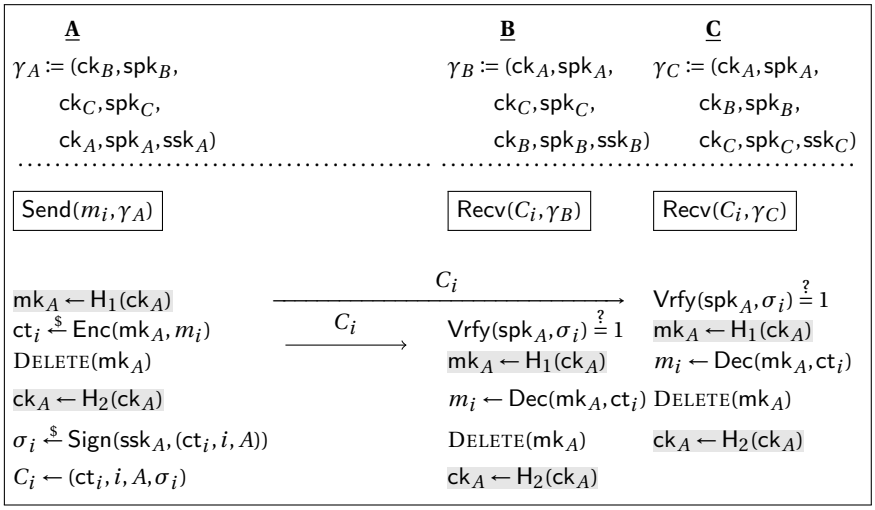

Figure 6.1: Simplified diagram for sending/receiving messages between three group members. For $ID \in \{A, B, C\}$, ID's initial sender key is $(\mathsf{ck}_{ID}, \mathsf{spk}_{ID})$. The state γ_{ID} of ID contains the sender keys of all group members.

Our modelling starts in Section 6.2 with the introduction of a primitive 2PC for two-party channels. We define a two-party channel with initialisation (Init), channel initialisation (InitCh), send (Send) and receive (Recv) algorithms. Notably, InitCh allows parties to adaptively bootstrap channels, and deviates from works on ratcheting-based two-party messaging that abstract authentication away [BSJ+17]. Our security model captures both forward security and post-compromise security. To model PCS, we introduce a crucial parameter, denoted as Δ and referred to as the *PCS bound*. This parameter, inspired by Alwen et al. [ACD19] and Blazy et al. [BBL+22], serves as an upper bound on the number of synchronous communication steps or *channel epochs* required to restore security following a compromise.

Our Primitive: Group Messenger. In Section 6.3, we define a new cryptographic primitive, Group Messenger (GM), which includes five stateful algorithms that: initialise a party's state (Init), send an application message (Send), receive an application message (Recv), execute a change proposal in the group (Exec), and process a change in the group (Proc). Supported group changes are: group creation, member addition, member removal, and sender key updates. Note this contrasts with the three-phase propose/commit/process flow for updates (the so-called propose-commit paradigm [ACJM20]) used by MLS and newer CGKA protocols.

We define a game-based security notion for GM that captures a partially active adversary with control over the Delivery Service, taking inspiration from previous CGKA modelling [ACJM20, BCV23]. In our model in Section 6.3.1, we capture the security of each protocol by parameterising the game with a *cleanness predicate* (sometimes safety predicate in other work), which excludes trivial attacks, including both those which are unavoidable and those indicative of security weaknesses.

Security Analysis of Sender Keys. With this formalism established, in Section 6.4.2 we define cleanness predicates for Sender Keys that precisely capture its security. We define three sub-predicates that restrict the capabilities of the adversary for message *challenges*, capturing confidentiality; for message *injections*, capturing integrity and authenticity; and for re-orderings and forgeries of control messages (*concurrency*), capturing the message ordering provided by the central server.

Notably, the restrictions that we impose via our cleanness predicates are necessary for the security proof to go through and reveal several shortcomings in the protocol. Examples include:

- Sender Keys achieves only a weak form of PCS through key updates. Healing from a compromise requires multiple messages (at least $\Delta + 1$), even if a user is removed during healing.
- Control messages lack proper authentication and are malleable. An adversary with partial control over the network, such as the server, can make arbitrary changes in the group membership (such as adding new users without any member's authorisation), which is a significant practical concern.
- Forward security is sub-optimal, as messages are malleable after they are sent if a state exposure occurs.

Towards proving Sender Keys secure, we provide a somewhat detailed description of Sender Keys for pedagogical reasons in Section 6.4.1 and a complete formalisation in Section 6.6.1; for reference we also provide two tables in Appendix A.4 explaining variables used in our security games and formal specification. These may be of independent interest, as in particular we provide the first complete formal specification of Sender Keys. In Section 6.6.2, we formally prove that Sender Keys is secure under standard assumptions in our (necessarily restricted) model. Since we treat the primitives (symmetric encryption, signatures, PRG, two party channels) as black boxes, one can build, e.g., post-quantum Sender Keys following our specification.

In Section 6.5.4, we compare our description of Sender Keys with the implementations in WhatsApp and Signal, clarifying the extent to which our findings are applicable to these popular apps. We remark that our core analysis is nevertheless implementation-agnostic, and the fact that we model the underlying two-party channels in a fine-grained fashion allows us to capture their impact on security of Sender Keys in the face of state exposure (i.e., FS and PCS).

Shortcomings and Proposed Improvements. Leaving aside the security limitations that are intrinsic to the design of the protocol, we find that one can improve both its security and efficiency in several aspects. Hence, in Section 6.5 we propose modifications to the protocol with the aim of securing group membership, strengthening the (weaker than expected) forward security for authentication, and integrating efficient post-compromise security updates. Notably,

our novel PCS update mechanism improves the key update mechanism implemented by our core protocol and performed in Signal, bringing down the total communication complexity from quadratic to linear in the group size. Moreover, as a result of our modular approach with respect to modelling two-party channels, our modelling can capture the security improvement (or weakening) that results from replacing the Double Ratchet by an alternative two-party messaging protocol. Note that our formal of Sender Keys (Section 6.6.1) also contains our modifications for Sender Keys+ (they are nonetheless visually separated from the core protocol).

We extend our modelling to establish the security of our modified protocol, called Sender Keys+. The main technical step involves redefining the cleanness predicates (Section 6.5), which are strictly less restrictive compared to those used for the original protocol. Notably, the adversary is now allowed to inject control messages (given the group has recovered from any state exposures). We also allow the adversary to mount more fine-grained attacks for application message forgeries, and allow *arbitrary* challenges after some party has updated over a refreshed channel (before, we could only re-allow challenges on the *updater*). In Section 6.6.3 we formally prove security additionally assuming a secure MAC and dual PRF.

6.1.3 Additional Related Work

The formal extension of CGKA to group messaging was explored by Alwen et al. [ACDT21a], while the key schedule of MLS was proven secure by Brzuska et al. [BCK22]. We provide more thorough comparison between CGKA-based protocols and Sender Keys/Sender Keys+ in Section 6.5.3.

The work of Cong et al. [CEST22] shares some similarities with ours as it also constructs group messaging from two-party channels and achieves $\mathcal{O}(n)$ key update complexity. However, they do not model two-party channels as a standalone primitive nor dynamic groups formally, and their protocols require more interaction than ours (e.g., the initial group key agreement protocol can take several rounds in general).

Concurrency, a crucial aspect in CGKA-based protocols, has been a central topic in works such as [AAN+22b, AAN+22a, BDR20]. Secure administration in CGKAs was explored in the previous chapter (Chapter 5) of this thesis. Weidner et al. [WKHB21] adopt a Sender Keys-like approach to construct a decentralised CGKA protocol but they do not capture group messaging, and their security model does not support message injections (hence considering a passive adversary). Moreover, the theorems in their work assume a non-adaptive adversary where the adversary must announce all queries at the game's outset. Their work nonetheless extends the scope of modern messaging to decentralised networks without a central authority, diverging from existing approaches that target centralised networks. A simplified (notably lacking forward security) decentralised variant of Sender Keys is implemented by the Session app [Jef20].

Also relevant to our work are secure two-party messaging protocols that propose alternatives to the Double Ratchet [MP16a], such as [JS18, PR18, DV19, ACD19, BRV20, PP22]. Inspired by more practical-oriented endeavors, we acknowledge recent cryptographic audits conducted on Telegram [AMPS22], Matrix [ACDJ23], Threema [KGP23], and WhatsApp's backup service [DFG+23].

Sender Keys. While some works on Sender Keys lack formalism and security proofs, they offer valuable insights. Rösler et al. [RMS18] evaluate Sender Keys, provide a high-level description of the protocol, and examine practical vulnerabilities in WhatsApp group chats. Multi-group security and key update mechanisms for Sender Keys are informally discussed by Cremers et al [CHK21]. In [BCG22], a preliminary analysis of the security of Sender Keys is carried out. While the paper only includes informal discussions and no proofs, it serves as an initial exploration for the ideas in the present work. We remark that the scope of [BCG22] is limited, as it does not formally develop a security model, and assumes that all two-party channels used by Sender Keys are *perfectly secure*, which is unrealistic and impossible to develop in practice.

Concurrent work by Albrecht et al. [ADJ24] develops a device-oriented security model and a proof for a recent specification of Matrix (i.e., for the updated protocol that mitigates the issues described in [ACDJ23]). For group messaging, Matrix implements the Megolm protocol, which is Sender Keys-inspired but still deviates significantly from our description in this chapter, particularly regading server interaction. Remarkably, [ADJ24] and our work arrive to similar conclusions in our analysis, such as the insecurity of group management and the challenges imposed by message ordering. Our works are complimentary and open new research directions. Examples include exploring whether the improvements behind Sender Keys+ can also be applied to Megolm, as well as extending our modelling to consider the (challenging) multi-device setting as they do.

6.2 Two-Party Channels

In this section, we present our approach towards capturing two-party channels as a standalone primitive for modelling Sender Keys.

6.2.1 Primitive Definition and Correctness

We start by defining two-party channels.

Definition 44 (Two-Party Channel)**.** A two-party channel scheme is a tuple of efficient algorithms $\mathsf{2PC} := (\mathsf{Init}, \mathsf{InitCh}, \mathsf{Send}, \mathsf{Recv})$ such that:

- $\gamma \xleftarrow{\$} \mathsf{Init}(1^\lambda, ID)$: The probabilistic initialisation algorithm takes a user identity ID as input and outputs an initial state γ.
- $\mathsf{acc} \xleftarrow{\$} \mathsf{InitCh}(\gamma, ID^*)$: The probabilistic channel initialisation algorithm takes a state γ

and a user identity ID^* as inputs and outputs an acceptance bit $\mathsf{acc} \in \{\mathsf{true}, \mathsf{false}\}$ and updates the caller's state.

- $(C, e_{2\mathsf{pc}}, i_{2\mathsf{pc}}) \xleftarrow{\$} \mathsf{Send}(\gamma, m, ID^*)$: The probabilistic sending algorithm takes as inputs a message m, the intended message recipient ID^* and a state γ, and outputs a ciphertext C, a channel epoch-index pair $(e_{2\mathsf{pc}}, i_{2\mathsf{pc}})$ corresponding to m (or $\perp$ upon failure), and updates the state.

- $(m, ID^*, e_{2\mathsf{pc}}, i_{2\mathsf{pc}}) \leftarrow \mathsf{Recv}(\gamma, C)$: The deterministic receiving algorithm takes as inputs a ciphertext C and a state γ, and outputs a message m, a user identity ID^* corresponding to the sender of m and a channel epoch-index pair $(e_{2\mathsf{pc}}, i_{2\mathsf{pc}})$ corresponding to m (or $\perp$ upon failure), and updates the state.

Our $\mathsf{2PC} := (\mathsf{Init}, \mathsf{InitCh}, \mathsf{Send}, \mathsf{Recv})$ primitive captures two initialisation functions. The first function initialises the state of a party by taking its ID as input, while the second function is used to initialise a communication channel with a counterpart ID^*. Consider two parties, ID and ID^* who intend to communicate over a two-party channel. Both parties initialise their states, γ_{ID} and γ_{ID^*} using the Init function. Subsequently, ID and ID^* initiates the communication channel by invoking $\mathsf{InitCh}(\gamma_{ID}, ID^*)$ and $\mathsf{InitCh}(\gamma_{ID^*}, ID)$ (note we assume both parties must call InitCh).[5] It is worth noting that, similar to DCGKA [WKHB21], our two-party channel primitive implicitly depends on some universally available and consistent public-key infrastructure.

Channel Epochs and Indices. Our notion of channel epochs is exactly the notion of epochs as defined in [ACD19] used to model the Double Ratchet protocol [MP16a] and is an example of an *ordinal* as in Section 4.2. For a given two-party channel, ID and ID' are each associated with a channel epoch $e_{2\mathsf{pc}}$, which corresponds to how many times the direction of communication has changed, alongside an index $i_{2\mathsf{pc}}$, indicating how many Send calls have been made by a sender or the latest message received by a receiver. Initially, the sender (say ID) sets $e_{2\mathsf{pc}} = 0$ and the receiver ID' sets $e_{2\mathsf{pc}} = -1$. Thereafter, party ID (resp. ID') is the sender in even (resp. odd) channel epochs, and the receiver in odd (resp. even) channel epochs. When a party is a sender in channel epoch $e_{2\mathsf{pc}}$ and receives a message from $e_{2\mathsf{pc}} + 1$, they advance to $e_{2\mathsf{pc}} + 1$; likewise they advance channel epochs when they are a receiver and then send a message. When a party sends as a sender, they increment their index $i_{2\mathsf{pc}}$, and as a receiver they set $i_{2\mathsf{pc}}$ to the message received with the highest index. Note that $i_{2\mathsf{pc}}$ represents the number of messages sent or received for a given channel epoch $e_{2\mathsf{pc}}$; $i_{2\mathsf{pc}}$ is set to 0 whenever a party has just become a sender again. Hence, we define a total ordering on channel epochs and indices $(e_{2\mathsf{pc}}, i_{2\mathsf{pc}})$ such that $(e_{2\mathsf{pc}}, i_{2\mathsf{pc}}) \leq (e'_{2\mathsf{pc}}, i'_{2\mathsf{pc}})$ when $e_{2\mathsf{pc}} < e'_{2\mathsf{pc}}$, or $e_{2\mathsf{pc}} = e'_{2\mathsf{pc}}$ and $i_{2\mathsf{pc}} \leq i'_{2\mathsf{pc}}$.

Oracles for Correctness. To capture correctness we employ several oracles that the adversary can query. We describe them briefly before describing the correctness notion.

[5] In practice, a party given pair of parties may initialise and tear down many channels between each other over time; by associating a given 'party' with several identifiers, our modelling here is without loss of generality.

INIT-CH(ID, ID')**:** This oracle initialises the two-party channel between parties ID and ID'.

SEND(ID, ID', m)**:** This oracle allows the sending of a message m (i.e., generating a ciphertext) from party ID to party ID'.

RECEIVE$(ID, ID', e_{2pc}, i_{2pc})$**:** This oracle delivers the ciphertext output by ID' for channel epoch (e_{2pc}, i_{2pc}) in a previous SEND call to ID.

EXPOSE(ID)**:** This oracle leaks the state of party ID to the adversary.

Game $\text{2PC-CORR}^{\mathscr{A}}_{\text{2PC}}$

1 : **for all** ID :
2 : $\quad \gamma[ID] \xleftarrow{\$} \text{Init}(1^\lambda, ID)$
3 : $\mathsf{M}[\cdot], \mathsf{CT}[\cdot] \leftarrow \bot$
4 : $\mathsf{CH} \leftarrow \emptyset$
5 : $\text{win} \leftarrow 0$
6 : $\mathscr{A}^{\mathcal{O}}$
7 : **return** win

Oracle INIT-CH(ID, ID')

1 : **require** $(ID, ID') \notin \mathsf{CH}$
2 : $\text{acc} \xleftarrow{\$} \text{InitCh}(ID', \gamma[ID])$
3 : **if** $\neg$acc
4 : $\quad \text{win} \leftarrow 1$
5 : $\mathsf{CH} \xleftarrow{\cup} (ID, ID')$
6 : **return** acc

Oracle EXPOSE(ID)

1 : **return** $\gamma[ID]$

Oracle SEND(ID, ID', m)

1 : **require** $(ID, ID') \in \mathsf{CH}$
2 : $(C, e_{2pc}, i_{2pc}) \xleftarrow{\$} \text{Send}(m, ID', \gamma[ID])$
3 : **if** $C = \bot \vee e_{2pc} = \bot \vee i_{2pc} = \bot$
4 : $\quad \text{win} \leftarrow 1$
5 : $\mathsf{M}[ID, ID', e_{2pc}, i_{2pc}] \leftarrow m$
6 : $\mathsf{CT}[ID, ID', e_{2pc}, i_{2pc}] \leftarrow C$
7 : **return** $(\text{win}, C, e_{2pc}, i_{2pc})$

Oracle RECEIVE$(ID, ID', e_{2pc}, i_{2pc})$

1 : **require** $(ID', ID) \in \mathsf{CH}$
2 : **require** $\mathsf{CT}[ID, ID', e_{2pc}, i_{2pc}] \neq \bot$
3 : $C \leftarrow \mathsf{CT}[ID, ID', e_{2pc}, i_{2pc}]$
4 : $(m, ID', e'_{2pc}, i'_{2pc}) \leftarrow \text{Recv}(C, \gamma[ID])$
5 : $m' \leftarrow \mathsf{M}[ID, ID', e_{2pc}, i_{2pc}]$
6 : **if** $(m', e'_{2pc}, i'_{2pc}) \neq (m, e_{2pc}, i_{2pc})$
7 : $\quad \text{win} \leftarrow 1$
8 : **return** win

Figure 6.2: 2PC-CORR correctness for 2PC where $\mathcal{O} = \{\text{INIT-CH}, \text{EXPOSE}, \text{SEND}, \text{RECEIVE}\}$. Lines in teal correspond only to bookkeeping and state update operations.

Correctness. We provide a correctness game played between a challenger and a computationally unbounded adversary (Figure 6.2). here, The adversary can invoke the INIT-CH oracle to establish a secure channel between ID and ID' complete; we require that $\text{InitCh}(ID, ID')$ is always successful when first called with input (ID, ID'). Subsequently, the adversary can dynamically query the SEND and RECEIVE oracles to execute the protocol. Note that the EXPOSE oracle is unrestricted, providing the adversary with leaked state information. It is worth noting that RECEIVE only allows the adversary to make the challenger call Receive on inputs that were output by the corresponding Send call in SEND.

We formally define correctness below.

Definition 45 (Correctness of 2PC)**.** We consider the 2PC-CORR$_{\mathsf{2PC}}$ game defined in Figure 6.2. A two-party channels scheme 2PC is correct or 2PC-CORR if for any (possibly unbounded) adversary $\mathscr{A}$ we have

$$\mathsf{Adv}^{\mathsf{2pc-corr}}_{\mathsf{2PC}}(\mathscr{A}) := \Pr[\mathsf{2PC\text{-}CORR}^{\mathscr{A}}_{\mathsf{2PC}} \Rightarrow 1] = 0\,.$$

Security. The Double Ratchet protocol has been the subject of several academic works [ACD19, CCD+20, CJSV22, BFG+22a] that analyse its security on a fine-grained level for two-party communication. When used by multiple parties in a group during the execution of Sender Keys, analysing the Double Ratchet protocol becomes complex, making it difficult to replace it with other protocols. To tame this complexity, we adopt a comparatively simpler notion of two-party communication in similar complexity to the formalism of Weidner et al. for their DCGKA protocol [WKHB21]. Our modelling captures forward security, is parameterised by the post-compromise security of the underlying channels and supports out-of-order message delivery. We define security after describing the security game and introducing some predicates below.

Oracles for Security. Compared to correctness, the adversary $\mathscr{A}$ will be given access to the following oracles in the security game:

INIT-CH(ID, ID')**,** SEND(ID, ID', m)**,** EXPOSE(ID)**:** As in correctness.

CHAL(ID, ID', m_0, m_1)**:** This oracle generates a message challenge where the adversary $\mathscr{A}$ provides two messages m_0 and m_1 of the same length, and party ID sends m_b to party ID'.

RECEIVE(ID, C)**:** This oracle delivers the ciphertext C to party ID.

Game Description. The notion of security defined in the 2PC-IND$_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ game in Figure 6.3 is parameterised by a cleanness predicate $\mathsf{C}_{\mathsf{2pc}}$ and a PCS bound $\Delta > 0$. Broadly, the cleanness predicate prevents the adversary from winning the game by making a trivial attack, i.e., via a bit guess (resp. forgery) based on a challenge (resp. delivery) using exposed key material. PCS after an exposure is parameterised by Δ, which is the number of message steps (i.e. number of times that the sender-receiver roles alternate) that the channel requires for healing.

The game starts by initialising the states of all parties and the dictionaries CH and M, which store challenge ciphertexts and all sent (including challenge) ciphertexts, respectively. It also initialises EI as a variable that tracks the channel epoch-index pair of a given channel (ID, ID'). Then, the adversary $\mathscr{A}$ can adaptively query all the oracles listed above. Finally, $\mathscr{A}$ outputs a guess b' of b that the challenger also outputs given that the execution has been *clean*, i.e., that the cleanness predicate $\mathsf{C}_{\mathsf{2pc}}$ holds.

$\mathscr{A}$ can win the game in two different ways. Firstly, it can make a correct guess of the bit b; note

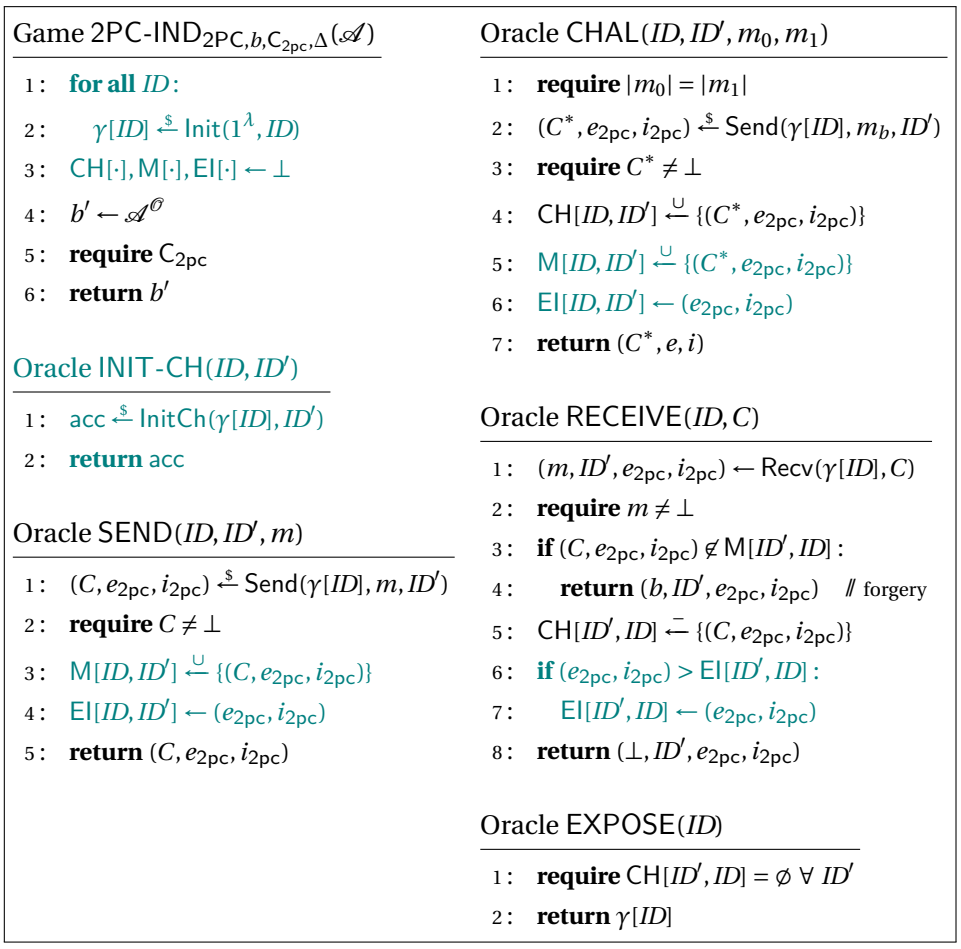

Figure 6.3: Indistinguishability (2PC-IND) security for 2PC where $\mathscr{O} = \{\mathsf{INIT\text{-}CH}, \mathsf{SEND}, \mathsf{CHAL}, \mathsf{RECEIVE}, \mathsf{EXPOSE}\}$. Lines in teal correspond only to bookkeeping and state update operations. Dictionaries CH, M and EI store challenged messages, sent messages, and channel epoch-index respectively.

(ignoring RECEIVE) the only operation that depends on b is the CHAL oracle. Ciphertexts C^* generated by $\mathsf{CHAL}(ID, ID', m_0, m_1)$ are stored in CH, and removed from CH once they are delivered. We use CH to prevent trivial forgeries, as if there is any challenge C^* that has not yet been delivered to ID' (and such that by correctness ID' must be able to decrypt it), then $\mathscr{A}$ cannot leak the state of ID' via $\mathsf{EXPOSE}(ID')$. Secondly, $\mathscr{A}$ can directly obtain b by making a successful forgery that is accepted via $\mathsf{RECEIVE}(ID, C)$. To leak b to $\mathscr{A}$, RECEIVE checks that C does not correspond to a message sent by the sender ID' in epoch (e_{2pc}, i_{2pc}), where (ID', e_{2pc}, i_{2pc}) are the outputs of the challenger's Recv call within the RECEIVE query.

We note that the challenge-and-send style of our game is analogous to the security game for GM (to be introduced). This game is also multi-user as it captures all channels at once, extending other single-user security models such as those in [ACD19, WKHB21]. Our modelling presents some similarities with the modelling of two-party channels in [WKHB21], but differs

from it in several aspects. It is also multi-challenge as CHAL can be called several times. Our Init algorithm does not require the public key of the counterpart as opposed to theirs (we capture this via InitCh for each channel), and correctness is captured as part of their security model. More importantly, their adversary is not allowed to attempt forgeries (the model only captures confidentiality of sent messages) or out of order delivery.

Predicates. The game is parametrised by the two-party channels cleanness predicate $\mathsf{C}_{\mathsf{2pc}}$, which we divide into two sub-predicates as $\mathsf{C}_{\mathsf{2pc}} := \mathsf{C}_{\mathsf{2pc\text{-}chal}} \wedge \mathsf{C}_{\mathsf{2pc\text{-}inj}}$. Both sub-predicates capture PCS and are additionally parametrised by the PCS bound Δ. We follow the blueprint of [ACD19] for the predicate definition.

$$
\begin{array}{l}
\boxed{\mathsf{C}_{\mathsf{2pc\text{-}chal}}:}\ \forall (i, ID, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) : q_i = \mathsf{EXPOSE}(ID) \wedge \\
\quad (e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) = \max\{\mathsf{EI}[ID, ID'; q_i], \mathsf{EI}[ID', ID; q_i]\}, \nexists\, (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}, j) : (i < j) \wedge \\
\quad [\ (q_j = \mathsf{CHAL}(ID', ID, \cdot, \cdot) \wedge (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}) = \mathsf{EI}[ID', ID; q_j] \wedge e'_{\mathsf{2pc}} < e_{\mathsf{2pc}} + \Delta) \vee \\
\quad (q_j = \mathsf{CHAL}(ID, ID', \cdot, \cdot) \wedge (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}) = \mathsf{EI}[ID, ID'; q_j] \wedge e'_{\mathsf{2pc}} < e_{\mathsf{2pc}} + \Delta)\ ]
\end{array}
$$

Figure 6.4: Predicate $\mathsf{C}_{\mathsf{2pc\text{-}chal}}$ where $\mathscr{A}$ makes oracle queries $q_1, \ldots, q_q$.

Challenge (Figure 6.4). Suppose that $\mathscr{A}$ exposes a party ID and later makes a CHAL query involving ID (either as a sender or as a receiver) and some other party ID'. Essentially, the predicate requires that the challenge message then must belong to a channel epoch e'_{2pc} that is Δ or more epochs past the channel epoch e_{2pc}, where e_{2pc} corresponds to the largest epoch value between ID and ID' at exposure time.

Injection (Figure 6.5). For an intuitive description, let C be a forged ciphertext that, when processed by Recv by some ID', claims to be from sender ID and from epoch-index $(e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}})$. Then, the predicate requires that if $\mathscr{A}$ exposes ID and later attempts to inject C to some ID', both the epoch e'_{2pc} of the (ID, ID') channel at injection time and e^*_{2pc} are Δ or more epochs further from the channel epoch e_{2pc} at exposure time. Considering both epochs $e_{\mathsf{2pc}}, e^*_{\mathsf{2pc}}$ prevents injections on out-of-order messages. Self-injections (i.e., where $ID = ID'$) are also restricted since the adversary can trivially mount such an attack on the Double Ratchet [ACD19].

$$
\begin{array}{l}
\boxed{\mathsf{C}_{\mathsf{2pc\text{-}inj}}:}\ \forall (i, ID, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) : q_i = \mathsf{EXPOSE}(ID) \wedge \\
\quad (e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) = \max\{\mathsf{EI}[ID, ID'; q_i], \mathsf{EI}[ID', ID; q_i]\}, \\
\quad \nexists\, (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}, e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}, j, C) : (i < j) \wedge (\min\{e^*_{\mathsf{2pc}}, e'_{\mathsf{2pc}}\} < e_{\mathsf{2pc}} + \Delta) \wedge \\
\quad (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}) = \min\{\mathsf{EI}[ID, ID'; q_j], \mathsf{EI}[ID', ID; q_j]\} \wedge \\
\quad [\ (q_j = \mathsf{RECEIVE}(ID', C) = (\cdot, ID, e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}) \wedge (C, e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}) \notin \mathsf{M}[ID, ID'; q_j]) \vee \\
\quad (q_j = \mathsf{RECEIVE}(ID, C) = (\cdot, ID', e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}) \wedge (C, e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}) \notin \mathsf{M}[ID', ID; q_j])\ ]
\end{array}
$$

Figure 6.5: Predicate $\mathsf{C}_{\mathsf{2pc\text{-}inj}}$ where $\mathscr{A}$ makes oracle queries $q_1, \ldots, q_q$.

In order to model relevant injection predicates for our GM primitive such as $\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj}}$ (Fig-

ure 6.11), we abuse notation and refer directly to $\mathsf{C}_{\mathsf{2pc\text{-}inj}}$. In the context of GM, we parametrise the predicate by a ciphertext C_{2pc}, and simply replace the belonging to M by the equivalent condition in the GM security game $(\cdot, C_{\mathsf{2pc}}) \notin \mathsf{M}[ID, ID'; q_j]$.

We formally define security below.

Definition 46 (Security of 2PC)**.** We consider the $\mathsf{2PC\text{-}IND}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}$ game defined in Figure 6.3. A two-party channels scheme 2PC satisfies indistinguishability or 2PC-IND security with respect to cleanness predicate $\mathsf{C}_{\mathsf{2pc}}$ and PCS bound $\Delta > 0$ if for any efficient adversary $\mathscr{A}$ we have

$$\begin{aligned}\mathsf{Adv}^{\mathsf{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{A}) &:= \left|\Pr[\mathsf{2PC\text{-}IND}_{\mathsf{2PC},1,\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{A}) \Rightarrow 1] - \Pr[\mathsf{2PC\text{-}IND}_{\mathsf{2PC},0,\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{A}) \Rightarrow 1]\right| \\ &= \mathsf{negl}\end{aligned}$$

Instantiations. By previous work [ACD19], the Double Ratchet can be seen to achieve a PCS bound of $\Delta = 3$. However, by replacing the Diffie Hellman key exchange component in the Double Ratchet (referred to as continuous key agreement [ACD19]) with a KEM, the PCS bound can be improved to $\Delta = 2$. This is optimal since if a user is exposed in channel epoch e_{2pc} and acts as the sender, then they can decrypt a message from channel epoch $e_{\mathsf{2pc}} + 1$ based on correctness requirements. While we do consider protocols with weak PCS and hence larger values of Δ, including $\Delta = \infty$ if new randomness is never injected in key derivation, protocols lacking forward security like TLS are considered insecure within our model.

For channel initialisation InitCh, an initial key exchange between the parties needs to be carried out. Typically this is done via the X3DH protocol [MP16b] and by relying on a PKI which we abstract away in this chapter but previously explored in Chapter 3.

Extensions. Our security model adopts the core requirements of the modelling in [ACD19]. Our cleanness predicates are designed to suit a large class of two-party messaging protocols parametrised by the PCS bound Δ. As a consequence, our analysis is not as fine-grained as possible, but we gain in readability and modularity.

The recent work of Blazy et al. [BBL+22] devises metrics (generalising our PCS bound in some sense) to classify different two-party messaging protocols based on their resilience to different adversarial behaviours and the resulting PCS guarantees. Future work could incorporate these factors into our modelling of two-party channels. Another direction would be to parametrise the PCS bound based on whether a party is exposed while acting as a sender or receiver on the channel (as opposed to considering a worst-case Δ). One could take an even more fine-grained approach in the style of [CCD+20] also. A bound of $\Delta = 1$ would be possible when only considering receiver exposure.

We also note that if one replaces the Double Ratchet with another protocol, either keeping [ACD19, CZ22, PP22] or dropping [JS18, PR18, DV19] support for out-of-order message delivery along the way, it is possible to consider less restrictive injection predicates. For in-

stance, by using signatures, one no longer needs to restrict self-injections. For simplicity, we also restrict injections corresponding to out-of-order delivery (all epochs $< e_{2\mathsf{pc}} + \Delta$ in $\mathsf{C}_{\mathsf{2pc\text{-}inj}}$); notice that restricting winning injections only on message epochs corresponding to old ciphertexts in-transit at corruption time is sufficient as done in [ACD19].

6.3 Group Messenger

We introduce our main cryptographic primitive called *Group Messenger* that captures sending and receiving application messages between members of a dynamic group. We note that our primitive captures a single group; extending it to multiple groups is straightforward by using group identifiers (see e.g. [BCV23]).

Definition 47 (Group Messenger)**.** A Group Messenger GM is a tuple of efficient algorithms $\mathsf{GM} := (\mathsf{Init}, \mathsf{Send}, \mathsf{Recv}, \mathsf{Exec}, \mathsf{Proc})$ such that:

- $\gamma \xleftarrow{\$} \mathsf{Init}(ID)$: The probabilistic initialisation algorithm takes as input a user identity ID and outputs an initial state γ.
- $C \xleftarrow{\$} \mathsf{Send}(\gamma, m)$: The probabilistic sending algorithm takes as inputs a message m and a state γ, and outputs a ciphertext C (or $\perp$ upon failure) and updates the state.
- $(m, ID^*, e, i) \leftarrow \mathsf{Recv}(\gamma, C)$: The deterministic receiving algorithm takes as inputs a ciphertext C and a state γ, and outputs a message m, an identity ID^* corresponding to the sender, a group epoch e and index i both corresponding to m (or $\perp$ upon failure), and updates the state.
- $T \xleftarrow{\$} \mathsf{Exec}(\gamma, \mathtt{cmd}, IDs)$: The probabilistic execution algorithm takes as inputs a command $\mathtt{cmd} \in \{\mathtt{crt}, \mathtt{add}, \mathtt{rem}, \mathtt{upd}\}$, a list of identities IDs and a state γ, and outputs a control message T (or $\perp$ upon failure) and updates the state.
- $b \leftarrow \mathsf{Proc}(\gamma, T)$: The deterministic processing algorithm takes as inputs a control message T and a state γ, and outputs an acceptance bit $b \in \{0,1\}$ and updates the state.

Finally, the (possibly empty) set of group members is stored as $\gamma.\boldsymbol{G}$

In our syntax, a distinction is made between *application* messages and *control* messages. Specifically, distinct algorithms are employed for the transmission and reception of application messages, as well as for the execution and processing of group modifications. These modifications, executed via Exec, are parameterised by a command `cmd` that encompasses operations including in this chapter user addition `add`, removal `rem`, group creation `crt`, or user key material update `upd`. Moreover, in scenarios where two-party messaging protocols are required for the group primitive (although not applicable to CGKAs such as TreeKEM [ACDT20]), two-party messages are formally assumed to be sent alongside or within ciphertexts or control

messages. Consequently, they are abstracted away from our syntax. Looking ahead, we will model the fact that ciphertexts and control messages are sent alongside two-party channel ciphertexts (and can be received with a different ciphertext or control message) when we define security in Section 6.3.1.

Message Epochs. We define a message epoch as a pair of integers (e, i), internal to the state γ of a party *ID*, that captures time and synchronisation between parties. Message epochs are central to our description of Sender Keys and security model. Each application message sent by *ID* corresponds to a single message epoch (e, i), which is output by the Recv algorithm at the receiver's end. The *epoch* e advances whenever *ID* processes a new group change (i.e., a control message). The *index* i advances when *ID* sends a new message. If control messages are delivered to group members in lockstep (i.e. sequentially in the same order), parties who have the same epoch e will have the same view of the group membership. We define a total ordering $(e, i) \leq (e', i')$ when $e < e'$, or $e = e'$ and $i \leq i'$. Nevertheless, we remark that 2PC channel epochs are independent from GM message epochs.

Oracles for Correctness. We introduce a game-based notion for GM correctness (as well as for security later), where the adversary $\mathscr{A}$ will have access to various oracles that we outline below.

CREATE(*ID*, *IDs*)**:** creates a group by executing $\mathsf{Exec}(\gamma, \texttt{crt}, IDs)$ with *ID* as the initiator, generating a control message T. Namely, $\mathscr{A}$ obtains $C_b \leftarrow \mathsf{Send}(\gamma[ID], m_b)$.

SEND(*ID*, *m*)**:** *ID* sends an application-level message m using the Send algorithm, producing a ciphertext C.

RECEIVE(*ID*, *ID*′ *e*, *i*)**:** *ID* receives the ciphertext sent by *ID*′ corresponding to message epoch (e, i) by calling $\mathsf{Recv}(\gamma[ID], C)$, where C is stored in $\mathsf{CT}[ID', e, i]$ by the challenger (note we only consider honest delivery for correctness).

ADD(*ID*, *ID*′) / REMOVE(*ID*, *ID*′) / UPDATE(*ID*)**:** *ID* adds *ID*′ / removes *ID*′ / refreshes *ID*'s secrets by calling $\mathsf{Exec}(\gamma[ID], \texttt{add}, ID')$ / $\mathsf{Exec}(\gamma[ID], \texttt{rem}, ID')$ / $\mathsf{Exec}(\gamma[ID], \texttt{upd}, ID)$, generating control message T. The e-th control message (i.e., the epoch e control message) is stored as $\mathsf{T}[i]$.

DELIVER(*ID*)**:** *ID* is delivered the next control message in T, if it exists, via $\mathsf{Proc}(\gamma[ID], \mathsf{T}[\mathsf{E}[ID]])$ (where *ID* is in epoch $\mathsf{E}[ID]$).

EXPOSE(*ID*)**:** Leaks the state γ of *ID* to $\mathscr{A}$.

Correctness. We capture correctness in a game played by an unbounded adversary in Figure 6.6. Our game captures several properties, given that all messages are generated honestly:

- *Message delivery*: Application messages (generated by SEND) must be received correctly by all group members (checked at lines 9-10 of RECEIVE).

Game M-CORR$_{GM}(\mathscr{A})$

1: **for all** ID:
2: $\quad \gamma[ID] \xleftarrow{\$} \mathsf{Init}(1^\lambda, ID)$
3: $\mathsf{T}[\cdot], \mathsf{M}[\cdot], \mathsf{CT}[\cdot] \leftarrow \bot$
4: $\mathsf{ep}, \mathsf{E}[\cdot], \mathsf{I}[\cdot] \leftarrow 0$
5: $\mathsf{G}[\cdot] \leftarrow \emptyset$
6: $\mathsf{win} \leftarrow 0$
7: $\mathscr{A}^{\mathcal{O}}$
8: **return** win

Oracle CREATE(ID, IDs)

1: **require** $ID \in IDs$
2: **require** $\mathsf{ep} = 0$
3: $T \xleftarrow{\$} \mathsf{Exec}(\gamma[ID], \texttt{crt}, IDs)$
4: $\mathsf{T}[\mathsf{ep}] \leftarrow T$; $\mathsf{ep} \leftarrow \mathsf{ep} + 1$
5: $\mathsf{G}[0], \mathsf{G}[1] \leftarrow IDs$
6: **return** T

Oracle SEND(ID, m)

1: **require** $ID \in \mathsf{G}[\mathsf{E}[ID]]$
2: $C \xleftarrow{\$} \mathsf{Send}(\gamma[ID], m)$
3: **if** $C = \bot$
4: $\quad \mathsf{win} \leftarrow 1$
5: $\mathsf{I}[ID] \leftarrow \mathsf{I}[ID] + 1$
6: $\mathsf{M}[ID, \mathsf{E}[ID], \mathsf{I}[ID]] \leftarrow m$
7: $\mathsf{CT}[ID, \mathsf{E}[ID], \mathsf{I}[ID]] \leftarrow C$
8: **return** C

Oracle ADD(ID, ID')

1: **require** $\mathsf{E}[ID] = \mathsf{ep}$
2: $T \xleftarrow{\$} \mathsf{Exec}(\gamma[ID], \texttt{add}, \{ID'\})$
3: **require** $T \neq \bot$
4: $\mathsf{T}[\mathsf{ep}] \leftarrow T$; $\mathsf{ep} \leftarrow \mathsf{ep} + 1$
5: $\mathsf{G}[\mathsf{ep}] \leftarrow \mathsf{G}[\mathsf{ep} - 1] \cup \{ID'\}$
6: **return** T

Oracle REMOVE(ID, ID')

1: **require** $\mathsf{E}[ID] = \mathsf{ep}$
2: $T \xleftarrow{\$} \mathsf{Exec}(\gamma[ID], \texttt{rem}, \{ID'\})$
3: **require** $T \neq \bot$
4: $\mathsf{T}[\mathsf{ep}] \leftarrow T$; $\mathsf{ep} \leftarrow \mathsf{ep} + 1$
5: $\mathsf{G}[\mathsf{ep}] \leftarrow \mathsf{G}[\mathsf{ep} - 1] \setminus \{ID'\}$
6: **return** T

Oracle UPDATE(ID)

1: **require** $\mathsf{E}[ID] = \mathsf{ep}$
2: $T \xleftarrow{\$} \mathsf{Exec}(\gamma[ID], \texttt{upd}, \{ID\})$
3: **require** $T \neq \bot$
4: $\mathsf{T}[\mathsf{ep}] \leftarrow T$; $\mathsf{ep} \leftarrow \mathsf{ep} + 1$
5: $\mathsf{G}[\mathsf{ep}] \leftarrow \mathsf{G}[\mathsf{ep} - 1]$
6: **return** T

Oracle EXPOSE(ID)

1: **return** $\gamma[ID]$

Oracle RECEIVE(ID, ID', e, i)

1: **require** $ID \in \mathsf{G}[e]$
2: **require** $\mathsf{CT}[ID', e, i] \neq \bot$
3: $C \leftarrow \mathsf{CT}[ID', e, i]$
4: (m', ID', e', i')
$\quad \leftarrow \mathsf{Recv}(\gamma[ID], C)$
5: $m \leftarrow \mathsf{M}[ID', e, i]$
6: **if** $\mathsf{E}[ID] < e$:
7: $\quad$ **if** $m' \neq \bot$
8: $\quad\quad \mathsf{win} \leftarrow 1$
9: **else if** $(m', e', i') \neq (m, e, i)$
10: $\quad \mathsf{win} \leftarrow 1$
11: **return** win

Oracle DELIVER(ID)

1: **require** $\mathsf{E}[ID] < \mathsf{ep}$
2: **require** $ID \in \mathsf{G}[\mathsf{E}[ID]]$
3: $\mathsf{acc} \leftarrow \mathsf{Proc}(\gamma[ID], \mathsf{T}[\mathsf{E}[ID]])$
4: **if** $\neg\mathsf{acc}$
5: $\quad \mathsf{win} \leftarrow 1$
6: $\mathsf{E}[ID] \leftarrow \mathsf{E}[ID] + 1$
7: **if** $ID \in \mathsf{G}[\mathsf{E}[ID]]$
8: $\quad$ **if** $\mathsf{G}[\mathsf{E}[ID]] \neq \gamma[ID].\boldsymbol{G}$
9: $\quad\quad \mathsf{win} \leftarrow 1$
10: $\mathsf{I}[ID] \leftarrow 0$
11: **return** win

Figure 6.6: Game defining M-CORR$_{GM}$ with adversary $\mathscr{A}$ for $\mathcal{O}$ = {CREATE, SEND, ADD, REMOVE, UPDATE, EXPOSE, RECEIVE, DELIVER}. Lines in teal correspond only to bookkeeping and state update operations.

- *Group evolution*: Group operations, namely `crt` (group creation), `add` (user addition), `rem` (user removal), and `upd` (key update), must have their intended effects on the group when received and processed (checked at lines 4-5 and 7-9 of DELIVER).

- *Group membership consistency*: The list of group members must be consistent among all group members, assuming they process the same sequence of control messages (checked at lines 7-9 of DELIVER)

- *Out-of-order delivery*: Messages corresponding to past epochs must be decryptable if

delivered out-of-order (checked at lines 9-10 of RECEIVE). Messages corresponding to future epochs must be rejected upon reception (checked at lines 6-8 of RECEIVE)

We define correctness below.

Definition 48 (Correctness of GM)**.** A Group Messenger $\mathsf{GM} := (\mathsf{Init}, \mathsf{Send}, \mathsf{Recv}, \mathsf{Exec}, \mathsf{Proc})$ is M-CORR if, for all (possibly unbounded) adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\mathrm{m-corr}}_{\mathsf{GM}}(\mathscr{A}) := \Pr[\mathsf{M\text{-}CORR}_{\mathsf{GM}}(\mathscr{A}) \Rightarrow 1] = 0\,,$$

where game $\mathsf{M\text{-}CORR}_{\mathsf{GM}}$ is defined in Figure 6.6.

6.3.1 Security Model

We introduce a game-based model of security for our Group Messenger primitive that captures the main desirable security properties of a group messaging scheme. In brief, our game $\mathsf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}$ captures a partially active adversary who can, in particular, expose the state of users and inject (possibly malformed) messages at any time. We consider the confidentiality of application messages with FS and PCS, and we also model the out-of-order delivery of application and control messages.

Oracles for Security. Compared to correctness, the adversary $\mathscr{A}$ will be given access to the following oracles in the security game:

$\mathsf{CREATE}(ID, IDs)$**,** $\mathsf{SEND}(ID, m)$**,** $\mathsf{ADD}(ID, ID')$**,** $\mathsf{REMOVE}(ID, ID')$**,** $\mathsf{UPDATE}(ID)$**,** $\mathsf{EXPOSE}(ID)$**:** As in correctness.

$\mathsf{CHAL}(ID, m_0, m_1)$**:** outputs a ciphertext C_b corresponding to the message m_b sent by ID, where b is the bit that parametrises the game. Namely, $\mathscr{A}$ obtains $C_b \leftarrow \mathsf{Send}(\gamma[ID], m_b)$.

$\mathsf{RECEIVE}(ID, C)$**:** ID receives a ciphertext C by calling $\mathsf{Recv}(\gamma[ID], C)$. The sender ID' is inferred from output of Recv. In the event of a successful forgery, $\mathscr{A}$ obtains the value of b (and can thus return b to immediately win the game).

$\mathsf{DELIVER}(ID, T)$**:** ID is delivered a control message T via $\mathsf{Proc}(\gamma[ID], T)$.

Game Description. We introduce the $\mathsf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}$ game in Figure 6.7, parameterised by bit b that has to be guessed by $\mathscr{A}$ as in a message indistinguishability game. The adversary wins the game if it directly guesses b correctly, which it can always do if it carries out a successful forgery (since it is given b directly in this case). The game is further parameterised by a protocol-specific *cleanness* predicate C_{gm} (sometimes safety predicate [ACDT20]) that rules out trivial attacks and captures the exact security of the protocol.

Game M-IND$_{\mathsf{GM},b,\mathsf{C}_{\mathsf{gm}}}(\mathscr{A})$

1: **for all** ID:
2: $\quad\gamma[ID] \xleftarrow{\$} \mathsf{Init}(1^\lambda, ID)$
3: $\mathsf{T}[\cdot], \mathsf{M}[\cdot], \mathsf{CH}[\cdot], \mathsf{SM}[\cdot] \leftarrow \bot$
4: $\mathsf{ep} \leftarrow 0$
5: $\mathsf{E}[\cdot], \mathsf{I}[\cdot] \leftarrow 0$
6: $b' \leftarrow \mathscr{A}^{\mathscr{O}}$
7: **require** C_{gm}
8: **return** b'

Oracle CREATE(ID, IDs)

1: **require** $ID \in IDs$
2: **require** $\mathsf{ep} = 0$
3: $T \xleftarrow{\$} \mathsf{Exec}(\gamma[ID], \texttt{crt}, IDs)$
4: $\mathsf{T}[\mathsf{ep}] \leftarrow T;\ \mathsf{ep} \leftarrow \mathsf{ep}+1$
5: **return** T

Oracle CHAL(ID, m_0, m_1)

1: **require** $|m_0| = |m_1|$
$\quad$// Send m_b based on b
2: $C^* \xleftarrow{\$} \mathsf{Send}(\gamma[ID], m_b)$
3: **require** $C^* \neq \bot$
4: $\mathsf{I}[ID] \leftarrow \mathsf{I}[ID]+1$
5: $\mathsf{CH}[ID, \mathsf{E}[ID], \mathsf{I}[ID]] \leftarrow C^*$
6: **return** C^*

Oracle ADD(ID, ID')

1: $T \xleftarrow{\$} \mathsf{Exec}(\gamma[ID], \texttt{add}, \{ID'\})$
2: **require** $T \neq \bot$
3: $\mathsf{T}[\mathsf{ep}] \leftarrow T;\ \mathsf{ep} \leftarrow \mathsf{ep}+1$
4: **return** T

Oracle REMOVE(ID, ID')

1: $T \xleftarrow{\$} \mathsf{Exec}(\gamma[ID], \texttt{rem}, \{ID'\})$
2: **require** $T \neq \bot$
3: $\mathsf{T}[\mathsf{ep}] \leftarrow T;\ \mathsf{ep} \leftarrow \mathsf{ep}+1$
4: **return** T

Oracle UPDATE(ID)

1: $T \xleftarrow{\$} \mathsf{Exec}(\gamma[ID], \texttt{upd}, \{ID\})$
2: **require** $T \neq \bot$
3: $\mathsf{T}[\mathsf{ep}] \leftarrow T;\ \mathsf{ep} \leftarrow \mathsf{ep}+1$
4: **return** T

Oracle SEND(ID, m)

1: $C \xleftarrow{\$} \mathsf{Send}(\gamma[ID], m)$
2: **require** $C \neq \bot$
3: $\mathsf{I}[ID] \leftarrow \mathsf{I}[ID]+1$
4: $\mathsf{M}[ID, \mathsf{E}[ID], \mathsf{I}[ID]] \leftarrow C$
5: **return** C

Oracle EXPOSE(ID)

1: **return** $\gamma[ID]$

Oracle RECEIVE(ID, C)

1: (m, ID', e, i)
$\quad\leftarrow \mathsf{Recv}(\gamma[ID], C)$
2: **if** $m \neq \bot$:
3: $\quad\mathsf{Update\ SM}[ID, ID']$
4: $\quad$**if** recv-forgery(C):
5: $\quad\quad$**return** b
6: **return**

Oracle DELIVER(ID, T)

1: $\mathsf{acc} \leftarrow \mathsf{Proc}(\gamma[ID], T)$
2: **require** acc
3: $\mathsf{E}[ID] \leftarrow \mathsf{E}[ID]+1$
4: $\mathsf{I}[ID] \leftarrow 0$
5: **if** proc-forgery(T):
6: $\quad$**return** b
7: **return**

Figure 6.7: Game defining M-IND$_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}$ with adversary $\mathscr{A}$ and cleanness predicate C_{gm} for $\mathscr{O} = \{\mathsf{CREATE}, \mathsf{CHAL}, \mathsf{ADD}, \mathsf{REMOVE}, \mathsf{UPDATE}, \mathsf{SEND}, \mathsf{EXPOSE}, \mathsf{RECEIVE}, \mathsf{DELIVER}\}$. Lines in teal correspond only to bookkeeping and state update operations.

Message Epochs. We define a function $\mathsf{m\text{-}ep}(ID, ID', q)$ that indicates the highest message epoch (e, i), as output by the Recv algorithm, for which a user ID has received a message from ID' at the time of query q for $ID \neq ID'$. For $ID = ID'$, this indicates the local state value for $(\mathsf{E}[ID], \mathsf{I}[ID])$. The m-ep function reflects the view of user ID' by user ID.

Dictionaries. The challenger keeps a record of messages and game variables in several dictionaries. The state of each party is stored in $\gamma[\cdot]$ and updated when an algorithm is called on a given $\gamma[ID]$. Ciphertexts and challenged ciphertexts are stored in M and CH, respectively, each of them indexed by an ID and a message epoch (e, i). The unique honest control message that starts a given epoch e is stored in $\mathsf{T}[e]$, and the most recent epoch of the group is stored in variable ep (note that we implicitly assume a total ordering of control messages). The current

message epoch of ID is stored in $\mathsf{E}[ID], \mathsf{I}[ID]$. Even if each control message in T corresponds to a single epoch, different parties can be in different epochs. We say ID is in epoch e before query q if the last control message processed by ID before query q is $\mathsf{T}[e]$.

The message epochs corresponding to skipped messages from sender ID' stored by ID are kept in $\mathsf{SM}[ID, ID']$. We keep SM updated in the RECEIVE oracle as follows: given a message epoch (e, i) and an ID' output by Recv, if $(e, i) \in \mathsf{SM}[ID, ID']$ then (e, i) is erased from $\mathsf{SM}[ID, ID']$. Otherwise, we add all pairs (e', i') such that $(e', i') < (e, i)$ and (e', i') corresponds to all messages sent by ID' not delivered to ID.

Outcome. After q oracle queries, $\mathscr{A}$ outputs a guess b' of b if the cleanness predicate C_{gm} is satisfied (otherwise the game aborts). $\mathscr{A}$ can win the game in three different ways: by directly guessing the challenge bit correctly, by injecting a forged application message via RECEIVE successfully, or by injecting a forged control message via DELIVER. The cleanness predicate C_{gm} parameterises the security of a given protocol by restricting the capabilities of the adversary to exclude a class of attacks. Additionally, we explicitly state predicates recv-forgery and proc-forgery in our game, which model the conditions under which a RECEIVE or DELIVER call result in a successful forgery (leaking b to $\mathscr{A}$). We expand on these predicates in Section 6.3.2.

We define security below.

Definition 49 (Message indistinguishability of GM)**.** A Group Messenger $\mathsf{GM} := (\mathsf{Init}, \mathsf{Send}, \mathsf{Recv}, \mathsf{Exec}, \mathsf{Proc})$ is M-IND with respect to cleanness predicate C_{gm} if, for all efficient adversaries $\mathscr{A}$, we have

$$\mathsf{Adv}^{\mathrm{m-ind}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) := \left|\Pr[\mathsf{M\text{-}IND}^{\mathscr{A}}_{\mathsf{GM},1,\mathsf{C}_{\mathsf{gm}}} \Rightarrow 1] - \Pr[\mathsf{M\text{-}IND}^{\mathscr{A}}_{\mathsf{GM},0,\mathsf{C}_{\mathsf{gm}}} \Rightarrow 1]\right| = \mathsf{negl}.$$

where game $\mathsf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}$ is defined in Figure 6.7.

Related Security Notions. Our security model takes inspiration from the game-based modelling developed for MLS and for CGKA (which we introduced formally in Section 5.2). Nevertheless, it is not ideal to adopt these models as they consider a single group key, which is not directly compatible with a Sender Keys (or similar) approach to group messaging. The closest security model to ours in the literature comes from that of Weidner et al. for DCGKA [WKHB21], which however does not consider message injections nor adaptive security.

Limitations. Our security game allows a single successful injection to occur, since after this point the adversary is given the secret bit for free. That is, we do not allow 'trivial' message forgeries that do not result in the adversary winning the game. Hence, full active security cannot be captured by our modelling. Like several other models in the literature (e.g., [ACDT20, KPPW+21, WKHB21]), our security model considers a single group (see [CHK21] for an analysis of cross-group security) and ignores randomness exposure or manipulation [BRV20].

6.3.2 Modelling Two-Party Channel Ciphertexts

Given that the GM protocol uses two-party channels (as Sender Keys does), these need to be modelled accurately within the GM security game, particularly to describe forgeries via the recv-forgery and proc-forgery predicates. We introduce additional notation to define how two-party ciphertexts can be sent alongside GM messages; we opt for such modelling for convenience, as in this way the adversary gets access to all two-party channels explicitly. We remark that this subsection can be skipped for GM protocols that do not employ two-party channels, since in this case, simply any ciphertext input to Proc (resp. Recv) that was not previously output by Exec (resp. Proc) would be considered a forgery.

Essentially, we want to capture the fact that an Exec or Send call can output *several* two-party channel ciphertexts, whereas Proc and Recv should only take as input a *single* two-party channel ciphertext (i.e., the one intended for the caller) for efficiency. We thus assume input/output ciphertexts and control messages for group messenger algorithms take the following form. Let C_{2pc} be a 2PC ciphertext and let T_{core} (resp. C_{core}) be the remaining part of a control (resp. application) message in the GM primitive. For output, we assume control messages output by Exec are of the form $(T_{\mathsf{core}}, C^1_{\mathsf{2pc}}, \ldots, C^k_{\mathsf{2pc}})$ for some k, and ciphertexts output by Send are of the form $(C_{\mathsf{core}}, C^1_{\mathsf{2pc}}, \ldots, C^k_{\mathsf{2pc}})$. For input, we assume control messages input to Exec (resp. to Recv) are of the form $(T_{\mathsf{core}}, C_{\mathsf{2pc}})$ (resp. $(C_{\mathsf{core}}, C_{\mathsf{2pc}})$).

Forgery Predicates. We define the predicates proc-forgery and recv-forgery in Figure 6.8 using the input/output semantics introduced above. Used in Figure 6.7, the purpose of these predicates is to handle ciphertext 'splitting' resulting from the use of two-party channels. Without accounting for this splitting, forgeries could be defined as usual. Essentially, we consider that a control message $T^* = (T^*_{\mathsf{core}}, C^*_{\mathsf{2pc}})$ is a forgery whenever either T^*_{core} or C^*_{2pc} are not part of an honestly generated message (i.e. in $\mathsf{T}[\cdot]$). Forgeries for Recv are defined analogously.

$$\boxed{\mathsf{proc\text{-}forgery}(T^* = (T^*_{\mathsf{core}}, C^*_{\mathsf{2pc}}))\colon} \; \nexists \left\{(T, \vec{C}), (T', \vec{C}')\right\} \subseteq \mathsf{T}[\cdot]\colon$$
$$(T^*_{\mathsf{core}}, C^*_{\mathsf{2pc}}) \in \left\{(T, C_i), (T', C_i), (T, C'_j)\right\} \wedge (C_i \in \vec{C}) \wedge (C'_j \in \vec{C}')$$
$$\boxed{\mathsf{recv\text{-}forgery}(C = (C^*_{\mathsf{core}}, C^*_{\mathsf{2pc}}))\colon} \; \nexists \left\{(C_0, \vec{C}), (C'_0, \vec{C}')\right\} \subseteq \mathsf{CH}[\cdot] \cup \mathsf{M}[\cdot]\colon$$
$$(C^*_{\mathsf{core}}, C^*_{\mathsf{2pc}}) \in \left\{(C_0, C_i), (C'_0, C_i), (C_0, C'_j)\right\} \wedge (C_i \in \vec{C}) \wedge (C'_j \in \vec{C}')$$

Figure 6.8: Predicates that determine what is considered a forgery in Figure 6.7 for algorithms Proc and Recv.

The predicates imply that it is *not* considered a forgery if a two-party ciphertext is received with a different control message/ciphertext than it was sent with. That is, the adversary is allowed to mix-and-match ciphertexts, i.e., by replacing the C_i corresponding to some T (resp. C_0) by C'_i corresponding to some other T' (resp. C'_0).

6.4 Sender Keys

In this section, we explain and provide a simplified protocol description for Sender Keys. We state our security theorem for Sender Keys, describe how we model the cleanness predicates that characterise its security as a Group Messenger and then sketch its security. In Section 6.6 we provide a complete protocol description and proof.

6.4.1 Protocol

We describe the Sender Keys protocol in our GM syntax according to the details inferred from [Wha20] and [M+16], although we acknowledge that our interpretation may not precisely match the closed-source implementation of WhatsApp. In this subsection we present a detailed overview of the main algorithms depicted in Figure 6.9. For Exec and Proc, we only present the remove operation as it involves key refreshing and is considered the most complex, while the create, add, and update operations follow a similar approach. For the sake of clarity, we make some simplifications in this section, but the complete protocol logic can be found later in Figures 6.19 to 6.22 (in Section 6.6, where we also provide supplementary descriptions of the protocol logic).

Two-Party Channels and the Server. The Sender Keys protocol assumes the existence of authenticated and secure two-party communication channels between each pair of users, which can be achieved through the use of Signal's Double Ratchet protocol [MP16a] also used by WhatsApp [Wha20]. Additionally, the protocol relies on a central server to distribute both control messages and application messages. We assume that the server provides a *total ordering for control messages*, ensuring that all parties process control messages in the correct order.[6] Total ordering is not required for application messages. User authentication is initially performed via the central server (modelled here with 2PC.InitCh), after which users authenticate other group members through the underlying two-party communication channel. We note that this deviates from other work in the literature such as [AJM22] where the *authentication service* is different to the *delivery service*.

Primitives. The protocol relies on standard primitives including a symmetric encryption scheme $\mathsf{SymEnc} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, a signature scheme $\mathsf{Sig} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$, and two different key derivation functions $\mathsf{H}_1, \mathsf{H}_2$ (our improved protocol also uses message authentication codes). We include formal definitions in Chapter 2.

State Initialisation. Each user is assumed to maintain a state γ containing: a secret key used for signing ssk, a list of current group members $\boldsymbol{G}$, the current epoch ep, the current index of their chain key i_{ck} (indicating the number of times the user's sender key has been ratcheted forward), a list of key counters kc (indicating the number of times that a sender key has been re-sampled since *ID* initialised their state), a dictionary of sender keys $\mathsf{SK}[\cdot] := (\mathsf{spk}_{ID}, \mathsf{ck}_{ID}, \mathsf{i}_{\mathsf{ck}})$

[6] We remark that total ordering is a standard assumption in the CGKA line of work [ACDT20, KPPW+21, ACJM20, AJM22, ACDT21a] and is assumed by MLS.

$\mathsf{Init}(ID, 1^\lambda)$

1 : $\gamma.ME \leftarrow ID$
2 : $\gamma.(\mathsf{ssk}, \boldsymbol{G}, \mathsf{ep}, \mathsf{i}_{ME}) \leftarrow \bot$
3 : $\gamma.(\mathsf{SK}[\cdot], \mathsf{MK}[\cdot], \mathsf{kc}[\cdot]) \leftarrow \bot$
4 : **return** γ

$\mathsf{Send}(m, \gamma)$

1 : **require** $ME \in \boldsymbol{G}$
2 : **if** $\mathsf{SK}[ME, \mathsf{kc}[ME]] = \bot$:
// Sample fresh sender key
3 : $\vec{C} \leftarrow \mathsf{PreSendFirst}()$
4 : **if** $\mathsf{i}_{ME} = 0$:
5 : $\vec{C} \leftarrow (\vec{C}, \mathsf{SendToMissing}())$
6 : $\mathsf{mk} \leftarrow \mathsf{H}_1(\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{ck})$
7 : $\mathsf{ct} \xleftarrow{\$} \mathsf{SymEnc.Enc}(\mathsf{mk}, m)$
// UpdateCK also updates i_{ME}
8 : $\mathsf{UpdateCK}(ME, \mathsf{kc}[ME])$
9 : $M \leftarrow (\mathsf{ct}, (\mathsf{ep}, \mathsf{i}_{ME}), \mathsf{kc}[ME], \mathsf{i}_{\mathsf{ck}}, ME)$
10 : $\sigma \xleftarrow{\$} \mathsf{Sig.Sign}(\mathsf{ssk}, M)$
11 : **return** $C := ((M, \sigma), \vec{C})$

$\mathsf{Exec}(\gamma, \mathtt{cmd} = \mathtt{rem}, ID)$

1 : **require** $ID \in \boldsymbol{G}$
2 : $\vec{C}[\cdot] \leftarrow \bot$
3 : $T \leftarrow (\mathtt{rem}, ME, ID, \mathsf{ep} + 1)$
4 : **return** $(T, \vec{C})$

$\mathsf{Recv}(\gamma, C = ((M, \sigma), C_{2\mathsf{pc}}))$

1 : **parse** M **as** $(\mathsf{ct}, (e, i), \mathsf{kc}', \mathsf{i}_{\mathsf{ck}}', ID)$
2 : **require** $ID \in \boldsymbol{G}$
3 : **if** $\mathsf{SK}[ID, \mathsf{kc}'] = \bot$:
4 : $(\mathsf{SK}[ID, \mathsf{kc}'], \mathsf{kc}^*, \mathsf{ep}', \mathsf{aux}, ID^*) \leftarrow \mathsf{2PC.Recv}(\gamma, C_{2\mathsf{pc}})$
5 : **require** $(ID, e, \mathsf{kc}') = (ID^*, \mathsf{ep}', \mathsf{kc}^*)$
6 : $\mathsf{DeleteOldCK}(ID, \mathsf{aux})$
7 : **else require** $C_{2\mathsf{pc}} = \bot$
8 : **require** $e \leq \mathsf{ep}$
9 : **require** $\mathsf{Sig.Vrfy}(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{spk}, \sigma, M)$
// Derive or fetch mk from state
10 : $\mathsf{mk} \leftarrow \mathsf{UpdateKeysRecv}()$
11 : $m \leftarrow \mathsf{SymEnc.Dec}(\mathsf{mk}, \mathsf{ct})$
12 : **return** (m, ID, e, i)

$\mathsf{Proc}(\gamma, (T = (\mathtt{rem}, ID, ID', \mathsf{ep}'), C_{2\mathsf{pc}}))$

1 : **require** $ID \in \boldsymbol{G} \wedge C_{2\mathsf{pc}} = \bot$
2 : **require** $\mathsf{ep}' = \mathsf{ep} + 1$
3 : $\boldsymbol{G} \xleftarrow{-} \{ID'\}$
4 : $\mathsf{ep} \leftarrow \mathsf{ep} + 1$; $\mathsf{i}_{ME} \leftarrow 0$
5 : **for all** $ID^* \in \boldsymbol{G}$:
6 : $\mathsf{kc}[ID^*] \leftarrow \mathsf{kc}[ID^*] + 1$
7 : $\mathsf{SK}[ID', \cdot] \leftarrow \bot$
8 : **if** $ID = ME$:
9 : $\gamma \leftarrow \bot$ // delete γ
10 : **return** true

Figure 6.9: Simplified Sender Keys protocol description (Init, Send, Recv, and example Exec and Proc operations for $\mathtt{cmd} = \mathtt{rem}$) built from symmetric encryption scheme SymEnc, signature scheme Sig, PRG $\mathsf{H} = (\mathsf{H}_1, \mathsf{H}_2)$ and two-party channels scheme 2PC. For readability we use and update state variables without explicitly referring to γ, and assume stateful algorithms implicitly return (possibly updated) state. For conditions of the form "**require** T" when T is false, the function outputs $\bot$ except for the state which is not updated. The full protocol description is provided in Section 6.6 where the helper functions PreSendFirst, SendToMissing, UpdateCK, DeleteOldCK and UpdateKeysRecv that appear in this Figure are defined.

indexed by a user ID and a key counter, and a list of message keys MK. The Init algorithm initialises the state variable of users; in practice this is done by a user when they install the messaging application.

Group Creation. This occurs via $\mathsf{Exec}(\gamma, \mathtt{crt}, IDs)$, which takes a list of users $\boldsymbol{G} := \{ID_1, \ldots, ID_{|\boldsymbol{G}|}\}$ as input; two-party channels are initialised by users upon processing the control

message via 2PC.InitCh (c.f. Section 6.6).

Message Sending. To send an application message m to the group, every $ID \in \boldsymbol{G}$ must have the caller's (ME) sender key. The process is as follows:

- If ME does not have a sender key, ME calls helper PreSendFirst, which first generates a fresh sender key $((\gamma.\mathsf{ssk}, \mathsf{spk}) \xleftarrow{\$} \mathsf{Sig.Gen}(1^\lambda)$ and $\mathsf{ck} \xleftarrow{\$} \{0,1\}^\lambda)$. The sender key is then set as $\mathsf{SK}[ME, \mathsf{kc}[ME]] \leftarrow (\mathsf{spk}, \mathsf{ck}, \mathsf{i}_{\mathsf{ck}})$ where $\mathsf{i}_{\mathsf{ck}} = 0$. ME shares this key with each $ID \in \boldsymbol{G}$ using 2PC.Send, resulting in a vector of ciphertexts $\vec{C}$.
- If ME has a non-empty sender key but not all parties have it ME, calls helper SendToMissing to share the key with them via 2PC.Send, therein updating $\vec{C}$.

Then, ME generates a new message key mk from their chain key $\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{ck}$, encrypts m using mk, and ratchets its chain key forward by setting $\mathsf{ck} \leftarrow \mathsf{H}_2(\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{ck})$. Finally, ME signs the ciphertext and sends it together with $\vec{C}$.

Message Receiving. To receive a message from ID, ME follows these steps:

- ME checks if they have ID's sender key $\mathsf{SK}[ID, \mathsf{kc}']$ corresponding to the key counter kc' indicated in the received message. If ME does not have it, they retrieve it from the two-party ciphertext $C_{2\mathsf{pc}}$ using 2PC.Recv, aborting the Recv call if the sender key cannot be found.
- ME performs epoch consistency checks and verifies the signature on the ciphertext using the signature public key $\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{spk}$.
- The message key mk required to decrypt the message is computed from the chain keys as $\mathsf{mk} \leftarrow \mathsf{H}_1(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{ck})$, and is deleted after use.

Out-of-Order Messages. In the scenario of out-of-order message delivery (handled by helper UpdateKeysRecv), the following cases arise (we let $\mathsf{i}_{\mathsf{ck}} := \mathsf{SK}[ID, \mathsf{kc}'].\mathsf{i}_{\mathsf{ck}}$):

- If the received message comes from a past epoch $(e, i) < (\mathsf{ep}, \mathsf{i}_{\mathsf{ck}})$, ME searches for the relevant skipped message key in MK.
- If $e = \mathsf{ep}$ and $i > \mathsf{i}_{\mathsf{ck}}$, ME ratchets ID's chain key $i - \mathsf{i}_{\mathsf{ck}}$ times, and stores the skipped message keys in MK.
- If $e > e$, the message reception fails since ME is not synchronised with the latest group epoch and cannot (even) determine whether the sender is still a member of the group.

Handling out-of-order message delivery constitutes a significant portion of the protocol's logic. For instance, parties must keep track of (and announce) the highest i_{ck} associated with a given kc. Failing to do so can result in correctness and security issues, as parties may overlook the need to store and delete keys in MK.

Key Updates. In at least Signal's implementation of Sender Keys (it is not mentioned in WhatsApp's white paper [Wha20]) a simple (but somewhat weak) on-demand key update mechanism is supported. A party *ME* can update its key material via $\mathsf{Exec}(\gamma, \mathtt{crt}, ME)$. This operation lazily samples a fresh sender key $(\mathsf{spk}, \mathsf{ck}, 0)$ and distributes it over the two-party channels. All users sample a fresh key after processing a removal.

Membership Changes. The protocol allows individual group members to be added or removed from the group via $\mathsf{Exec}(\gamma, \mathtt{add}, ID)$ and $\mathsf{Exec}(\gamma, \mathtt{rem}, ID)$. These operations result in the distribution of a control message T to the group sent in clear. Newly added members are also sent a welcome 2PC ciphertext containing group information. Note that we model single adds/removes for simplicity but this can be extended in a straightforward manner to handle batched group changes.

Upon processing a control message via $\mathsf{Proc}(\gamma, T)$, *ME* proceeds as follows:

- If some ID^* is being removed at epoch e, *ME* erases all sender keys corresponding to ID^* (except for skipped message keys).[7] The sender keys corresponding to other users are replaced with new ones when receiving messages from epoch $e' \geq e$, ensuring messages sent concurrently with the removal can still be received by *ME*.
- If some ID^* is being added, *ME* initialises its 2PC with ID^* via $\mathsf{2PC.InitCh}$.
- If *ME* is itself removed, it erases its state. If it is added to some group (or processes a create message), it initialises two-party channels with every $ID \in \boldsymbol{G}$.

Note that after either updating or adding or removing a user, for performance and security reasons, new sender keys are only distributed once a party sends his first message.

6.4.2 Security

In this subsection, we argue that Sender Keys as described in Section 6.4 is secure with respect to our security model in Section 6.3.1. However, the security captured by our cleanness predicate is far from theoretically optimal since Sender Keys is relatively weak in security, and so in Section 6.5 we strengthen it by modifying the protocol in different ways. Our predicates are parameterised by the security of the underlying two-party channels. We first state our main theorem below.

Theorem 20. Consider IND-CPA symmetric encryption scheme SymEnc, SUF-CMA signature scheme Sig, PRG function $\mathsf{H} : \{0,1\}^\lambda \to \{0,1\}^\lambda \times \{0,1\}^\lambda$ and 2PC-IND two-party channels scheme 2PC with respect to PCS bound δ and cleanness predicate $\mathsf{C}_{\mathsf{2pc}}$ (Figures 6.4 and 6.5). Then, for Group Messenger Sender Keys (Figures 6.19 to 6.22) we have, with respect to cleanness

[7] A different deletion schedule may be applied as long as these keys are clearly marked as being no longer valid, e.g., if ID^* announces its maximum i_{ck} value over two-party channels when it processes its own removal.

predicate $\mathsf{C}_{\mathsf{gm}} = \mathsf{C}^{\Delta}_{\mathsf{sk}}$ (Figure 6.13), that, for every efficient adversary $\mathscr{A}$ that makes at most q oracle queries, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathrm{m-ind}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq 2 \cdot \epsilon_{\mathsf{2pc}} + q^2 \cdot (\epsilon_{\mathsf{2pc}} + \epsilon_{\mathsf{sym}} + q \cdot \epsilon_{\mathsf{prg}}) + q \cdot \epsilon_{\mathsf{sig}} \ ,$$

where $\epsilon_{\mathsf{sym}} = \mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathsf{SymEnc}}(\mathscr{B})$, $\epsilon_{\mathsf{sig}} = \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{B})$, $\epsilon_{\mathsf{prg}} = \mathsf{Adv}^{\mathrm{prg}}_{\mathsf{H}}(\mathscr{B})$ and $\epsilon_{\mathsf{2pc}} = \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B})$.

We define C_{sk} below and prove the theorem in Section 6.6.2. A proof sketch is provided at the end of this section. Our security notion is adaptive as users can adaptively call oracles and in particular compromise users. Security is tighter when we restrict the game to consider non-adaptive adversaries as described below.

Corollary 1. Under the same conditions of Theorem 20, and considering a non-adaptive security game, for every efficient adversary $\mathscr{A}$ that makes at most q oracle queries, one can build an adversary $\mathscr{B}$ such that

$$\mathsf{Adv}^{\mathrm{m-ind}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq 2 \cdot \epsilon_{\mathsf{2pc}} + q \cdot (\epsilon_{\mathsf{sym}} + q \cdot \epsilon_{\mathsf{prg}}) + q \cdot \epsilon_{\mathsf{sig}} \ ,$$

where $\epsilon_{\mathsf{sym}}, \epsilon_{\mathsf{sig}}, \epsilon_{\mathsf{prg}}$ and ϵ_{2pc} are defined above.

Sender Keys and Two-Party Channels. To illustrate how the cleanness predicates for Sender Keys must depend on the underlying two-party channels, consider a strongly secure two-party channel 2PC that provides optimal FS and PCS. Now, consider an execution of Sender Keys where all parties share the same view of the group $G = \{ID_1, ID_2, ID_3\}$, in which

1. ID_1 generates a control message $(T_{\mathsf{core}}, \vec{C})$ to remove party ID_3 ($q_1 = \mathsf{REMOVE}(ID_1, ID_3)$),
2. ID_1 and ID_2 process T ($q_{2,1} = \mathsf{DELIVER}(ID_1, (T_{\mathsf{core}}, \vec{C}[ID_1]))$, $q_{2,2} = \mathsf{DELIVER}(ID_2, (T_{\mathsf{core}}, \vec{C}[ID_2]))$),
3. $\mathscr{A}$ exposes ID_2 ($q_3 = \mathsf{EXPOSE}(ID_2)$);
4. ID_1 sends an application message ($q_4 = \mathsf{SEND}(ID_1, m)$).

Recall that in step 4, ID_1 samples a new sender key that it sends to ID_2 over 2PC, since processing remove messages results in the sender keys of all parties being refreshed. Even with optimally-secure 2PC, the adversary will be able to decrypt the key sent over 2PC (by the correctness of the channel) and thus decrypt the ciphertext output in query q_4.

Cleanness. Our goal is to describe a suitable cleanness predicate C_{sk} for Sender Keys. The intuition behind this cleanness predicate is based on the following observations about the protocol:

- The exposure of a group member compromises the security of subsequent chain and message keys[8] until a secure key refresh takes place. This enables the adversary to forge messages since they also gain access to the exposed signature keys.
- Control messages can be trivially forged and injected by a network adversary as they are not authenticated.
- Forward-secure confidentiality holds except for messages delivered out-of-order since parties only delete message keys after using them, so a message that is delayed forever results in the corresponding message key never being deleted.[9]
- All parties recover from state exposure (via $\mathsf{EXPOSE}(ID)$) after security on the two-party channels is restored (considering the PCS bound Δ) and then either a) a removal is made effective, or b) all parties update their keys successfully.

To formalise the security predicate we introduce conventions for tracking the channel epochs of each user's two-party channels. We assume the game M-IND maintains the largest channel epoch-index for each user's two-party channels over time. The game obtains this information by observing the channel epoch-index pairs generated by the 2PC.Send and 2PC.Recv operations within the group messenger. Specifically, we use a variable of the form $\mathsf{EI}[ID, ID']$, where $\mathsf{EI}[ID, ID']$ represents the largest channel epoch-index pair from ID's perspective for the channel between them and user ID', as for two-party channels. More generally, two-party state variables that we use below can be tracked easily by an M-IND adversary such that our predicates are well-defined.

The $\mathsf{refresh}_\Delta$ Predicate. We define the predicate $\mathsf{refresh}_\Delta(ID, ID', q_i, e)$, parameterised by the PCS bound $\Delta > 0$ of the underlying two-party channels. Informally, given that ID' is exposed in query q_i ($q_i = \mathsf{EXPOSE}(ID')$), $\mathsf{refresh}_\Delta(ID, ID', q_i, e)$ is true if the (ID, ID') channel has healed and *then* ID has sampled a fresh sender key in or by epoch e (or will do so upon their next Send call). If the predicate is true, ID has recovered from the exposure in q_i.

More formally, let $(e_{2\mathsf{pc}}, i_{2\mathsf{pc}}) = \max\{\mathsf{EI}[ID', ID; q_i], \mathsf{EI}[ID, ID'; q_i]\}$. Then $\mathsf{refresh}_\Delta(ID, ID', q_i, e)$ is true if a) for $(e'_{2\mathsf{pc}}, i'_{2\mathsf{pc}}) = \mathsf{EI}[ID', ID; q_j]$ for some $j > i$, $e'_{2\mathsf{pc}} \geq e_{2\mathsf{pc}} + \Delta$ holds; and b) during query q_k with $k \geq j$, member ID processes one of the following control messages corresponding to epoch e:

1. a removal of some member ID^*,
2. an addition of ID itself,
3. a group creation message, or

[8] Although it is not captured in our model, note that the exposure of a message key alone only compromises the message it refers to and does not (computationally) leak information about the chain key or other message keys.

[9] In practice, applications like WhatsApp and Signal bound the amount of (logical) time that keys are active for and the total number of keys that can be stored at once.

4. an update from ID itself.

As such, if ID executes (and processes) an update that involves sending new key material over a refreshed two-party channel, this key material should be safe. We also define a simpler predicate refresh-s(ID, e) which is true if member ID processes one of the aforementioned control messages corresponding to epoch e. Observe that both $\mathsf{refresh}_\Delta$ and refresh-s events may only happen when ID moves to a new group epoch e.

Cleanness for Sender Keys. We divide our cleanness predicate into three components (challenge, injection, concurrency) that we specify below. The final predicate (the conjunction of these three components) is defined in Figure 6.13.

Challenge (Figure 6.10). The effect of this predicate is to prevent challenges on exposed users (i.e., due to EXPOSE calls). After exposing (with query q_i) *any* user ID', adversarial queries to CHAL are disallowed for every ID in the group until $\mathsf{refresh}_\Delta(ID, ID', q_i, e)$ occurs for some later epoch $e > \mathsf{E}[ID'; q_i]$. Note that this only restricts challenge queries q_j where $i < j$. To capture forward security precisely, some challenges made before an exposure ($i > j$) are also forbidden. These affect messages sent by some ID in message epochs $(e, i) \geq \mathsf{m\text{-}ep}(ID', ID, q_i)$, which correspond to keys that ID' still stores (including skipped message keys stored at exposure time) or can derive due to being in a previous message epoch (for example if the user is offline).

$$
\begin{aligned}
&\boxed{\mathsf{C}^{\Delta}_{\text{sk-chall:}}}\ \forall (i, j, ID, ID') : q_i = \mathsf{EXPOSE}(ID') \wedge q_j = \mathsf{CHAL}(ID, \cdot, \cdot),\\
&\quad \big(i > j \wedge \mathsf{m\text{-}ep}(ID', ID, q_i) > (\mathsf{E}[ID; q_j], \mathsf{I}[ID; q_j]) \wedge\\
&\quad (\mathsf{E}[ID; q_j], \mathsf{I}[ID; q_j]) \notin \mathsf{SM}[ID', ID; q_i]\big)\\
&\quad \vee \big(i < j \wedge \exists e : \mathsf{E}[ID'; q_i] < e \leq \mathsf{E}[ID; q_j] \wedge \mathsf{refresh}_\Delta(ID, ID', q_i, e)\big)
\end{aligned}
$$

Figure 6.10: Challenge cleanness predicate for Sender Keys where the adversary makes oracle queries $q_1, \ldots, q_q$.

Injection (Figure 6.11). Firstly, let us recall the two-party ciphertext splitting semantics defined in Section 6.3.2. Namely, a GM ciphertext C naturally splits into $C = (C_{\mathsf{core}}, C_{\mathsf{2pc}})$ where C_{2pc} is processed by the two-party channels. An injection is said to have occurred when a message with a forged C_{core} and/or C_{2pc} was successfully processed.

We define the injection predicate to prevent injections of *application messages* coming from a user that has been exposed and has not refreshed its keys (we consider control messages in the last predicate). We start with the definition for C_{core}. After exposing *a specific* user ID' with query q_i, $\mathscr{A}$ cannot make a query $q_j = \mathsf{RECEIVE}(ID, C)$ to impersonate ID' with a forgery ciphertext C corresponding to some epoch e^* (i.e., such that tuple (ID', e^*) is output by $\mathsf{Recv}(\gamma[ID], C)$ in the game) in the following situations:

1. $e^* \geq \mathsf{E}[ID'; q_i]$ and there hasn't been a $\mathsf{refresh}_\Delta(ID', ID, q_i, e')$ event for the sender ID' at some epoch e' such that $\mathsf{E}[ID'; q_i] < e' \leq e^*$, where the receiver ID has also processed the key update from ID''s message at injection time, i.e., $\mathsf{E}[ID; q_j] \geq e'$.

2. $e^* < \mathsf{E}[ID'; q_i]$ but the signature key of ID' at epoch e^* was the same key as in the exposure epoch $\mathsf{E}[ID'; q_i]$. Formally, this is expressed by the condition that there has not been any event $\mathsf{refresh\text{-}s}(ID', e')$ for an epoch $e^* < e' \leq \mathsf{E}[ID'; q_i]$.

For $C_{2\mathsf{pc}}$, we directly adopt the injection cleanness predicate $\mathsf{C}_{2\mathsf{pc\text{-}inj}}$ used to define two-party channel security (Figure 6.5). For additional clarity, we parametrise the predicates by the ciphertexts $C_{\mathsf{core}}, C_{2\mathsf{pc}}$. We also define the auxiliary predicate $\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}})$ in Figure 6.11.

$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}}):}\ \forall (i, j, ID, ID'):$
$(C_{\mathsf{core}}, \cdot) \notin \mathsf{M}[ID', \cdot; q_j] \wedge (i < j) \wedge q_i = \mathsf{EXPOSE}(ID') \wedge$
$q_j = \mathsf{RECEIVE}(ID, (C_{\mathsf{core}}, \cdot)) \wedge (\cdot, e^*, \cdot, ID') \leftarrow \mathsf{Recv}(\gamma[ID; q_j], (C_{\mathsf{core}}, \cdot))$ in q_j,
$\exists e' : [\ (\mathsf{E}[ID'; q_i] < e' \leq e^* \wedge \mathsf{refresh}_{\Delta}(ID', ID, q_i, e'))$
$\vee\ (e^* < e' \leq \mathsf{E}[ID'; q_i] \wedge \mathsf{refresh\text{-}s}(ID', e'))\ ]$

$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj}}:}\ \forall\ C_{\mathsf{core}}, C_{2\mathsf{pc}}, \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}}) \wedge \mathsf{C}_{2\mathsf{pc\text{-}inj}}(C_{2\mathsf{pc}})$

Figure 6.11: Auxiliary core injection cleanness predicate (top) and injection cleanness predicate (bottom) for Sender Keys, where the adversary makes oracle queries $q_1, \ldots, q_q$. The injection cleanness predicate additionally uses $\mathsf{C}_{2\mathsf{pc\text{-}inj}}$ (Figure 6.5).

Concurrency (Figure 6.12). This predicate ensures several properties in the protocol. Firstly, it enforces that users process control message in the same order (albeit they need not be synchronised beyond this restriction). Additionally, it prevents the injection of *all* control messages. It is important to note that control messages are not signed in the core protocol, making injections trivial. Furthermore, the predicate guarantees that every user proposing a group change (via the Exec, ADD, REMOVE or UPDATE oracles) is in the most recent epoch. In practice, this predicate ensures that there is a unique honest control message in each epoch of the game. To complete the predicate, we further enforce that the two-party channel ciphertext $C_{2\mathsf{pc}}$ input to each DELIVER call is either honest or is allowed by the two-party channel injection predicate $\mathsf{C}_{2\mathsf{pc\text{-}inj}}$ (Figure 6.5).

The concurrency predicate ensures both security and correctness by addressing scenarios where members propose concurrent group changes or process group changes in different orders. Without enforcing this predicate, the behaviour of the protocol, as specified in this work, becomes ill-defined.

$\boxed{\mathsf{C}_{\mathsf{sk\text{-}con}}:}\ \forall (i, ID) : q_i = \mathsf{DELIVER}(ID, (T_{\mathsf{core}}, C_{2\mathsf{pc}})), \exists j < i :$
$q_j = (\mathsf{ADD}$ or REMOVE or UPDATE or $\mathsf{CREATE})(ID, \cdot) \wedge \exists e' : (T, \vec{C}) = \mathsf{T}[e'] \wedge$
$(C_{2\mathsf{pc}} \in \vec{C} \vee \vec{C} = \perp \vee \mathsf{C}_{2\mathsf{pc\text{-}inj}}(C_{2\mathsf{pc}})) \wedge (\mathsf{E}[ID; q_i] = e' - 1 = \mathsf{E}[ID; q_j])$

Figure 6.12: Concurrency cleanness predicate in the ideal case where the adversary makes oracle queries $q_1, \ldots, q_q$. This predicate additionally uses $\mathsf{C}_{2\mathsf{pc\text{-}inj}}$ (Figure 6.5).

Limitations and Extensions. Our cleanness predicate enforces a total ordering on control messages, in contrast to considering causal ordering such as in Weidner et al.'s mod-

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}}:\ \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}chall}} \wedge \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj}} \wedge \mathsf{C}_{\mathsf{sk\text{-}con}}}$$

Figure 6.13: Sender Keys cleanness predicate which makes use of sub-predicates defined in Figures 6.10 to 6.12.

elling [WKHB21], or no ordering at all. This assumption is consistent with real-world protocols (as in WhatsApp) where a central server is trusted to provide such an ordering, but makes our model unsuitable for decentralised protocols. If our security model allowed for it, one could modify our cleanness predicates to allow for 'trivial' injections that are non-winning, by not giving the adversary the challenger's bit b given that the forgery is trivial (i.e., it violates the injection predicate). Our concurrency predicate and security model could be strengthened to allow several Exec calls in an epoch, from which the network chooses one that is processed to all parties, which has been modelled for TreeKEM in the past [ACDT20].

Proof Sketch for Theorem 20. Towards proving the theorem (see Section 6.6.2 for the full proof), we construct a series of hybrids. We first transition to a game where injections on the two-party channels are disallowed, following from their underlying security. After that, we transition to a game where oracle RECEIVE never outputs challenge bit b, reducing the transition to SUF-CMA signature security, while still excluding trivial injections due to cleanness. Then, we move to a game where the adversary is limited to a single CHAL query, losing a factor of q in the resulting reduction. Subsequently, we transition to a game where the message key used in the CHAL query (if it exists) is replaced by a uniformly random key that remains unknown to the adversary due to cleanness, and the two-party ciphertexts that send the key's ancestor chain key are replaced by dummy ciphertexts, which follows from the 2PC security and the PRG security of $(\mathsf{H}_1, \mathsf{H}_2)$. Finally, we directly reduce to the IND-CPA security of the symmetric encryption scheme.

6.5 Analysis and Improvements

For the proof of security of Sender Keys (Theorem 20) to go through, we need to impose severe restrictions on the adversarial behaviour through the cleanness predicate C_{sk}. Hence, even though we managed to prove Sender Keys secure, we did so in a weak model that reveals important security shortcomings of the protocol. In this section, we elaborate on these limitations and propose changes to enhance security while maintaining efficiency. Some of these findings were presented in a preliminary analysis in [BCG22].

6.5.1 Security Analysis and Limitations

Injection of Control Messages. Our first observation is that control messages lack user authentication, necessitating a high level of trust in the server to prevent the crafting of its own messages. To address this, in predicate $\mathsf{C}_{\mathsf{sk\text{-}con}}$ we need to enforce that every delivered control

message has been honestly generated. A server deviating from standard behavior could mount a host of attacks. We give three examples below.

Censorship attack: The server can remove any member(s) ID from $\boldsymbol{G}$ such that all remaining members assume ID left the group by himself, whilst ID believes a different user ID' removed him.

- The server delivers a control message $T := (\texttt{rem}, ID, ID, \cdot)$ where $(T, \ldots) \leftarrow \mathsf{Exec}(\cdot, \texttt{rem}, ID)$ to every $ID' \in \boldsymbol{G} \setminus \{ID\}$.
- The server delivers a control message $T' := (\texttt{rem}, ID', ID, \cdot)$ where $(T', \ldots) \leftarrow \mathsf{Exec}(\cdot, \texttt{rem}, ID)$ to $ID \in \boldsymbol{G}$.

Burgle into the group attack: This attack, observed by Rösler et al. [RMS18], allows the server to add any member(s) ID to $\boldsymbol{G}$. For this, the server just delivers a control message $T := (\texttt{add}, \cdot, ID, \cdot)$ where $(T, \ldots) \leftarrow \mathsf{Exec}(\cdot, \texttt{add}, ID)$ to every $ID' \in \boldsymbol{G}$.

Unsafe group administration: In general, administration cannot be enforced or trusted due to the lack of authentication of control messages, as discussed in Section 5.1.

Weak Post-Compromise Security. Sender Keys offers only a limited form of PCS. Essentially, a $\mathsf{refresh}_\Delta$ event is the only possibility for ID to recover from a state compromise. This event only occurs whenever another user is removed or whenever ID triggers an on-demand update (or trivially when ID is new to the group). On-demand updates are supported by our primitive syntax and protocol description, but it is not clear whether they are implemented in practice beyond Signal (there is no mention to them in WhatsApp's white paper [Wha20], for instance).

Moreover, the update mechanism is not satisfactory. Since only the updater ID refreshes its sender key, this allows a passive adversary to eavesdrop on messages sent by any other group member due to the adversary's knowledge of the chain keys corresponding to those members. Extending the update mechanism to the entire group in a naive manner would result in a total communication complexity of $\mathcal{O}(n^2)$.

PCS and Two-Party Channels. PCS guarantees are even weaker due to the reliance of Sender Keys on two-party channels. As parametrised by $\mathsf{refresh}_\Delta$, if ID sends new key material over a two-party channel with ID' that has not been healed (after Δ back-and-forth messages) since the last exposure of either ID or ID', then such key material is still compromised. In practice, if the state of ID is compromised, both the group and the two-party sessions will be exposed. Therefore, unless parties refresh their individual two-party channels consciously (by sending each other messages), executing updates or removals in the group session will not have the desired healing effect.

In the real world, usually not all pairs of members of a group exchange private messages regularly, hence not refreshing their two-party channels. The fact that even manually triggering a

key update does not necessarily heal the group from a state compromise conveys an important security limitation.

Lack of Forward Security on Authentication. Beyond PCS limitations, we observe that the forward security guarantees for authentication provided by Sender Keys are sub-optimal. Consider a simple group $\boldsymbol{G} = \{ID_1, ID_2\}$ and the attack described in Figure 6.14.

1: $q_1 = \mathsf{SEND}(ID_1, m)$ generates ciphertext C encrypted under mk and signed under ssk_1.
2: $q_2 = \mathsf{EXPOSE}(ID_1)$, where $\mathscr{A}$ obtains ssk_1, but not mk.
3: $\mathscr{A}$ modifies C and signs it again under ssk_1 to create a forgery C' corresponding to the same message epoch as C.
4: $q_3 = \mathsf{RECEIVE}(ID_2, ID_1, C')$, which is a successful injection.

Figure 6.14: Attack on authentication forward security in Sender Keys.

Note that q_3 is a forbidden query by $\mathsf{G}_{\mathsf{sk\text{-}inj}}$. q_3 attempts to inject a message that corresponds to key material used *before* the state exposure, hence one can envision stronger FS where queries like q_3 are allowed. This attack can occur naturally if ID_2 is offline when m is first sent by ID_1.

An attack of a similar nature can also occur in a messaging scheme where the same signature keys are re-used across groups, and are refreshed at different times, as pointed out by Cremers et al. [CHK21].

Additional Remarks. In a Sender Keys group, each user is associated with a distinct symmetric key, resulting in a state that contains $\mathcal{O}(n)$ secret material at all times. However, the exposure of a single member compromises the keys of all group members, rendering this use of multiple keys ineffective for enhancing security. The primary advantage of employing multiple keys is for concurrency reasons. Nevertheless, in large groups, this approach can pose scalability challenges. As a result, it is possible to explore trade-offs between the level of concurrency supported and the amount of secret material that needs to be stored at any given time. At the opposite end of the spectrum, all users could maintain a single symmetric chain that is the *same* for everyone. This approach, suitable for scenarios where concurrent message transmission is unlikely, reduces the state size to $\mathcal{O}(1)$ and requires $\mathcal{O}(n)$ PCS updates. Moreover, this would improve security by reducing PCS updates to the sending and processing of a single constant-sized message, which represents an optimal solution.

Sender Keys, as described in Section 6.2.1, is susceptible to randomness exposure and randomness manipulation attacks. Namely, the adversary does not need to leak a member's state, but simply control the randomness used by the device, inhibiting any form of PCS. Protection against this family of attacks can be attained at small cost if freshly generated keys are hashed with the state [JS18] and the classic NAXOS trick used in authenticated key exchange [LLM07].

Other attacks are possible but unavoidable unless symmetric encryption for application

messages is replaced by some form of public-key encryption. Suppose that the adversary $\mathscr{A}$ exposes user ID ($\gamma \leftarrow \mathsf{EXPOSE}(ID)$) and then calls $C \leftarrow \mathsf{CHAL}(ID, m_0, m_1)$. $\mathscr{A}$ can trivially win since it can derive mk used in the CHAL query trivially from γ. Some ratcheting protocols provide strong security in that, if a user is impersonated towards, their state should 'diverge' and no longer be useful for decrypting messages from honest parties [PR18, JS18, BRV20]. This is not possible to achieve in a Sender Keys-like protocol (nor is achieved by the Double Ratchet or MLS for messages in the same symmetric ratchet) since the key schedule for message encryption is deterministic and independent of previously received messages.

Although our formalism does not capture multi-group security, it can be adapted to capture it, for example by using group identifiers to label different groups. In principle, reusing the same two-party channels between parties that are in several groups together could lead to an *increase* in security by faster healing (since more messages would be sent on the same channel).[10] As argued by Cremers et al. [CHK21], if a party reuses the same signature key across groups (ignoring any privacy concern from doing so), PCS authenticity guarantees would improve, since replacing the key after channels are healed would immediately heal authenticity across all groups (rather than just one). We leave it open to explore cross-group security further and in particular capture multi-group security formally.

6.5.2 Proposed Improvements: Sender Keys+

We propose several improvements to Sender Keys below. Our improvements are constrained by the desire to retain the performance characteristics and structure of Sender Keys. In particular, we retain $\mathcal{O}(1)$-sized ciphertexts, do not increase key sizes, and require only standard cryptographic primitives. Our improved version of Sender Keys, which we call Sender Keys+, is presented fully in Figures 6.19 to 6.22. We formalise security by introducing several modifications to our cleanness predicate that we describe at the end of this section.

Secure Control Messages. A simple way of resolving the attacks in Section 6.5.1 would be for users to sign their own control messages and verify signatures before processing control messages. Additional protocol logic for correctness is required, namely that users who craft a control message but have not shared their sender key yet (because they have not spoken in the group) generate a signature key pair and share their public key over the two-party channels.

By introducing this tweak, we can weaken the cleanness predicate such that it no longer enforces honest control message delivery ($\mathsf{C}_{\mathsf{sk}^+\text{-}\mathsf{con}}$). On the other hand, we need to introduce the restriction that no secret signature key ssk can be known to the adversary at delivery time, similar as in the injection predicate. We do so by introducing a new control predicate $\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-}\mathsf{ctr}}$ that follows the blueprints of the injection predicate (Figure 6.11), and that we show in Figure 6.15.

[10] Note this would not reduce security upon state exposure since all of a given party's state, and therefore channels, are assumed (in this thesis) to be compromised at the same time.

Improved Forward-Secure Authentication. We propose two possible improvements that address the attack in Section 6.5.1 (Figure 6.14) to varying extents.

MACing from the Chain Key. The first improvement, which has minimal overhead, is to MAC the application messages with a MAC key $\tau\mathsf{k}$ that we derive via an additional $\mathsf{H}_3(\mathsf{ck})$. The modification is done in the Send algorithm as follows: given an unsigned ciphertext $\tilde{C} = (\mathsf{ct}, (e, i), ME)$, we obtain the MAC tag $\tau \leftarrow \mathsf{MAC.Tag}(\tau\mathsf{k}, \tilde{C})$. Then, we sign the ciphertext with the appended tag $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{ssk}, (\tilde{C}, \tau))$. The verification of the MAC tag is easily carried out at the receiver's end. We include this simple tweak in the protocol in Figures 6.19 to 6.22. Naturally, symmetric encryption could instead be replaced with an AEAD to achieve the same effect.

The main security improvement that results from this upgrade is that, in the attack in Section 6.5.1, the adversary additionally needs knowledge of $\tau\mathsf{k}$ to forge the MAC tag. Hence, one of the following situations must occur before delivery:

- The sender *ID* is exposed before the message is sent. Then, both $\tau\mathsf{k}$ and ssk are compromised.
- The sender *ID* is exposed after the message is sent (leaking ssk), and another group member *ID*′ is exposed before the message is delivered (leaking $\tau\mathsf{k}$).

In particular, the attack of Figure 6.14 no longer results in a successful message delivery. The MAC key can be stored together with the message key for out-of-order messages, such that the MAC can always be verified in a correct execution of the protocol. We note that insider attacks (forgeries from other group members) cannot be prevented by MACing, but we do not model these.

The modified injection predicate that results from this improvement is shown in Figure 6.17. Essentially, we define an auxiliary predicate $\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-inj-extra}}$ that considers the security given by the message/MAC keys (similarly as in Figure 6.10). Then, the modified $\mathsf{C}_{\mathsf{sk}^+\text{-inj}}$ is the logical disjunction of the former injection predicate with $\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-inj-extra}}$, and is hence strictly weaker.

Ratcheting Signature Keys. An alternative mitigation strategy for the attack of Figure 6.14 is to ratchet signature keys. Let $(\mathsf{ssk}, \mathsf{spk})$ be *ID*'s signature key pair, where spk is part of its sender key. Before sending a new message m to the group, *ID* can generate a new key pair $(\mathsf{ssk}', \mathsf{spk}') \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$. Then, *ID* can attach the new spk' to the ciphertext corresponding to encrypting m, and sign the package using ssk. This (by now standard) countermeasure not only provides strong forward security but also post-compromise security for the authentication of messages. Nevertheless, it involves larger overhead, so it may not be desirable in all scenarios and we refrain from including it in Sender Keys+.

Efficient PCS Updates. We propose an asynchronous update mechanism to refresh all chain keys at once, recovering PCS on-demand for the whole group with a single update (and $\mathcal{O}(n)$

complexity for a group of n users). Recall that our Group Messenger primitive supports updates via $\mathsf{Exec}(\gamma, \mathtt{upd}, \{ID\})$.

A Naive Solution. Let ID be the updating party. ID generates a new sender key for himself as in the case of a remove operation; namely samples a fresh ck and a fresh $(\mathsf{ssk}, \mathsf{spk}) \xleftarrow{\$} \mathsf{Sig.Gen}(1^\lambda)$. Additionally, ID samples randomness $r \xleftarrow{\$} \{0,1\}^\lambda$. Then, it distributes $(\mathsf{ck}, \mathsf{spk}, r)$ over the two-party channels. Upon reception, every group member (including ID itself) sets $\mathsf{SK}[ID] \leftarrow (\mathsf{ck}, \mathsf{spk})$; and then for every $ID' \in \boldsymbol{G}$, set $\mathsf{SK}[ID'].\mathsf{ck} \leftarrow \mathsf{H}'(\mathsf{SK}[ID'].\mathsf{ck}, r)$, where $\mathsf{H}' : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ is a secure key derivation function. Since r is freshly sampled and distributed securely, all chain keys recover from exposure. Note that r must be used and erased immediately, as all updated chain keys are exposed if r is leaked at any future time.

Our Solution. The previous solution fails in out-of-sync scenarios such as the following. Suppose that ID' is in message epoch $(1,1)$ when ID sends an update message T. Then, ID' speaks in the group before receiving T (for example, while being offline), ratcheting its key to $(1,2)$. All group members will update the chain key $\mathsf{ck}^{1,1}_{ID'}$ (i.e. corresponding to the message epoch $(e,i) = (1,1)$) in $\mathsf{SK}[ID']$, but ID' will be in message epoch $(1,2)$ (and therefore will have erased $\mathsf{ck}^{1,1}_{ID'}$). In general, if there are application messages in transit concurrently with the update, users will be out-of-sync.

To support asynchronicity, we propose that all parties ratchet their chain key N times forward, where N is a fixed constant that we call the *concurrency bound* (for example $N = 100$; in practice the cost of executing 100 hash function calls sequentially is negligible). In the event that ℓ messages have been sent out-of-sync, then the chain key is ratcheted $N - \ell$ times instead. Then, parties update the ratcheted chain keys with the sent randomness r. To synchronise between them and the update initiator ID, the latter sends a list with his view of the key indices of each group member (in the control message). We describe the update protocol as part of Figures 6.19 to 6.22. Note that this mechanism requires the assumption of total ordering of control messages to avoid overlapping updates.

The security improvement is reflected in the challenge cleanness predicate in Figure 6.16. The predicate follows the blueprint of the challenge predicate for Sender Keys (Figure 6.10), except that now it also suffices that *some* arbitrary member ID^* that has a healed channel with ID' updates after the exposure, and that ID processes such update before the challenge.

Efficient Remove Operations. The previous update mechanism can be extended to improve the efficiency of group removals from $\mathcal{O}(n^2)$ (everyone needs to generate and distribute a new key) to $\mathcal{O}(n)$ in terms of communication complexity. Note a removal can be made effective if the party that sends the remove message T distributes update material among all group members except for the removed party ID'. If ID' leaves, the next member that speaks in the group must also trigger an update. This tweak, like our solution above, has the drawback that the signature keys are not refreshed. To tame the written complexity of the protocol, we do not include this tweak in Sender Keys+. Furthermore, considering the minimal overhead of updates, they could potentially become the preferred method for sharing sender keys in the

group under all circumstances (we also do not use this tweak when a user samples their first sender key at group creation time or upon being added to a group). This approach allows the group to achieve PCS almost for free.

Cleanness Predicates for Sender Keys+. The cleanness predicates corresponding to our improvements and described informally above are detailed in Figures 6.15 to 6.17, and the joint predicate $\mathsf{C}^{\Delta}_{\mathsf{sk}^+}$ in Figure 6.18.

$$\boxed{\mathsf{C}_{\mathsf{sk}^+\text{-con}}:}\ \forall(i,ID): q_i = \mathsf{DELIVER}(ID,T), \exists j < i:$$
$$q_j = (\mathsf{ADD}\text{ or }\mathsf{REMOVE}\text{ or }\mathsf{UPDATE}\text{ or }\mathsf{CREATE})(ID,\cdot)\ \wedge$$
$$\exists e' : \mathsf{E}[ID;q_i] = e' = \mathsf{E}[ID;q_j]$$

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-ctr-core}}(T_{\mathsf{core}}):}\ \forall(i,j,ID,ID'):$$
$$(T_{\mathsf{core}},\cdot) \notin \mathsf{T}[\cdot] \wedge (i<j) \wedge q_i = \mathsf{EXPOSE}(ID')\ \wedge q_j = \mathsf{DELIVER}(ID,(T_{\mathsf{core}},\cdot)),$$
$$\exists e' : \big[\ (\mathsf{E}[ID';q_i] < e' \le e^* \wedge \mathsf{refresh}_{\Delta}(ID',ID,q_i,e')) \vee (e^* < e' \le \mathsf{E}[ID';q_i] \wedge \mathsf{refresh\text{-}s}(ID',e'))\ \big]$$

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-ctr:}}}\ \forall\ T_{\mathsf{core}}, C_{2\mathsf{pc}} : \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}ctr\text{-}core}}(T_{\mathsf{core}}) \wedge \mathsf{C}_{2\mathsf{pc\text{-}inj}}(C_{2\mathsf{pc}})$$

Figure 6.15: Modified concurrency predicate, additional auxiliary core control predicate, and additional control predicate for Sender Keys+.

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-chall}}:}\ \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}chall}} \vee \big[i<j \wedge \exists e,e',k,ID^* : \mathsf{E}[ID';q_i] < e' < e \le \mathsf{E}[ID;q_j] \wedge$$
$$q_k = \mathsf{UPDATE}(ID^*) \wedge \mathsf{ep}[q_k] = e-1 \wedge \mathsf{refresh}_{\Delta}(ID^*,ID',e',q_k)\big]$$

Figure 6.16: Modified challenge cleanness predicate for Sender Keys+.

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-inj-extra}}(C_{\mathsf{core}}):}\ \ \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}})$$
$$\vee \big[i>j \wedge \mathsf{m\text{-}ep}(ID',ID,q_i) > (\mathsf{E}[ID;q_j],\mathsf{I}[ID;q_j])\ \wedge (\mathsf{E}[ID;q_j],\mathsf{I}[ID;q_j]) \notin \mathsf{SM}[ID',ID;q_i]\big]$$
$$\vee \big[i<j \wedge \exists e : \mathsf{E}[ID';q_i] < e \le \mathsf{E}[ID;q_j] \wedge \mathsf{refresh}_{\Delta}(ID,ID',q_i,e)\big]$$
$$\vee \big[i<j \wedge \exists e,e',k,ID^* : \mathsf{E}[ID';q_i] < e' < e \le \mathsf{E}[ID;q_j] \wedge$$
$$q_k = \mathsf{UPDATE}(ID^*) \wedge \mathsf{ep}[q_k] = e-1 \wedge \mathsf{refresh}_{\Delta}(ID^*,ID',e',q_k)\big]$$

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-inj:}}}\ \forall\ C_{\mathsf{core}}, C_{2\mathsf{pc}} : \mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-inj-extra}}(C_{\mathsf{core}}) \wedge \mathsf{C}_{2\mathsf{pc\text{-}inj}}(C_{2\mathsf{pc}})$$

Figure 6.17: Modified auxiliary injection predicate and injection predicate for Sender Keys+. We remark that the additional logic simply mimics the structure of $\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-chall}}$.

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+}:}\ \mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-chall}} \wedge \mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-inj}} \wedge \mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-con}} \wedge \mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-ctr}}$$

Figure 6.18: Modified cleanness predicate for Sender Keys+.

6.5.3 Sender Keys/Sender Keys+ vs CGKA

As remarked in the introduction, Sender Keys (and especially Sender Keys+) offers different efficiency and security trade-offs over CGKA-based protocols. We provide a detailed comparison below.

PCS. When a user *ID* is exposed, the confidentiality of all subsequent messages is lost in both CGKA and Sender Keys(+). For an update to take effect in Sender Keys(+), all two-party channels must have healed. In this case, an update by ID' only heals the confidentiality of messages sent by ID' in Sender Keys, as opposed to the confidentially of messages sent by *all* members in Sender Keys+.

It is worth noting that both Sender Keys and Sender Keys+ require up to PCS bound Δ messages (or rounds) to heal after a compromise (due to the two-party channels) in addition to the update message. In contrast, a single message suffices for some CGKA protocols [ACDT20, KPPW+21, ACJM20, AHKM22].

Update Efficiency. In Sender Keys+, an update message requires $O(n)$ communication by the updating user, where each member is sent a constant-size message. In TreeKEM variants, or in general binary-tree-based CGKAs, updates involve best-case $O(\log n)$ size for the updating user and have to be entirely downloaded by each member, involving a total $O(n \log n)$ download overhead. Nevertheless, this can degrade to $O(n)$ per member (i.e., $O(n^2)$ total). The multi-recipient PKE approach taken by Hashimoto et al. [HKP+21] achieves the same asymptotic complexity as Sender Keys+, although with larger concrete costs.

Insider Security. The attack of Alwen et al. [AJM22] that highlights the need for IND-CCA (and not only IND-CPA) public-key encryption in TreeKEM also applies to symmetric encryption in Sender Keys, but can be mitigated by using a MAC (appropriately) or an AEAD. Considering the analysis of Alwen et al. [AJM22], it is not clear how to mount fake group attacks as they do, although if different users process different control messages, they may end up with different views of the group. This attack however also applies to CGKAs in general.

Separately, we note that Sender Keys(+) does not suffer from the forward security issues from MLS's CGKA [ACDT20].

6.5.4 Sender Keys in Practice

We discuss different aspects of how our formalism compares to the implementation of Sender Keys by WhatsApp and Signal in practice.

AEAD for encryption. Both WhatsApp [Wha20] and Signal[11] use AES-256 in CBC mode for Sender Keys, which does not provide authentication guarantees.

Control messages in WhatsApp. Add and remove operations are processed following [RMS18, Sec. 5.2.2, (p. 10)] via 'modification messages' which contain similar information to our control messages. To our knowledge, dedicated sender key updates are not supported by WhatsApp, as the feature is not mentioned in their white paper [Wha20].

[11] https://github.com/signalapp/libsignal/blob/3b7f3173cc4431bc4c6e55f6182a37229b2db6fd/rust/protocol/src/group_cipher.rs#L43C29-L43C29

Signal group management. At least on Android, Signal has since 2021 required the use of so-called v2 groups, which encompasses Signal's private group management system [CPZ20].[12] Here, Sender Keys is used whenever possible, although occasionally pairwise channels are used as a fallback.[13] The state of the group membership is dictated by the central server; effectively, adds and removes are totally ordered by it. Our control message abstraction captures this latter fact, but our formalism does not capture the private group system. However, our signing of control messages in Sender Keys+ provides more guarantees on group membership given total ordering on control messages: a malicious Signal server can, e.g., re-add removed users without authorisation from group members [CPZ20, p. 45/46].

Updates in Signal. Every sender key must be updated by default every two weeks and a global maximum of 90 days[14]; these are sent via two-party channels, and there is no central control message sent. In this chapter, we opted to model update control messages following the CGKA line of work, for clarity, and because they are particularly well-suited to our improved update mechanism from Section 6.5.2.

6.6 Sender Keys and Sender Keys+: Full Protocols and Security

6.6.1 Protocol Descriptions

We introduce the full Sender Keys and Sender Keys+ protocols in Figures 6.19 to 6.22, extending the descriptions in Section 6.4 and Section 6.5. In Tables A.1 and A.2 (Appendix A.4) we describe the variables and dictionaries used here and in this chapter in general which may be especially useful in this subsection.

Below, we make some additional remarks intended to help the reader to parse the pseudocode which is inherently complex, not least due to the additional variables and logic that is required by our modifications.

Sent and Unsent Sender Keys. When a new user *ID* joins and a member *ME* processes the message via $\mathsf{Proc}(\cdot, T = (\mathtt{add}, \cdot))$, *ID* does not receive the sender key of *ME* until *ID* speaks again. Hence, *ME* needs to keep track of this newly added user; it does so via the $\mathsf{no\text{-}SK}[\cdot]$ dictionary. Namely, $\mathsf{no\text{-}SK}[ID] = \mathtt{true}$ in the view of *ME* if *ME* has not sent his sender key to *ID* yet. This functionality is captured in the SendToMissing algorithm.

Sending Control Messages without a Sender Key. A different scenario is that *ME* calls Exec and generates a control message *T* which, in Sender Keys+, needs to be signed. In the event that *ME* does not have a working sender key yet (e.g., due to a recent removal), *ME* gener-

[12] https://github.com/signalapp/Signal-Android/blob/0775fc7ead818a8380e7e374d15898cbabccaa9c/app/src/main/java/org/thoughtcrime/securesms/jobs/PushGroupSendJob.java#L325C76-L325C76

[13] See the 'for' loop at https://github.com/signalapp/Signal-Android/blob/0775fc7ead818a8380e7e374d15898cbabccaa9c/app/src/main/java/org/thoughtcrime/securesms/messages/GroupSendUtil.java#L249C1-L249C52

[14] https://github.com/signalapp/Signal-Android/commit/35393fc33165e5b1417e7b1a7d6f85d0d7919c6f

Init$(ID, 1^\lambda)$

$\gamma.ME \leftarrow ID$
$\gamma.(\mathsf{ssk}, \boldsymbol{G}, \mathsf{ep}, \mathsf{i}_{ME}, \text{last-kc}) \leftarrow \bot$
$\gamma.\mathsf{SK}[\cdot,\cdot] \leftarrow \bot$ // $(ID, \mathsf{ck}) \rightarrow (\mathsf{spk}_{ID}, \mathsf{ck}_{ID}, \mathsf{i}_{\mathsf{ck}})$
$\gamma.\mathsf{MK}[\cdot,\cdot] \leftarrow \bot$ // $(ID, (\mathsf{kc}, \mathsf{i}_{\mathsf{ck}})) \rightarrow (\mathsf{mk}, \tau\mathsf{k})$
$\gamma.\text{no-}\mathsf{SK}[\cdot] \leftarrow \text{false}$ // $ID \rightarrow$ boolean
$\gamma.\text{max-}\mathsf{i}_{\mathsf{ck}}[\cdot,\cdot] \leftarrow \bot$ // $\mathsf{kc} \rightarrow \mathsf{i}_{\mathsf{ck}}$
$\gamma.\mathsf{kc}[\cdot] \leftarrow \bot$ // $ID \rightarrow \mathsf{kc}$
$\gamma.\mathsf{rs}[\cdot] \leftarrow \bot$ // kc → random coin
$\gamma.\gamma_{2\mathsf{pc}} \leftarrow \mathsf{2PC.Init}(ID)$
return γ

Send(γ, m)

require $ME \in \boldsymbol{G}$
$\vec{C}[\cdot] \leftarrow \bot$
if $\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{i}_{\mathsf{ck}} = \bot$:
  // Sample sender key if needed
  $\vec{C} \leftarrow \mathsf{PreSendFirst}()$
if $\mathsf{i}_{ME} = 0$:
  $\vec{C} \leftarrow (\vec{C}, \mathsf{SendToMissing}())$
$(\mathsf{mk}, \tau\mathsf{k}) \leftarrow (\mathsf{H}_1, \mathsf{H}_3)(\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{ck})$
$\mathsf{ct} \xleftarrow{\$} \mathsf{SymEnc.Enc}(\mathsf{mk}, m)$
$\mathsf{UpdateCK}(ME, \mathsf{kc}[ME])$
$M \leftarrow (\mathsf{ct}, (\mathsf{ep}, \mathsf{i}_{ME}), \mathsf{kc}[ME], \mathsf{i}_{\mathsf{ck}}, ME)$
$\tau \leftarrow \mathsf{MAC.Tag}(\tau\mathsf{k}, M)$
$\sigma \xleftarrow{\$} \mathsf{Sig.Sign}(\mathsf{ssk}, M, \tau)$
return $C := ((M, \tau, \sigma), \vec{C})$

UpdateCK(ID, kc')

$\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{ck} \leftarrow \mathsf{H}_2(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{ck})$
$\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{i}_{\mathsf{ck}} \leftarrow \mathsf{SK}[ID, \mathsf{kc}'].\mathsf{i}_{\mathsf{ck}} + 1$
if $ID = ME$:
  $\mathsf{i}_{ME} \leftarrow \mathsf{i}_{ME} + 1$

Recv$(\gamma, C = ((M, \tau, \sigma), C_{2\mathsf{pc}}))$

parse M **as** $(\mathsf{ct}, (e, i), \mathsf{kc}', \mathsf{i}_{\mathsf{ck}}', ID)$
require $ID \in \boldsymbol{G}$
if $\mathsf{SK}[ID, \mathsf{kc}'] = \bot$:
  // Receive ID's sender key via 2PC if needed
  $((\mathsf{SK}[ID, \mathsf{kc}'], \mathsf{kc}^*, \mathsf{ep}', \text{max-}\mathsf{i}_{\mathsf{ck}}', \text{last-kc}'), ID^*, \cdot, \cdot)$
    $\leftarrow \mathsf{2PC.Recv}(C_{2\mathsf{pc}})$
  require $ID^* = ID \wedge \mathsf{ep}' = e \wedge \mathsf{kc}^* = \mathsf{kc}$
  $\mathsf{DeleteOldCK}(ID, \text{max-}\mathsf{i}_{\mathsf{ck}}', \text{last-kc}')$
else require $C_{2\mathsf{pc}} = \bot$
require $e \leq \mathsf{ep}$
require $\mathsf{Sig.Vrfy}(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{spk}, \sigma, M, \tau)$
$(\mathsf{mk}, \tau\mathsf{k}) \leftarrow \mathsf{UpdateKeysRecv}()$
require $\mathsf{MAC.Vrfy}(\tau\mathsf{k}, M, \tau)$
$m \leftarrow \mathsf{SymEnc.Dec}(\mathsf{mk}, \mathsf{ct})$
return (m, ID, e, i)

UpdateKeysRecv()

$\mathsf{i}_{\mathsf{ck}} \leftarrow \mathsf{SK}[ID, \mathsf{kc}']$
// Store skipped keys in MK given out-of-order delivery
while $\mathsf{i}_{\mathsf{ck}} < \mathsf{i}_{\mathsf{ck}}'$:
  $(\mathsf{mk}, \tau\mathsf{k}) \leftarrow (\mathsf{H}_1, \mathsf{H}_3)(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{ck})$
  $\mathsf{MK}[ID, (\mathsf{kc}', \mathsf{i}_{\mathsf{ck}}')] \leftarrow (\mathsf{mk}, \tau\mathsf{k})$
  $\mathsf{UpdateCK}(ID, \mathsf{kc}')$
  $\mathsf{i}_{\mathsf{ck}}'$++
if $\mathsf{i}_{\mathsf{ck}} > \mathsf{i}_{\mathsf{ck}}'$:
  require $\mathsf{MK}[ID, (\mathsf{kc}', \mathsf{i}_{\mathsf{ck}}')] \neq \bot$
  $(\mathsf{mk}, \tau\mathsf{k}) \leftarrow \mathsf{MK}[ID, (\mathsf{kc}', \mathsf{i}_{\mathsf{ck}}')]$
  // Delete stored key for forward security
  $\mathsf{MK}[ID, (\mathsf{kc}', \mathsf{i}_{\mathsf{ck}}')] \leftarrow \bot$
else :
  $(\mathsf{mk}, \tau\mathsf{k}) \leftarrow (\mathsf{H}_1, \mathsf{H}_3)(\mathsf{SK}[ID', \mathsf{kc}'].\mathsf{ck})$
  $\mathsf{UpdateCK}(ID, \mathsf{kc}')$
return $(\mathsf{mk}, \tau\mathsf{k})$

Figure 6.19: Sender Keys and Sender Keys+ description (part 1 of 4: Init, Send/Recv, two helpers). State variables are implicitly stateful where relevant for readability. Text in black colour corresponds to standard Sender Keys. Coloured text corresponds to the modifications in Sender Keys+ from Section 6.5.2: blue text corresponds to securing control messages via signatures, teal text corresponds to MACing for forward security and violet text corresponds to PCS updates.

ates an ephemeral sender key containing only a signature key spk. The key is immediately distributed over the two-party channels via OneTimeSpk. We remark that whenever both SendToMissing and OneTimeSpk are executed in the same algorithm (such as in Exec), only

Exec(γ, cmd = rem, ID)

1: **require** $ID \in \boldsymbol{G}$
2: $\vec{C}[\cdot] \leftarrow \bot$
3: $T \leftarrow (\texttt{rem}, ME, ID, \mathsf{ep}+1)$
4: $\vec{C} \leftarrow (\mathsf{SendToMissing}(), \mathsf{OneTimeSpk}())$
5: **return** $(T, \sigma := \mathsf{Sig.Sign}(\mathsf{ssk}, T), \vec{C})$

Proc(γ, $(T = (\texttt{rem}, ID, ID', \mathsf{ep}'), \sigma, C_{2\mathsf{pc}})$)

1: **require** $ID \in \boldsymbol{G}$
2: **if** $\mathsf{SK}[ID, \mathsf{kc}[ID]].\mathsf{spk} = \bot$:
3: $((\mathsf{SK}[ID, \mathsf{kc}[ID]], \mathsf{kc}^*, \mathsf{ep}^*, \mathsf{max\text{-}i_{ck}}', \mathsf{last\text{-}kc}'), ID^*, \cdot, \cdot) \leftarrow \mathsf{2PC.Recv}(C_{2\mathsf{pc}})$
4: **require** $ID^* = ID$
5: **require** $\mathsf{ep}' = \mathsf{ep}^* + 1$
6: **require** $\mathsf{kc}^* = \mathsf{kc}[ID]$
7: $\mathsf{DeleteOldCK}(ID, \mathsf{max\text{-}i_{ck}}', \mathsf{last\text{-}kc}')$
8: **else require** $C_{2\mathsf{pc}} = \bot$
9: **require** $\mathsf{Sig.Vrfy}(\mathsf{SK}[ID].\mathsf{spk}, \sigma, T)$
10: **require** $\mathsf{ep}' = \mathsf{ep}+1$
11: $\boldsymbol{G} \leftarrow \boldsymbol{G} \setminus \{ID'\}$
12: $\mathsf{ep} \leftarrow \mathsf{ep}+1$; $\mathsf{i}_{ME} \leftarrow 0$
13: **for all** $ID^+ \in \boldsymbol{G}$:
14: $\mathsf{kc}[ID^+] \leftarrow \mathsf{kc}[ID^+]+1$
15: $\mathsf{SK}[ID', \cdot] \leftarrow \bot$
16: **if** $ID = ME$:
17: $\gamma \leftarrow \bot$
18: **return** true

Exec(γ, cmd = add, ID)

1: **require** $ID \notin \boldsymbol{G}$
2: $\vec{C}[\cdot] \leftarrow \bot$
3: $T \leftarrow (\texttt{add}, ME, ID, \mathsf{ep}+1)$
4: $\vec{C} \leftarrow (\mathsf{SendToMissing}(), \mathsf{OneTimeSpk}())$
5: $\mathsf{welcome} \leftarrow (\boldsymbol{G}, \mathsf{kc}, \mathsf{ep}, \mathsf{spk})$
6: $(\vec{C}[ID], \cdot, \cdot) \xleftarrow{\$} \mathsf{2PC.Send}(ID, \mathsf{welcome})$
7: **return** $(T, \sigma := \mathsf{Sig.Sign}(\mathsf{ssk}, T), \vec{C})$

Proc(γ, $(T = (\texttt{add}, ID, ID' \neq ME, \mathsf{ep}'), \sigma, C_{2\mathsf{pc}})$)

1: **require** $ID \in \boldsymbol{G}$
2: **if** $\mathsf{SK}[ID, \mathsf{kc}[ID]].\mathsf{spk} = \bot$:
3: $((\mathsf{SK}[ID, \mathsf{kc}[ID]], \mathsf{kc}^*, \mathsf{ep}^*, \mathsf{max\text{-}i_{ck}}', \mathsf{last\text{-}kc}'), ID^*, \cdot, \cdot) \leftarrow \mathsf{2PC.Recv}(C_{2\mathsf{pc}})$
4: **require** $ID^* = ID$
5: **require** $\mathsf{ep}' = \mathsf{ep}^* + 1$
6: **require** $\mathsf{kc}^* = \mathsf{kc}[ID]$
7: $\mathsf{DeleteOldCK}(ID, \mathsf{max\text{-}i_{ck}}', \mathsf{last\text{-}kc}')$
8: **else require** $C_{2\mathsf{pc}} = \bot$
9: **require** $\mathsf{Sig.Vrfy}(\mathsf{SK}[ID].\mathsf{spk}, \sigma, T)$
10: **require** $\mathsf{ep}' = \mathsf{ep}+1$
11: $\boldsymbol{G} \leftarrow \boldsymbol{G} \cup \{ID'\}$
12: $\mathsf{ep} \leftarrow \mathsf{ep}+1$; $\mathsf{i}_{ME} \leftarrow 0$
13: $\mathsf{acc} \leftarrow \mathsf{2PC.InitCh}(ID')$
14: **require** acc
15: $\mathsf{no\text{-}SK}[ID'] \leftarrow \mathsf{true}$
16: $\mathsf{kc}[ID'] \leftarrow 0$
17: **return** true

Proc(γ, $(T = (\texttt{add}, ID, ME, \mathsf{ep}'), \sigma, C_{2\mathsf{pc}})$)

1: $((\boldsymbol{G}', \mathsf{kc}', \mathsf{ep}^*, \mathsf{spk}), ID^*, \cdot, \cdot) \leftarrow \mathsf{2PC.Recv}(C_{2\mathsf{pc}})$
2: **require** $ID^* = ID$
3: $\mathsf{SK}[ID, \mathsf{kc}'[ID]].\mathsf{spk} \leftarrow \mathsf{spk}$
4: **require** $\mathsf{Sig.Vrfy}(\mathsf{SK}[ID, \mathsf{kc}'[ID]].\mathsf{spk}, \sigma, T)$
5: **require** $\mathsf{ep}^* + 1 = \mathsf{ep}'$
6: $(\boldsymbol{G}, \mathsf{kc}, \mathsf{ep}) \leftarrow (\boldsymbol{G}', \mathsf{kc}', \mathsf{ep}')$
7: **for all** $ID' \in \boldsymbol{G} \setminus \{ME\}$:
8: $\mathsf{acc} \leftarrow \mathsf{2PC.InitCh}(ID')$
9: **require** acc
10: $\mathsf{kc}[ME], \mathsf{i}_{ME} \leftarrow 0$
11: **return** true

Exec(γ, cmd = crt, IDs)

1: $\boldsymbol{G} \leftarrow IDs$; $\vec{C}[\cdot] \leftarrow \bot$
2: $T \leftarrow (\texttt{crt}, ME, IDs)$
3: $\vec{C} \leftarrow \mathsf{OneTimeSpk}()$
4: **return** $(T, \sigma := \mathsf{Sig.Sign}(\mathsf{ssk}, T), \vec{C})$

Figure 6.20: Sender Keys and Sender Keys+ description (part 2 of 4: Proc/Exec part 1 of 2).

Proc(γ, (T = (crt, ID, IDs), σ, C_{2pc}))

1: **if** SK[ID, kc[ID]].spk = $\perp$:
2: ((SK[ID, kc[ID]], kc*, ep*, ·, ·), ID^*, ·, ·) ← 2PC.Recv(C_{2pc})
3: **require** $ID^* = ID$
4: **require** ep* = kc* = $\perp$
5: **else require** $C_{2pc} = \perp$
6: **require** Sig.Vrfy(SK[ID, kc[ID]].spk, σ, T)
7: **require** ep = $\perp$
8: $\boldsymbol{G} \leftarrow IDs$
9: **for all** $ID' \in \boldsymbol{G}$:
10: acc ← 2PC.InitCh(ID')
11: **require** acc
12: kc[ID'] ← 0
13: ep, i_{ME} ← 0
14: **return** true

Exec(γ, cmd = upd, ME)

1: **require** $ME \in \boldsymbol{G}$
2: Upd-Ind[·] ← $\perp$
3: (spk, ssk) $\xleftarrow{\$}$ Sig.Gen
4: $r \xleftarrow{\$} \{0,1\}^\lambda$
5: Upd-Ind[ME] ← SK[ME, kc[ME]].i_{ck}
6: ck $\xleftarrow{\$} \{0,1\}^\lambda$
7: SK[ME, kc[ME] + 1] ← (spk, ck, i_{ck})
8: m ← (SK[ME, kc[ME] + 1], kc[ME] + 1, ep, max-i_{ck}[last-kc], last-kc, r)
9: $\vec{C}$[·] ← $\perp$
10: **for all** $ID \in \boldsymbol{G} \setminus \{ME\}$:
11: ($\vec{C}$[ID], ·, ·) $\xleftarrow{\$}$ 2PC.Send(ID, m)
12: Upd-Ind[ID] ← SK[ID', kc[ID]].i_{ck}
13: rs[kc[ME]] ← r
14: T ← (upd, ME, kc[ME] + 1, ep + 1, Upd-Ind)
15: **return** (T, σ := Sig.Sign(ssk, T), $\vec{C}$)

Proc(γ, (T = (upd, ID, kc′, ep′, Upd-Ind), σ, C_{2pc}))

1: **require** Sig.Vrfy(SK[ME].spk, σ, T)
2: **if** $ID = ME$:
3: **require** $C_{2pc} = \perp$
4: r ← rs[kc′]; rs[kc′] ← $\perp$
5: **else** :
6: ((SK[ID, kc[ID] + 1], kc*, ep*, max-i_{ck}′, last-kc′, r), ID^*, ·, ·) ← 2PC.Recv(C_{2pc})
7: **require** $ID^* = ID$
8: **require** ep′ = ep* + 1
9: **require** kc* = kc[ID] + 1
10: **require** ep′ = ep + 1
11: ep ← ep′; i_{ME} ← 0
12: kc[ID] ← kc[ID] + 1
13: DeleteOldCK(ID, max-i_{ck}′, last-kc′)
// Hash forward N times before hashing with r
14: **for all** $ID' \in \boldsymbol{G} \setminus \{ID\}$:
15: **if** SK[ID', kc[ID']].ck = $\perp$:
16: **continue** (line 14)
17: **while** SK[ID', kc[ID']].i_{ck} < Upd-Ind[ID'] :
18: (mk, τk) ← (H_1, H_3)(SK[ID', kc[ID']].ck)
19: MK[ID', (kc[ID'], SK[ID', kc[ID']].i_{ck})] ← (mk, τk)
20: UpdateCK(ID', kc[ID'])
21: ℓ ← SK[ID', kc[ID']].i_{ck} − Upd-Ind[ID']
22: **require** $\ell < N$
23: ck′ ← SK[ID', kc[ID']].ck
24: **do** $N - \ell$ times : ck′ ← H_2(ck′)
25: SK[ID', kc[ID']].ck ← F(ck′, r)
26: **return** true

Figure 6.21: Sender Keys and Sender Keys+ description (part 3 of 4: Proc/Exec part 2 of 2).

one of them will output a non-blank ciphertext, depending on whether the caller's signature key SK[ME, kc[ME]].spk exists or not.

Index Updates. Most of the protocol logic behind our new update mechanism is explained in Section 6.5. Due to the synchronisation issues mentioned there, the update initiator sends his view (message epoch) of everyone else's sender key. This information is stored in the Upd-Ind[·] dictionary, which is sent as part of the control message.

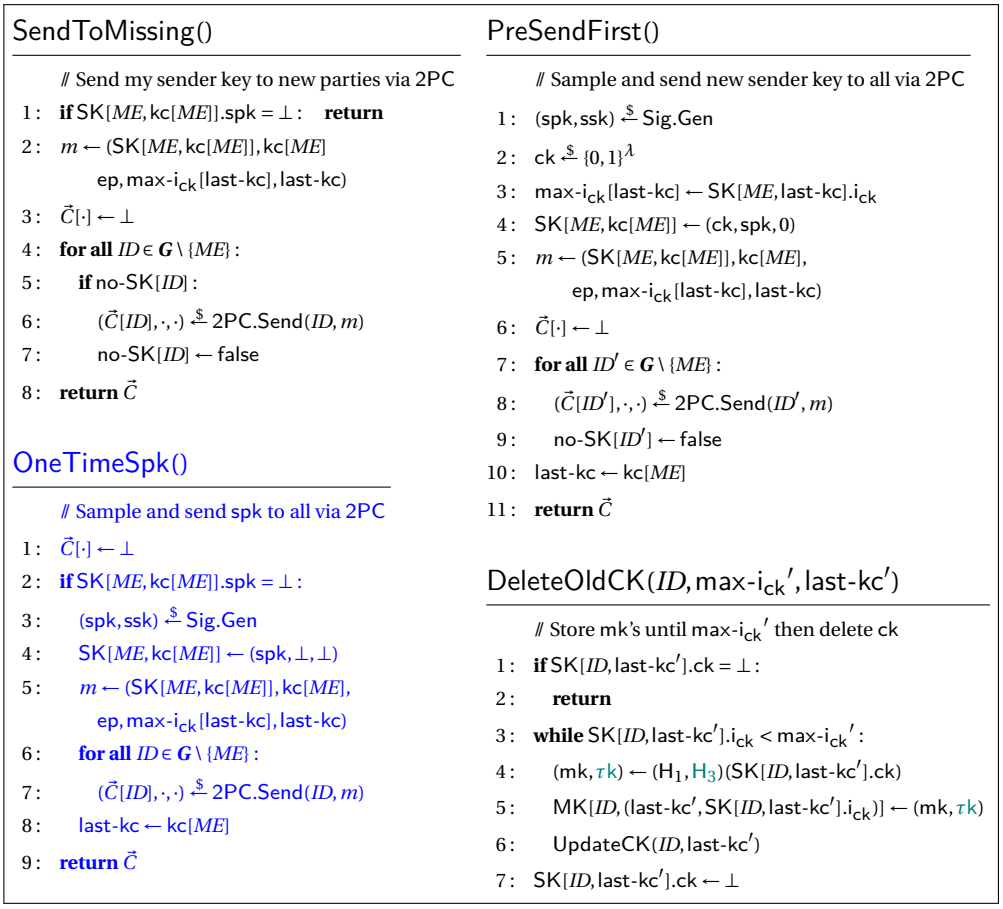

Figure 6.22: Sender Keys and Sender Keys+ description (part 4 of 4: remaining helpers).

Additional State Variables. The state variables max-i$_{ck}$[·] and last-kc were omitted in Figure 6.9. Essentially, these variables keep track of the maximum index max-i$_{ck}$[*ID*, kc], corresponding to the sender key SK[*ID*, kc], for which a message was sent. This is critical to determine what skipped message keys (if any) should be stored in MK so that chain keys can eventually be deleted for forward security. This synchronisation mechanism occurs in the Recv algorithm via the two-party channels, where last-kc specifies the last key counter the bound max-i$_{ck}$ refers to.

Correctness. For completeness, we observe here that Sender Keys and Sender Keys+ are correct as in Definition 48. The argument, which we sketch below, proceeds by a similar case analysis to what we did for IAS and DGS in Section 5.3, except here we argue with respect to the relevant winning conditions of Figure 6.6:

- *Message delivery*: By construction of Sender Keys/+ and the correctness of 2PC, for a given message epoch, all active group members will derive the same message key, from which this follows from the correctness of SymEnc (note we also must invoke the

correctness of other primitives here and below, e.g., the fact that all Sig.Vrfy checks pass).

- *Group evolution*: This follows from the consistency between how state variable $\boldsymbol{G}$, by construction of Exec and Proc, and dictionary G in Figure 6.6 are updated.
- *Group membership consistency*: This follows similarly to the previous point.
- *Out-of-order delivery*: Messages from past epochs or future indices in the current epoch are handled correctly by construction of UpdateKeysRecv which stores skipped message keys (as well as the correctness of other protocol components like 2PC as above). Ciphertexts from future epochs will always be rejected since the caller of Recv compares their local variable ep with epoch value e attached to each ciphertext output by Send.

6.6.2 Sender Keys Security

We here prove Theorem 20, i.e., the security of our core Sender Keys protocol. Let $\mathscr{A}$ be an adversary against the Sender Keys protocol that plays the GM message indistinguishability game $\mathsf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}$ (Figure 6.7) with respect to cleanness predicate $\mathsf{C}_{\mathsf{gm}} = \mathsf{C}^{\Delta}_{\mathsf{sk}}$ (Figure 6.13), and let $q_1,\ldots,q_q$ be the oracle queries of $\mathscr{A}$ in a given execution. The proof follows a series of hybrid games, where Game Γ_0 is the original game in Figure 6.7.

Exposed Keys and Key Sequences. Before diving into the details, we characterise the set of exposed chain keys $\mathsf{ExpKeys}_{\mathsf{ck}}$ as those keys that can be (trivially) derived by the adversary following its state exposure queries. To this end, we observe that all chain keys (and also message keys) generated during protocol execution are uniquely identified by three parameters: the epoch number e, the key index i, and the owner ID; we label them as $\mathsf{ck}^{e,i}_{ID}$ and $\mathsf{mk}^{e,i}_{ID}$. Given a user ID, its chain keys form *sequences* such that the key in message epoch $(e,i+1)$ is deterministically derived from the key in (e,i) as $\mathsf{ck}^{e,i+1}_{ID} = \mathsf{H}_1(\mathsf{ck}^{e,i}_{ID})$ in the chain. Formally, let $q_i = \mathsf{DELIVER}(ID,(T_{\mathsf{core}},C_{2\mathsf{pc}}))$ where T_{core} is either: a remove message for $ID' \neq ID$, a create message, a message that adds ID to the group, or an update message for ID, and $q_j = \mathsf{DELIVER}(ID,(T'_{\mathsf{core}},C'_{2\mathsf{pc}}))$ where T'_{core} is either the next remove message for any $ID' \in \boldsymbol{G}$ for $\boldsymbol{G}$ from the perspective of ID before q_i where $j > i$, or an update message for ID; otherwise, $q_j = q_{q+1}$ (where $\mathsf{E}[ID;q_{q+1}]$ denotes the state of $\mathsf{E}[ID]$ *after* query q_q) if no such query was made. Let also $e = \mathsf{E}[ID;q_i]$ and $\tilde{e} = \mathsf{E}[ID;q_j]$. Then,

$$\mathsf{ck}^{e,0}_{ID},\ldots,\mathsf{ck}^{e,i_e}_{ID},\mathsf{ck}^{e+1,0}_{ID},\ldots,\mathsf{ck}^{\tilde{e}-1,i_{\tilde{e}}}_{ID}$$

is the chain key sequence for ID in epochs e to $\tilde{e}$. Note since $\mathsf{ck}^{e,0}_{ID}$ is the first key generated after a removal or update that $\mathsf{ck}^{e,0}_{ID}$ is generated using fresh randomness and distributed to all parties via two-party channels.

For every $q_j = \mathsf{EXPOSE}(ID_j)$ query such that $(e_j,i_j) \leftarrow \mathsf{m\text{-}ep}(ID_j,ID',q_j)$, all the chain keys of ID' that are exposed are exactly those in the chain key sequence of ID' starting from epoch

(e_j, i_j). If we denote this set by $\mathsf{EpCK}^{(j)}_{ID'}$, then we have that

$$\mathsf{ExpKeys}_{\mathsf{ck}} = \bigcup_{j=1}^{k} \left\{ \mathsf{ck}^{e,i}_{ID'} : ID' \in \boldsymbol{G}[ID_j; q_{i_j}] \wedge (e,i) \in \mathsf{EpCK}^{(j)}_{ID'} \right\}.$$

where $\boldsymbol{G}[ID_j; q_{i_j}] = ID_j.\gamma.\boldsymbol{G}$ represents the view of the group of ID_j at the time of query q_{i_j}. Note that, for any group member, if one of its keys in a chain key sequence is in $\mathsf{ExpKeys}_{\mathsf{ck}}$, all the subsequent keys until either a removal or update for ID_j is processed are also in $\mathsf{ExpKeys}_{\mathsf{ck}}$.

Hybrid Games. We define the main sequence of games below. We then bound the corresponding advantages by a series of lemmas.

Game Γ_0: This is the original M-IND game, parameterised by cleanness predicate $\mathsf{C}_{\mathsf{gm}} = \mathsf{C}^{\Delta}_{\mathsf{sk}}$.

Game Γ_1: In this game, we remove the **return** b conditions whenever $\mathsf{DELIVER}(ID, (T_{\mathsf{core}}, C_{2\mathsf{pc}}))$ or $\mathsf{RECEIVE}(ID, (C_{\mathsf{core}}, C_{2\mathsf{pc}}))$ are called such that $C_{2\mathsf{pc}}$ was not previously output in some previous oracle query that outputs a ciphertext or control message.

Game Γ_2: In this game, we remove the **return** b condition in the RECEIVE and DELIVER oracles, so that they always returns nothing. Hence, the adversary can no longer win the game by injecting.

Game Γ_3: In this game, we allow a single call to the CHAL oracle, as opposed to arbitrarily many calls.

Game Γ_4: In this game, all chain keys and corresponding message keys in ID's key sequence that includes ck such that $\mathsf{H}_2(\mathsf{ck}) = \mathsf{mk}$, where mk is the message key used in the underlying Send call in the $\mathsf{CHAL}(ID, \cdot, \cdot)$ query (if it exists), are replaced by uniformly random values. In addition, all 2PC ciphertexts that transmit the chain key ck or keys earlier in the key sequence leading to ck are replaced with encryptions of 0^{ℓ} where ℓ is the length of the message encrypted.

To complete the proof, Game Γ_4 is simulated by an IND-CPA SymEnc adversary. In the lemmas hereafter, we assume that an adversary $\mathscr{B}$ simulating a hybrid for $\mathscr{A}$ can efficiently determine whether $\mathscr{A}$ has violated the cleanness predicate, and aborts execution since $\mathscr{B}$ can no longer win.

Lemma 17. There exists an adversary $\mathscr{B}_1$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g0}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g1}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + 2 \cdot \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{2\mathsf{pc}},\Delta}(\mathscr{B}_1).$$

Proof. We proceed by constructing $\mathscr{B}_{1,b'}$, a $\mathsf{2PC\text{-}IND}_{\mathsf{2PC},b,\mathsf{C}_{2\mathsf{pc}},\Delta}$ adversary that simulates Game Γ_0/Game Γ_1 for adversary $\mathscr{A}$ depending on $\mathsf{2PC\text{-}IND}_{\mathsf{2PC},b,\mathsf{C}_{2\mathsf{pc}},\Delta}$ bit b and $\mathsf{M\text{-}IND}_{\mathsf{GM},b',\mathsf{C}_{\mathsf{gm}}}$ bit b'. At a high level, $\mathscr{B}_{1,b'}$ simulates all $\mathsf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}$ oracle queries using

its own oracles and otherwise simulating locally, except every $\mathsf{CHAL}(ID, m_0, m_1)$ is simulated as if the challenger's bit is b'. In more detail, $\mathscr{B}_{1,b'}$ replaces all:

- $\mathsf{2PC.InitCh}(ID')$ calls from ID by the output of query $\mathsf{INIT\text{-}CH}(ID, ID')$.
- $\mathsf{2PC.Send}(m, ID')$ calls from ID by the output of $\mathsf{SEND}(ID, ID', m)$.
- $\mathsf{2PC.Recv}(C)$ calls from ID as follows: $\mathscr{B}_{1,b'}$ first calls $\mathsf{RECEIVE}(ID, C)$, which outputs $(b, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$. If $b \neq \bot$, $\mathscr{B}_{1,b'}$ returns b to its challenger and stops simulating. Otherwise, by cleanness and construction of Sender Keys (argued below), C must have been previously output by a $\mathsf{SEND}(ID, ID', m)$ call. In this case, $\mathscr{B}_{1,b'}$ thus replaces the Recv call with $(m, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$.

In addition:

- If $\mathscr{A}$ calls $\mathsf{EXPOSE}(ID)$, $\mathscr{B}_{1,b'}$ uses the output of its own $\mathsf{EXPOSE}(ID)$ call and its state from locally simulating to respond to $\mathscr{A}$'s query.
- Finally, $\mathscr{B}_{1,b'}$ outputs the same bit as $\mathscr{A}$.

Note that by construction of Sender Keys, after 2PC.Recv is called in Recv and Proc calls, the output is checked so it is "appropriate" for the context it is called in (i.e., it is consistent with the received C_{core} or T_{core} (e.g., by checking ID and epoch matches with the input C)). In addition, in a given Sender Keys Recv/Proc call that does *not* invoke 2PC.Recv, $C_{\mathsf{2pc}} = \bot$ is enforced, preventing GM forgeries that include an arbitrary C_{2pc} value that is simply ignored. Thus, if $\mathscr{B}_{1,b'}$ outputs $(b, \ldots)$ from RECEIVE or DELIVER such that $b \neq \bot$, it must be that a valid 2PC forgery was made.

Using the triangle inequality, we have

$$\begin{aligned}
\mathsf{Adv}^{\mathrm{g0}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) &= \left|\Pr[G^1_0 \Rightarrow 1] - \Pr[G^0_0 \Rightarrow 1]\right| \\
&\leq \left|\Pr[G^1_0 \Rightarrow 1] - \Pr[G^1_1 \Rightarrow 1]\right| + \left|\Pr[G^0_0 \Rightarrow 1] - \Pr[G^0_1 \Rightarrow 1]\right| \\
&\quad + \left|\Pr[G^1_1 \Rightarrow 1] - \Pr[G^0_1 \Rightarrow 1]\right| \\
&\leq \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B}_{1,1}) + \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B}_{1,0}) + \mathsf{Adv}^{\mathrm{g1}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A})
\end{aligned}$$

where the inequality $\left|\Pr[G^{b'}_0 \Rightarrow 1] - \Pr[G^{b'}_1 \Rightarrow 1]\right| \leq \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B}_{1,b'})$ holds because the simulation is perfect given RECEIVE never outputs $(b, \ldots)$ with $b \neq \bot$, and when RECEIVE does output such a $(b, \ldots)$, $\mathscr{B}_{1,b'}$'s advantage is at least as large as $\mathscr{A}$'s since in this case $\mathscr{B}_{1,b'}$ always outputs the correct bit. The result follows. □

Lemma 18. There exists an adversary $\mathscr{B}_2$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g1}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + q \cdot \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{B}_2).$$

Proof. Let E be the event that $\mathscr{A}$ in a clean execution of Game Γ_1 (i.e., when cleanness predicate C evaluates to true) calls $\mathsf{RECEIVE}(ID,(C_{\mathsf{core}},C_{2\mathsf{pc}}))$ that outputs $m \neq \perp$ for some ID such that $(C_{\mathsf{core}},\vec{C})$ was not previously output by SEND, where $C_{2\mathsf{pc}}$ is some (correct) 2PC ciphertext by definition of Game Γ_1. Observe first that Game Γ_1 and Game Γ_2 are identical given $\neg E$. Note that at most q signature keys are sampled and sent over the two-party channels during the game's execution: the first Send call in an epoch and the (single) $\mathsf{Exec}(\cdot,\mathtt{crt},IDs)$ call result in one signature key being sampled in each call.

Then, let E_i be the event that $\mathscr{A}$'s first call to $\mathsf{RECEIVE}$ with argument $(ID,(C_{\mathsf{core}},C_{2\mathsf{pc}}))$ satisfying the conditions of E is such that the internal Recv call outputs ID' corresponding to the i-th signature key sampled by the challenger (i.e., the i-th sender key).

We define $\mathsf{SUF\text{-}CMA}$ adversary $\mathscr{B}_{ID',e}$ who simulates for Game Γ_1 adversary $\mathscr{A}$ given E_i holds. $\mathscr{B}_i$ simulates as follows. $\mathscr{B}_i$ locally simulates for $\mathscr{A}$ and responds to all of $\mathscr{A}$'s queries except queries involving the i-th sender key sampled: let ID be the key holder. For these queries, ID sets spk in variable SK to the $\mathsf{SUF\text{-}CMA}$ public key pk. $\mathscr{B}_i$ generates all signatures associated with spk via $\mathsf{SUF\text{-}CMA}$ oracle SIGN.

Finally, consider when $\mathscr{A}$ makes their first query of the form $\mathsf{RECEIVE}(ID,C)$ that returns $m \neq \perp$ that was not previously output by SEND. Observe that $C = (C_{\mathsf{core}} = (M,\sigma),C_{2\mathsf{pc}})$ and $M = (c,(e,i),\mathsf{kc}',\mathsf{i_{ck}}',ID)$ where σ is a signature on $(c,(e,i),\mathsf{i_{ck}},ID)$. As forgeries on $C_{2\mathsf{pc}}$ are disallowed, only C_{core} can possibly be the source of the forgery. By construction of Recv, $(c,(e,i),\mathsf{kc}',\mathsf{i_{ck}}',ID)$ must be different from values previously input to Recv for a non-bottom value to be output, and, by definition of event E_i, C_{core} must differ from values previously output by SEND. Moreover, by cleanness, $\mathscr{A}$ must not have been able to make a state exposure that enables it to access signature secret key ssk. Thus, the simulation is well-defined and signature σ is a valid forgery, and so $\mathscr{B}_i$ extracts σ from C and returns (M,σ) to its $\mathsf{SUF\text{-}CMA}$ challenger.

Finally, we have

$$\begin{aligned}
\mathsf{Adv}^{\mathrm{g1}}_{\mathsf{GM},\mathsf{C_{gm}}}(\mathscr{A}) &= \left|\Pr[G_1^1 \Rightarrow 1 \wedge \neg E] - \Pr[G_1^0 \Rightarrow 1 \wedge \neg E] + \Pr[G_1^1 \Rightarrow 1 \wedge E] - \Pr[G_1^0 \Rightarrow 1 \wedge E]\right| \\
&\leq \mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C_{gm}}}(\mathscr{A}) + \left|\Pr[G_1^1 \Rightarrow 1 \wedge E] - \Pr[G_1^0 \Rightarrow 1 \wedge E]\right| \\
&\leq \mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C_{gm}}}(\mathscr{A}) + \Pr[E] \\
&\leq \mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C_{gm}}}(\mathscr{A}) + \sum_i \Pr[E_i] \\
&\leq \mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C_{gm}}}(\mathscr{A}) + \sum_i \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{B}_i) \\
&\leq \mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C_{gm}}}(\mathscr{A}) + q \cdot \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{B}_2),
\end{aligned}$$

where the last step holds by combining each $\mathscr{B}_i$ into $\mathscr{B}_2$. □

Lemma 19. There exists an adversary $\mathscr{B}_3$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq q_{\mathsf{chal}} \cdot \mathsf{Adv}^{\mathrm{g3}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{B}_3)$$

where $q_{\mathsf{chal}} \leq q$ denotes the number of CHAL oracle queries made by $\mathscr{A}$.

Proof. We adopt the same high-level strategy as the proof of Lemma 6 in [ACDT19] (conference version [ACDT20]). Let H_0 be exactly Game Γ_2 with $b = 0$. For $i \in [1, q_{\mathsf{chal}}]$, let H_i be exactly H_{i-1} except the i-th query $\mathsf{CHAL}(ID, m_0, m_1)$ uses m_1 (i.e., acts as if the challenge bit is $b = 1$). Observe first that we have $\mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) = \left|\Pr[H_0 \Rightarrow 1] - \Pr[H_{q_{\mathsf{chal}}} \Rightarrow 1]\right|$. We will show, for $i \in [1, q_{\mathsf{chal}}]$, that there exists adversary $\mathscr{B}_{3,i}$ playing Game Γ_2 such that $|\Pr[H_i \Rightarrow 1] - \Pr[H_{i-1} \Rightarrow 1]| = \mathsf{Adv}^{\mathbf{g3}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{B}_{3,i})$. The claimed result then follows by applying the sequence of hybrids and the triangle inequality.

$\mathscr{B}_{3,i}$ simulates as follows. For $\mathscr{A}$'s first $i-1$ $\mathsf{CHAL}(ID, m_0, m_1)$ calls, $\mathscr{B}_{3,i}$ calls $\mathsf{SEND}(ID, m_1)$ and returns the result. For $\mathscr{A}$'s ith $\mathsf{CHAL}(ID, m_0, m_1)$ call, $\mathscr{B}_{3,i}$ calls $\mathsf{CHAL}(ID, m_0, m_1)$ and returns the result. For $\mathscr{A}$'s subsequent $\mathsf{CHAL}(ID, m_0, m_1)$ calls, $\mathscr{B}_{3,i}$ calls $\mathsf{SEND}(ID, m_0)$ and returns the result. If $\mathscr{A}$ ever makes an EXPOSE query that would trivially allow for them to decrypt any challenge ciphertext, or has previously called EXPOSE such that the resulting CHAL query would be trivially decryptable, $\mathscr{B}_{3,i}$ aborts. Note that this condition can be efficiently determined based on $\mathscr{A}$'s oracle queries. $\mathscr{B}_{3,i}$ processes all other queries using its own oracles.

Note that if $\mathscr{A}$'s (multi-challenge) execution satisfies the cleanness predicate, then so too does $\mathscr{B}_{3,i}$'s. To see this, note that $\mathscr{B}_{3,i}$ and $\mathscr{A}$ make the same queries to all oracles except for CHAL and SEND. In particular, since $\mathscr{B}_{3,i}$ makes the same EXPOSE queries as $\mathscr{A}$ (given $\mathscr{B}_{3,i}$ does not abort) and less CHAL queries, there are the same or possibly less opportunities for the challenge predicate to fail in $\mathscr{B}_{3,i}$'s execution as compared to $\mathscr{A}$'s. Moreover, the additional SEND queries that $\mathscr{B}_{3,i}$ makes do not affect any predicates. Thus, if $\mathscr{B}_{3,i}$'s challenge bit b is 0, then $\mathscr{B}_{3,i}$ perfectly simulates H_{i-1}, and similarly $\mathscr{B}_{3,i}$ perfectly simulates H_i given $b = 1$. The result follows. □

Lemma 20. There exists an adversary $\mathscr{B}_4$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g3}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq q \cdot (\mathsf{Adv}^{\mathrm{g4}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B}_4) + q \cdot \mathsf{Adv}^{\mathrm{prg}}_{\mathsf{H}}(\mathscr{B}_4))$$

Proof. The proof of this lemma proceeds via hybrid sub-games. Consider the (restricted) chain key sequence in an execution of Game Γ_3 starting from epoch e until key epoch (ID, e', i') corresponding to the output of Send in the $\mathsf{CHAL}(ID, \cdot, \cdot)$ call, if it exists. If it does not exist, then Game Γ_3 adversary $\mathscr{A}$ has no advantage as their execution is independent of the challenge bit. Otherwise, we replace all chain keys and their corresponding message keys in this sequence by uniformly random values. Note that, in the protocol in Figures 6.19 to 6.22, we model H_1 and H_2 as a PRG $\mathsf{H} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$. Namely, $\mathsf{H}(\mathsf{ck}_i) = (\mathsf{ck}_{i+1}, \mathsf{mk}_i)$ outputs an updated chain

key and a new message key.

Let E_{ck} be the event where call $\mathsf{CHAL}(ID,\cdot,\cdot)$ is made such that the underlying Send call uses message key mk iteratively derived from some ck, where $\mathsf{ck} = \mathsf{ck}_{ID}^{e,0}$, for some e, is the start of ID's corresponding chain key sequence. Observe that at most q such chain key sequences are possible at the start of an execution of Game Γ_3 where $\mathscr{A}$ makes q oracle queries, and each such ck is associated with a corresponding ID (the user who sampled ck). Denote for simplicity this chain key sequence by $\mathsf{ck}_1,\ldots,\mathsf{ck}_m$ (where $\mathsf{ck}_1 := \mathsf{ck}_{ID}^{e,0}$, $\mathsf{ck}_m := \mathsf{ck}_{ID}^{e',i'}$ and $m \leq q$). For each possible E_{ck}, we can construct a sequence of hybrids H_i for $i = -1,0,\ldots,m$ as follows. Game H_{-1} is as in the original Game Γ_3. Game H_0 differs from Game H_{-1} in that all two-party ciphertexts encrypting ck or ancestors in its key sequence are replaced with encryptions of dummy message 0^ℓ for messages of length ℓ. For $i \geq 1$, Game H_{i-1} differs from Game H_i in that, in the latter, we replace both mk_{i-1} and ck_i by uniformly random $r_{i-1} \xleftarrow{\$} \mathscr{K}$ and $s_i \xleftarrow{\$} \mathscr{W}$, respectively. Finally, Game H_m is Game Γ_4; all non-exposed keys are independent.

We first construct $\mathsf{2PC\text{-}IND}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ adversary $\mathscr{B}$ that simulates for adversary $\mathscr{A}$ playing (as we will argue) Game H_{-1} or H_0 depending on its challenger's bit. $\mathscr{B}$ simulates similarly to $\mathscr{B}_1$ in the proof of Lemma 17 except when simulating $\mathsf{2PC.Send}(m,ID')$ calls. Here, instead of replacing all such calls with the output of $\mathsf{SEND}(ID,ID',m)$, $\mathscr{B}$ replaces calls that encrypt ck or its key sequence ancestors in m with the output of $\mathsf{CHAL}(ID,ID',m,0^{|m|})$. $\mathscr{B}$ otherwise simulates identically. Observe that in a clean execution of H_{-1}, chain key ck_1 must not be exposed, and that $\mathscr{B}$, who is parametrised by ck, can deduce exactly which $\mathsf{2PC.Send}$ calls to replace with a CHAL call. It follows that $\mathscr{B}$ simulates H_{-1} given the challenge bit is 0 and H_0 given it is 1.

Note in a clean execution of H_0 that the starting key in the chain $\mathsf{ck}_{ID}^{e,0} \notin \mathsf{ExpKeys}_{\mathsf{ck}}$ is generated by ID using fresh randomness. Besides, $\mathsf{ck}_{ID}^{e,0}$ is only sent over two-party channels that contains no information about the key due to the previous hop (noting in a clean execution that the channels must have healed if previously compromised), so it is hidden from $\mathscr{A}$. Now, let $\mathscr{A}$ be an adversary that interpolates between any two Games H_{i-1} and H_i. Then, we can create an adversary $\mathscr{B}$ against PRG indistinguishability from $\mathscr{A}$ as follows. Since by induction in H_{i-1} the seed ck_{i-1} of the PRG is a uniformly random value, $\mathscr{B}$ simply embeds a PRG challenge in mk_{i-1} and ck_i (recall that we consider a PRG with an expansion factor of 2 such that $\mathsf{H}(\mathsf{ck}_{i-1}) = (\mathsf{ck}_i,\mathsf{mk}_{i-1})$. Then, $\mathscr{B}$ simulates the rest of the game locally and returns the guess of $\mathscr{A}$.

It follows that the simulation is perfect. By considering all events E_{ck} and the union bound, it follows that there exists $\mathscr{B}_4$ so that:

$$\mathsf{Adv}^{\mathrm{g3}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq q \cdot (\mathsf{Adv}^{\mathrm{g4}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B}_4) + q \cdot \mathsf{Adv}^{\mathrm{prg}}_{\mathsf{H}}(\mathscr{B}_4))$$

□

Lemma 21. There exists an adversary $\mathscr{B}$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g4}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathsf{SymEnc}}(\mathscr{B})$$

Proof. We reduce to the security of the encryption scheme. Let $\mathscr{A}$ be an adversary against Game Γ_4. Then, we can build an adversary $\mathscr{B}$ against the IND-CPA security of the encryption scheme. Let b^* be the (hidden) bit that parameterises the IND-CPA game of $\mathscr{B}$. Then, $\mathscr{B}$ simulates Game Γ_4 for $\mathscr{A}$ except for the challenge query $q^* = \mathsf{CHAL}(ID^*, m_0, m_1)$, where it proceeds as follows. $\mathscr{B}$ receives m_0, m_1 from $\mathscr{A}$ and forwards them to the IND-CPA challenger, who outputs a ciphertext c^*. Then, $\mathscr{B}$ crafts a ciphertext C^* as if it originated from ID^* and sends it to $\mathscr{A}$. The simulation continues until the game finishes, and $\mathscr{B}$ returns the same guess b' as $\mathscr{A}$.

As the message key mk used to encrypt the challenge message in the original Game Γ_4 is a uniformly random key, as we argued above, the simulation is perfect. Hence, the lemma follows. □

Finally, Theorem 20 follows by combining the sequence of hybrids above.

Proof Strategy for Corollary 1. The hybrids are defined similarly except that they differ in the definition of Game Γ_1 and Game Γ_4. Let the resulting sequence of games be denoted Game Γ_1', Game Γ_2', Game Γ_3', Game Γ_4'. In Game Γ_1', all two-party channel ciphertexts that cannot be trivially decrypted by the adversary are replaced with encryptions of dummy strings of the form 0^ℓ. Game Γ_4' differs as in Game Γ_4 except that 2PC ciphertexts are not changed. We can then essentially directly use the above lemmas except for Lemmas 17 and 20:

- For Game Γ_1', note that since $\mathsf{2PC\text{-}IND}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ adversary $\mathscr{B}_{1,b'}$ is given all of $\mathscr{A}$'s queries in advance, it can efficiently deduce which 2PC.Send queries to replace with CHAL and SEND depending on which ciphertexts can be trivially exposed by $\mathscr{A}$ or not. It then follows that

$$\mathsf{Adv}^{\mathrm{g0}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g1'}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + 2 \cdot \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B}_1)$$

- For Game Γ_4', the reduction no longer needs to guess ck, since this information can be efficiently derived from the sequence of queries $q_1, \ldots, q_q$ initially given to $\mathscr{A}$. A sequence of hybrids H_i for $i \geq 0$ can then be directly constructed; note we can ignore the hop between H_{-1} and H_0 since Game Γ_1 already handles this. It then follows that

$$\mathsf{Adv}^{\mathrm{g3'}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g4'}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + q \cdot \mathsf{Adv}^{\mathrm{prg}}_{\mathsf{H}}(\mathscr{B}_4)$$

The result follows.

6.6.3 Sender Keys+ Security

In this subsection, we prove the security of our Sender Keys+ protocol. We do so with respect to the modified cleanness predicate in Figure 6.18. The proof follows similar steps as the proof for Theorem 20.

From our Sender Keys+ protocol, we model $\mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3$ as a PRG $\mathsf{H} : \{0,1\}^\lambda \to \{0,1\}^{3\lambda}$, and the KDF F used for the update operation as a dual PRF. Note the proof would still work with respect to the Sender Keys predicate $\mathsf{C}^\Delta_{\mathsf{sk}}$ (Figure 6.13) as the new predicate $\mathsf{C}^\Delta_{\mathsf{sk+}}$ (Figure 6.18) is strictly less restrictive. Towards the proof, we re-define the notion of key sequences introduced in Section 6.6.2 to capture our new update mechanism. We now consider chain key sequences for ID starting with $\mathsf{ck}^{e,0}_{ID}$ where zero or more update operations from $ID' \neq ID$ are applied to a chain key of the form $\mathsf{ck}^{e',i'}_{ID}$. In addition, updates from parties $ID \neq ID'$ now result in a new key sequence. We state the theorem below.

Theorem 21. Let $\mathsf{SymEnc} := (\mathsf{Enc}, \mathsf{Dec})$ be a $(q, \epsilon_{\mathsf{sym}})$-$\mathsf{IND\text{-}CPA}_{\mathsf{SymEnc}}$ symmetric encryption scheme, $\mathsf{Sig} := (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ a $(q, \epsilon_{\mathsf{sig}})$-$\mathsf{SUF\text{-}CMA}_{\mathsf{Sig}}$ signature scheme, $\mathsf{H} : \{0,1\}^\lambda \to \{0,1\}^\lambda \times \{0,1\}^\lambda$ a $(q, \epsilon_{\mathsf{prg}})$-$\mathsf{PRG}_{\mathsf{H}}$ function, F a $(q, \epsilon_{\mathsf{dprf}})$-$\mathsf{dPRF}$ function, MAC a $(q, \epsilon_{\mathsf{mac}})$-$\mathsf{SUF\text{-}CMA}_{\mathsf{MAC}}$ message authentication code and $\mathsf{2PC}$ a $(q, \epsilon_{\mathsf{2pc}})$-$\mathsf{2PC\text{-}IND}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}$ two-party channels scheme for PCS bound $\Delta > 0$. Then Sender Keys+ (Figures 6.19 to 6.22) is

$$(q, 2 \cdot \epsilon_{\mathsf{2pc}} + q^3 \cdot (\epsilon_{\mathsf{2pc}} + \epsilon_{\mathsf{sym}} + N \cdot q \cdot \epsilon_{\mathsf{prg}} + q \cdot \epsilon_{\mathsf{dprf}} + q \cdot \epsilon_{\mathsf{mac}}) + q \cdot \epsilon_{\mathsf{sig}})\text{-}\mathsf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}$$

with respect to predicate $\mathsf{C}_{\mathsf{gm}} = \mathsf{C}^\Delta_{\mathsf{sk+}}$ (Figure 6.18) and concurrency bound N, where the two-party channels predicate $\mathsf{C}_{\mathsf{2pc}}$ is defined in Figures 6.4 and 6.5.

Hybrid Games. We define the main sequence of games below.

Game Γ_0: This is the original M-IND game, parameterised by cleanness predicate $\mathsf{C}_{\mathsf{gm}} = \mathsf{C}^\Delta_{\mathsf{sk+}}$.

Game Γ_1: In this game, we remove the **return** b conditions whenever $\mathsf{DELIVER}(ID, (T_{\mathsf{core}}, C_{\mathsf{2pc}}))$ or $\mathsf{RECEIVE}(ID, (C_{\mathsf{core}}, C_{\mathsf{2pc}}))$ are called such that C_{2pc} was not previously output in some previous oracle query that outputs a ciphertext or control message.

Game Γ_2: In this game, we completely remove the **return** b condition in the DELIVER oracle and in the RECEIVE oracle for forgeries that occur when predicate $\mathsf{C}^\Delta_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}})$ is true for ciphertexts of the form C_{core}.

Game Γ_3: In this game, we allow a single call to the CHAL oracle, as opposed to arbitrarily many calls.

Game Γ_4: In this game, all chain keys and corresponding message and MAC keys in ID's key sequence that includes ck such that $\mathsf{H}_2(\mathsf{ck}) = \mathsf{mk}$ and $\mathsf{H}_3(\mathsf{ck}) = \tau\mathsf{k}$, where mk (resp. $\tau\mathsf{k}$) is the message key (resp. MAC key) used in the underlying Send call in the $\mathsf{CHAL}(ID, \cdot, \cdot)$ query (if it exists), are all replaced by uniformly random values. In addition, all 2PC ciphertexts that

transmit the chain key ck or keys earlier in the key sequence leading to ck are replaced with encryptions of 0^ℓ where ℓ is the length of the message encrypted.

Game Γ_5: In this game, we completely remove the **return** b condition in the RECEIVE oracles. Hence, the adversary cannot win the game by injecting. To complete the proof, Game 5 is simulated by an IND-CPA SymEnc adversary.

Lemma 22. There exists an adversary $\mathscr{B}_1$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g0}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g1}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + 2 \cdot \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B}_1).$$

Proof. The proof is essentially identical to that of Lemma 17 so we omit it. □

Lemma 23. There exists an adversary $\mathscr{B}_2$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g1}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + q \cdot \mathsf{Adv}^{\mathrm{suf-cma}}_{\mathsf{Sig}}(\mathscr{B}_2).$$

Proof. The proof follows the same high-level idea as for Lemma 18. That is, we consider events E_i for $i \in [1, q']$ for some $q \leq q$ such that the first successful forgery is made using the i-th signature key pair sampled. For this proof, we consider forgeries now over both RECEIVE and DELIVER rather than just RECEIVE. By construction of Sender Keys+, DELIVER forgeries given $\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}})$ is true only occur as a result of a signature forgery. Thus, by a very similar reduction to Lemma 18, the result follows. Note that unlike in the proof of Theorem 20 at this point, we have not yet completely disallowed injections on RECEIVE: we still allow forgeries that are permitted by $\mathsf{C}^{\Delta}_{\mathsf{sk+}}$ but disallowed by $\mathsf{C}^{\Delta}_{\mathsf{sk}}$. □

Lemma 24. There exists an adversary $\mathscr{B}_3$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g2}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq q_{\mathsf{chal}} \cdot \mathsf{Adv}^{\mathrm{g3}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{B}_3)$$

where $q_{\mathsf{chal}} \leq q$ denotes the number of CHAL oracle queries made by $\mathscr{A}$.

Proof. The proof is identical to that of Lemma 19 so we omit it. □

Lemma 25. There exists an adversary $\mathscr{B}_4$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g3}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq q^2 \cdot \left(\mathsf{Adv}^{\mathrm{g4}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + \mathsf{Adv}^{\mathrm{2pc-ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathscr{B}_4) + N \cdot q \cdot \mathsf{Adv}^{\mathrm{prg}}_{\mathsf{H}}(\mathscr{B}_4) + q \cdot \mathsf{Adv}^{\mathrm{2prf}}_{\mathsf{F}}(\mathscr{B}_4)\right)$$

where N is the concurrency bound (c.f. Section 6.5.2).

Proof. The proof diverges from the proof of Lemma 20 in order to handle the new update mechanism. As in Lemma 20, we consider the event E_{ck} where the $\mathsf{CHAL}(ID, \cdot, \cdot)$ call invokes Send with key mk in the key sequence starting from ck. Lemma 20 then constructs hybrids $H_{-1}, H_0, \ldots, H_m$ given E_{ck}.

Observe that, fixing ck, mk could have been derived from zero or more update operations initiated by $ID' \neq ID$. Now, by cleanness, after invoking the security of the two-party channels, mk is hidden from the adversary. If no update operations were made, then it must be that ck is not exposed. If there was one update operation, then by cleanness, either the update secret r or ck are hidden (or possibly both).

Let i be the i-th last update operation applied to form mk, where $i \in [0, q']$ for some $q' < q$. Let $E_{\mathsf{ck},i}$ be the event that the i-th last update operation is hidden from the adversary for $i \geq 1$, and $i = 0$ where ck itself is secure. Observe that $E_{\mathsf{ck}} = \cup_{i \in [0,q']} E_{i,q}$ for some $q' < q$.

Our proof strategy then is as follows. For each $E_{\mathsf{ck},i}$, we first hop by invoking the security of the two-party channels to replace all two-party ciphertexts that communicate 'safe' value r or ck (or a descendent in the key sequence) with encryptions of dummy messages (hopping between H_{-1} and H_0). The simulation proceeds analogously to that in Lemma 20. Then (hopping to H_m):

- For $i = 0$, we hop by iteratively replacing all relevant H_j calls using the PRG assumption; there are at most $N \cdot q$ such queries. We replace each F call (of which there are at most q) along the way with a uniform value by the dual PRF assumption, keying F with ck'.
- For $i \geq 1$, we first hop by replacing the first call to F made by the challenger with r with a uniform value using the dual PRF assumption and keying F with r. We then hop using the PRG assumption on H_j and keying PRF F thereafter with ck'; note there are at most $N \cdot i \leq N \cdot q$ such calls.

By a similar argument to Lemma 20, the hop with the two-party channels and the hops with the PRG and dual PRF are sound. The result follows by combining the sequence of hybrids. □

Lemma 26. There exists an adversary $\mathscr{B}_5$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathsf{g4}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathsf{g5}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) + q \cdot \mathsf{Adv}^{\mathsf{suf\text{-}cma}}_{\mathsf{MAC}}(\mathscr{B}_5).$$

Proof. Let E be the event that a successful forgery to $\mathsf{RECEIVE}$ is made in a clean execution of Game Γ_4. Note Game Γ_4 and Game Γ_5 are identical given $\neg E$. Let E_i be the event that $\mathscr{A}$'s first forgery with $\mathsf{RECEIVE}$ is with respect to the i-th key sequence starting from ck; note $E = \cup_{i \in [1,q']} E_i$ for some $q' \leq q$. Observe by the previous game hop that the corresponding MAC key $\tau\mathsf{k}$ is uniform. We construct a $\mathsf{SUF\text{-}CMA}_{\mathsf{MAC}}$ adversary $\mathscr{B}$ that simulates for Game Γ_4 adversary $\mathscr{A}$ given E_i holds. $\mathscr{B}$ locally simulates all calls except MAC query using $\tau\mathsf{k}$ on message m during the simulation is replaced with the output of query TAG. Finally, when $\mathscr{A}$ makes a successful forgery using $\tau\mathsf{k}$, $\mathscr{B}$ extracts the tag t from the message and returns it to its challenger. The simulation is perfect and so the result follows by a similar derivation as in Lemma 18. □

Lemma 27. There exists an adversary $\mathscr{B}$ with similar running time to $\mathscr{A}$ such that

$$\mathsf{Adv}^{\mathrm{g5}}_{\mathsf{GM},\mathsf{C}_{\mathsf{gm}}}(\mathscr{A}) \leq \mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathsf{SymEnc}}(\mathscr{B})$$

Proof. The proof is essentially identical to that of Lemma 21 so we omit it. □

7 Conclusion

In this thesis, we have considered four different aspects of messaging, the first two concerning two-party communication (Part I), namely post-quantum X3DH key exchange (Chapter 3) and active attack detection (Chapter 4), and the latter two group messaging (Part II), namely group administration (Chapter 5) and the Sender Keys protocol (Chapter 6), each of which we have reasoned about in the framework of provable security.

7.1 Primitive Summary

We introduced several primitives and abstractions over the course of this thesis to capture different aspects of messaging. Here, we summarise and compare them, and consider how they may be instantiated. As done for this thesis as a whole, we present our comparison in two parts, namely in terms of 1) two-party communication and 2) group communication.

7.1.1 Two-Party Communication

Authentication. The goal of Chapter 3 is to construct a post-quantum X3DH-like authenticated key exchange protocol, namely K-Waay. We capture this in our deniable authenticated key exchange (DAKE) primitive that we introduce in Section 3.3. The syntax is somewhat non-standard as it generalises previous work in the BatchReceive algorithm that allows a *receiver* to simultaneously finish several key exchange sessions with different *senders*. Apart from this, our security notion can be seen as a Bellare-Rogaway-style definition [BR94] that combines aspects of the models of Hashimoto et al. [HKKP22] and Brendel et al. [BFG+22b]. DAKE assumes a public key infrastructure (PKI) is made available to all parties for registering long-term keys (in particular, our model captures for malicious key registration).

(Authenticated) Ratcheted Communication. In Chapter 4, we define the *ratcheted communication* (RC) primitive to capture stateful message exchange between two parties where ciphertexts can be re-ordered or dropped (i.e., supports *immediate decryption* [ACD19]). To capture an additional out-of-band channel that parties use for detecting active attacks, we

then define the more general *authenticated ratcheted communication* (ARC) primitive. These primitives abstract away authentication in the Init algorithm that outputs the initial states of both parties; we assume this is implemented in practice through something like K-Waay.

Syntactically, RC and ARC both have stateful Send and Receive algorithms which take associated data as input. Both algorithms output an *ordinal*, such as an index, which captures information about the order of ciphertexts. ARC further provides two algorithms, AuthSend and AuthReceive, that model sending and receiving authentication tags over the out-of-band channel. We propose several instantiations of both RC and ARC throughout the chapter with different security/efficiency trade-offs. All of our constructions make black-box use of an RC – one construction (Figure 4.8) requires the underlying RC to be RID-secure (Definition 37), and the rest only require correctness properties (Definitions 35 and 36) on the underlying RC to hold.

Two-Party Channels. In Chapter 6 we introduce another abstraction for two-party communication, namely *two-party channels* (2PC). This primitive is tailored to its use in instantiating Sender Keys and also supports immediate decryption. By contrast to Init in RC and ARC, we allow a given party to initialise channels with several parties over time via InitCh after initialising their own states with Init. In addition, our Send and Receive algorithms for 2PC do not take associated data as input, and we specifically assume ordinals are pairs of integers, which we call *channel epochs* that behave like the epoch/index pairs of Alwen et al. [ACD19]. We do not provide an explicit instantiation for 2PC here but we note that InitCh can be instantiated as above via an AKE and PKI, and Send and Receive via, e.g., the Double Ratchet protocol [ACD19] as done in implementations of Sender Keys in practice.

7.1.2 Group Communication

Authentication. Throughout Chapter 5, like our DAKE K-Waay, our protocols directly assume the existence of a PKI on which parties can upload keys. Unlike for DAKE for which we captured static long-term keys, we assume that parties can update their keys over time. Note that our protocols do not require a two-party AKE protocol: in practice in MLS, parties upload so-called *key bundles* that parties use as bootstrapping key material when a party is added to a group. In Chapter 6, authentication is instead abstracted away in the underlying two-party channels. Thus, Sender Keys and Sender Keys+ as written in Chapter 6 do not directly invoke a PKI, although two-party channels in practice are of course instantiated with PKI and AKE.

Continuous Group Key Agreement. In Chapter 5, we study continuous group key agreement (CGKA) and a generalisation that we introduce, namely *administrated CGKA* (A-CGKA). In CGKA, a dynamic group agrees on a sequence of secret keys over time (that can be used to build group messaging proper [ACDT21a] but may be useful in other contexts). Our formulation of CGKA, following the current MLS standard, is in the propose-and-commit paradigm, where group changes and key updates are first proposed (Prop) before being combined and committed (Commit) by some group member, after which group members process the corre-

sponding commit message (Proc). We discuss and compare variants of CGKA in the literature in Section 5.2.1.

Extending plain CGKA, A-CGKA provides first-class support for the addition, removal and updating of the keying material of *group administrators*. In Section 5.3, we formalise and prove secure two A-CGKA constructions, namely *individual admin signatures*, or IAS, and *dynamic group signature*, or DGS.

Group Messenger. In Chapter 6, we model Sender Keys and our enhanced protocol Sender Keys+ as instances of a *group messenger*, a primitive which we introduce. Unlike CGKA, a group messenger captures the sending and receiving of application messages (via Send and Recv). Our syntax here does not capture input associated data, and only Recv outputs an ordinal, which we simply assume is an epoch/index pair. Epochs here increase whenever the group membership changes (`add`/`rem`) or a group member performs a key update operation (`upd`). Group operations are captured via dedicated algorithms Exec and Proc: for simplicity we do not work in the propose-and-commit paradigm but it is straightforward to capture this in a modified group messenger primitive.

7.2 Discussion and Future Work

7.2.1 Composability

Throughout this thesis, we have worked with game-based security definitions, which do not immediately or necessarily provide security guarantees when the corresponding primitives are used in a broader context and under concurrent composition. Note our DAKE indistingishability notion (Definition 26) considers a single real/random challenge query (TEST in our syntax); multi-challenge security follows with q multiplicative tightness loss for q challenge queries via a standard argument. Moreover, results of Brzuska et al. [BFWW11] indicate that Bellare-Rogaway-style key indistinguishability notions provide security guarantees for a broad class of protocols which assume that key exchange has taken place at initialisation time. As many two-party ratcheting and messaging primitives assume this already, including our RC primitive, they benefit from these composition results. Since these results only consider two-party key exchange, a natural direction is to devise similar ones for game-based CGKA. Previous work on both two-party [BFG+22a, CJSV22] and group messaging [AJM22, HKP+21, AHKM22, ACJM20] has alternatively considered simulation-based security notions, mostly in the *universal composability* framework [Can01] (the work of Jost et al.[JMM19b] in a variant of the *constructive cryptography* framework being an exception here).

7.2.2 Model Limitations

Whether the security notions are game-based or simulation-based, one must be careful interpreting the security claims of a given work. As we note below in more detail, there are

attack vectors, such as randomness manipulation [BRV20], that we do not always capture in our security notions in this thesis.

Our notion of state exposure is not very fine-grained: in Chapter 3, we allow individual sesssion states and long-term keys to be exposed separately, but otherwise we assume the entire secret state of a party is given to the adversary at once. Consequently the effects of partial leakage or state exposure are only captured through these means: bridging this gap further, for example considering leakage resilience in messaging, is of interest.

Our security notions, like all work we are aware of on messaging, further assume where relevant that code is faithfully and honestly executed at all times which may not always hold in practice.

7.2.3 Unification and Verification

Due to the complexity of messaging primitives, different works tend to introduce their own primitive syntax and security notions that can differ enough to render comparison across works complex and time-consuming. For pedagogical purposes and to enable easier formal comparison between different works, it is thus of interest to overcome this complexity. The work of Jost et al. [JMM19b] provided some recourse for two-party messaging: they show components of protocols in the literature [PR18, JS18, ACD19, DV19, JMM19a] can be expressed with some minor modifications in their composable framework. In addition, some authors have also compared related work in their particular security model, including the game-based models of Durak and Vaudenay [DV18] and Cremers and Zhao [CZ22].

In addition, the complexity of messaging makes it error-prone and difficult to verify. It is not always sufficient to reduce the complexity by considering a simplified security definition either, since attack vectors can and have previously been overlooked when taking this approach [AJM22]. Formal analysis and verification is therefore a natural direction to pursue, and has successfully been applied in messaging [KBB17, WPBB23]. However, there is still a gap in the literature for messaging protocols as much of the previous work considers symbolic/Dolev-Yao-style modelling rather than full computational security. Nonetheless, results like the analysis of Signal's PQXDH key exchange protocol [BJKS23b] in CryptoVerif [Bla07] are promising and indicate that formal computational analysis of different messaging protocols is in reach.

7.2.4 Deniable Post-Quantum X3DH (Chapter 3)

In Chapter 3, we proved the key indistinguishability of our protocol K-Waay in a model that considers state exposures like Hashimoto et al.'s [HKKP22] but is nonetheless weaker (our protocol is provably secure under a notion similar to that of Brendel et al. [BFG+22b]). This is mainly since K-Waay only uses ephemeral split-KEM keys. As noted by Brendel et al. [BFG+20], however, it seems much more difficult to construct split-KEM secure under

several encapsulation/decapsulation queries, which we leave as important future work.

An interesting line of research would be to try to build other unforgeable IND-1BatchCCA split-KEMs that are more efficient (mostly in key and ciphertext size). One obvious direction would be to work over structured lattices [LS15, LPR10, SSTX09]. Indeed, Ring/Module-LWE with hints (similar to our Extended-LWE problem) have already been analysed from a theoretical point of view [BJRW21, MKMS22]. We also believe that our techniques can also be applied in the ring setting. However, for security purposes one needs to take a ring dimension d to be at least linear in the security parameter λ which becomes problematic when proving deniability. Indeed, the leaked hint is informally the product of secret keys of both parties. Thus, in the ring setting the hint would be at least a single polynomial, which contains $d = O(\lambda)$ coefficients. We predict that this would result with much larger reduction loss than what we have now. However, the concrete analysis is left as future work.

On the more practical side, it would be informative to benchmark our protocol and others in a real-life scenario or something close to it, and to implement other ring signatures schemes to have a more complete comparison. In light of the recent deployment of the PQXDH protocol [KS23], it would be prudent also to benchmark this and compare it with our protocols, replacing our choice of Kyber-512 with Kyber-1024 to be consistent with PQXDH which uses the latter KEM (note that the non-standard primitive used for the deniable PQ protocols should nonetheless be more of a bottleneck).

One could also try to build one-time ring signatures that are both efficient and provably secure (possibly in the QROM). In turn, these could possibly be used to build efficient ephemeral split-KEMs. For instance, Scafuro and Zhang [SZ21] designed an efficient linkable one-time ring signatures from hash functions alone and proved the security of the scheme in the ROM. It would be of interest to understand whether split-KEMs can be built out of such a construction or a variant, and/or to prove security in the QROM. Different parameter sets for K-Waay to achieve higher levels of security could also be provided and benchmarked.

7.2.5 Active Attack Detection in Messaging (Chapter 4)

Firstly, it is of interest to evaluate our protocols/transformations and determine more practically the overhead that active attack detection incurs. We believe that our most practical scheme is the epoch-based s-RID-secure one (Section 4.6.3) since the overhead can reasonably expected to be a single hash and a relatively small amount of indices in practice. By applying a comparable optimisation for r-RID in future work and with some benchmarking and evaluation, it may be deemed viable in some deployments to achieve r-RID security. Note also here that our lower bound for r-RID security considers a worst-case execution where one party is continually sending without receiving any messages from their counterpart.

A natural direction is to explore active attack detection in *group messaging*. In a first step, one could generalise the RECOVER/RID definitions to the group setting and in particular capture

any additional properties that may only arise in the group setting. In a second step, as done in this thesis for the two-party case, it would be prudent to design practical schemes and explore different trade-offs between security and efficiency. One could leverage properties of a given group messaging scheme along the way – for example, it may be prudent to take advantage of the assumption of in-order delivery provided by CGKA control messages in MLS to avoid the inherent complexity of out-of-order delivery on the messaging layer. Rather than modelling pairwise two-party out-of-band channels, assuming the existence of a 'global' out-of-band channel may simplify protocol design and make more sense practically to capture something like a public ledger.

7.2.6 Group Administration (Chapter 5)

We consider possible extensions of A-CGKA as a primitive and corresponding construction ideas. We note that these extensions may provide stronger security guarantees, or additional functionality, at reduced cost if the number of admins is small.

Admins Beyond CGKA. CGKA is not a suitable formalism for some group messaging protocols used in practice like pairwise channels and Sender Keys (the latter by Signal and WhatsApp [BCG23b]). In these protocols, each user is associated with their own key or keying material rather than a common group secret. Nevertheless, an IAS-like protocol can be easily adapted to this setting. For Sender Keys, admins could replace their keying material at a low cost (a signature attesting to their new signing key) for PCS authentication guarantees. We leave it as useful future work to formalise group administration beyond CGKA.

Telegram, although not end-to-end encrypted, offers fine-grained administration features like message filtering and delays. Some of these could be conceivably implemented cryptographically, e.g., by entrusting admins to process messages or through NIZKs.

Private Admins. In some applications, it may be desirable to hide the set of admins from (non-admin) users within a group (or between themselves). DGS could achieve some notion of administrative privacy if the underlying admin CGKA provides privacy guarantees. IAS could be modified to achieve anonymity guarantees using ring signatures [RST01]. However, there is overhead with ring signatures over regular signatures, at a minimum to parse the anonymity set.

In MLS' TreeKEM protocol, proposals are constant-sized, but commits are variable, which leaks information about the contents of the commit even if it is encrypted. Thus, padding is required at a minimum for privacy. In the MLS standard, ciphertexts (`PrivateMessage`) leak the group ID, epoch and message content type (proposal or commit) in plaintext[1], which need to be hidden for additional privacy. The work of Hashimoto et al. [HKP22] tackles this issue by hiding this information and further protecting against some additional leakage. In particular, their CGKA compiler allows group members to anonymously authenticate themselves, taking

[1] https://www.rfc-editor.org/rfc/rfc9420.html#name-encoding-and-decoding-a-pri

advantage of the CGKA secret, like DGS, to this end. In practice, additional attack vectors like timing and traffic analysis preclude privacy also, which are considered by some messaging systems [TGL+17, CSM+20] that do not, however, provide FS and PCS; it remains open to, e.g., adapt CGKA to defend against these attack vectors. As discussed in the introduction, the Signal Private Group System [CPZ20] hides the group membership from non-members (although not all metadata); the mechanism could be extended for admins but adapting the technique to (A)-CGKA is also open. Recently, some steps towards preserving anonymity in messaging even under state exposure have been made [DHRR22, BRT23].

External Admins. Our A-CGKA constructions assume the admins comprise a subset of all group members, i.e., $G^* \subseteq G$. Some applications may be better suited for *external* administration. For example, an online platform may wish to control the set of conversation participants to ensure they are subscribers but nevertheless ensure they are provided confidentiality. External admins who then attempt to add users that group members do not trust can be detected on the protocol level, rather than the less well-defined application level as previously done.

Given that the underlying CGKA allows for external commits, it is straightforward to administrate IAS and DGS-based groups externally. Namely, the admin who is approving the change can inspect proposals and make commit messages for the corresponding parties. However, TreeKEM and its variants are not ideal for this since the committer is the party who derives new group secrets and can thus violate confidentiality. One way around this issue is to essentially write a wrapper around each CGKA algorithm which declares that some CGKA group members are not actually in the group. Here, the wrapper would also force admins to delete group secrets as soon as they derive them, and would not consider admins as part of the group; the solution is however clearly vulnerable to corrupted admins.

One conceptually simple solution is to allow commit messages from regular users which play the role of proposals which have to be "committed" (e.g. signed) by admins. Additional machinery like the use of NIZKs is however required in the malicious setting to enable admins to verify that such commits are well-formed.

Hierarchical Admins. In messaging apps like Telegram and WhatsApp, the group creator has stronger capabilities than other admins. For instance, the group creator can never be removed by another admin. Extending this concept, one can conceive a hierarchy of administrators of several levels, e.g., of the form $G^{**} \subseteq G^* \subseteq G$, where G^{**} are super-administrators. Extending IAS, one can imagine using signatures that attest to other signatures in a chain-of-trust fashion. DGS can be extended by considering many CGKAs where the $(i+1)$-th CGKA must sign commit messages for the i-th CGKA for each $i \geq 1$. Attribute-based admins would enable greater flexibility in the access structure.

Muting Admins. It is possible to provide some cryptographic guarantees to the process of muting conversation participants. Although we do not explicitly consider group messaging proper, we sketch how such a solution would look. One solution entails a DGS-like construction in which members must sign messages using a common signature key spk derived from

a secondary CGKA; honest group members would then process application messages if and only if they are signed using the common signature key (i.e., only from the set of unmuted users). Then, muted members will be able to filter messages from other muted users (since they could still be informed of the state of spk over time), but they will not able to sign their own messages. Mechanisms that enable a central server to filter messages while maintaining privacy [HS20] can be also be integrated into the encryption layer over A-CGKA. Nevertheless, we note that in messaging services where the identity of the group members is known, muted members can generally bypass a ban by sending individual messages to all group members using two-party messages. At an application level, muting group members is a functionality supported by both Signal and Telegram.

Threshold Admins. One issue with our A-CGKA constructions is that security breaks down if a single admin is compromised. To improve the robustness of the protocol, a protocol can enforce that some $k > 1$ admins must attest to a particular commit before it may be processed, which can be achieved using threshold cryptography [Sho00].

Decentralised Admins. To allow for network decentralisation, it is straightforward in theory for a given messaging group G to simply execute a state machine replication protocol [CL+99] to order commit messages and require that users reliably broadcast [BT85] all proposal messages. Given that group members who are expected to execute the protocol on, e.g., mobile devices, may not be available often, thus leading to liveness (and possibly unintended safety) violations in protocol execution, a natural solution is to entrust administrators to provide messages to users. These admins could indeed execute consensus.

7.2.7 Sender Keys (Chapter 6)

Our security model, both for two-party channels and our Group Messenger primitive that captures Sender Keys and Sender Keys+, could be extended to encompass attack vectors like randomness manipulation, successful message injections and insider threats not explicitly captured in our modelling.

Investigating the practical behaviour of Sender Keys would provide valuable insights for improved modelling and the identification of potential vulnerabilities. In particular, Sender Keys is commonly supplemented by additional mechanisms not considered in our study, such as support for multiple devices and encrypted cloud backups (the latter considered in isolation recently by Davies et al. [DFG+23]) that increase the attack surface. Conversely, features of Signal such as sealed sender (by increasing privacy) and delivery receipts (by refreshing keys) can also improve security. Benchmarking both the baseline and extended Sender Keys protocols would also contribute to assessing their practicality.

Additionally, it is important to address the challenges that arise when total order is violated, and to design a protocol that avoids the drawbacks associated with decentralised continuous group key agreement (DCGKA) such as the need for multi-round communication [WKHB21].

Towards a more concurrency-friendly Sender Keys protocol, an important direction is the design of a mechanism for resolving ties in control messages that are sent concurrently. The way that this is achieved for group membership in Signal's private group system [CPZ20], as discussed in Section 5.4.2 in the context of group administration, is by the central server maintaining the canonical set of group members (bypassing the need for group members to agree between themselves). It could be of interest therefore to model such a system composed with Sender Keys, since the existing modelling of the private group system does not itself capture messaging proper as written (even given it is modelled in a variant of the universal composability framework [Can01]).

A Appendices

A.1 QROM Preliminaries

In this section, we assume the random oracles output values in $\{0,1\}^n$ for some integer n. We first recall the notion of extractable random oracle simulator introduced by Don et al. [DFMS22] and the corresponding properties as presented by Huguenin-Dumittan and Vaudenay [HV22], and a useful lemma. We refer the reader to the original paper for more details.

Definition 50 (Extractable RO [DFMS22])**.** An extractable RO-simulator is a tuple $(\mathsf{S},\mathsf{Ext})$, where S is a compressed RO efficiently simulatable and Ext is the extractor, such that the following properties hold.

1. If the extractor is never called, the simulator is indistinguishable from a (standard) RO.
2. Any two subsequent independent queries to S commute.
3. Any two subsequent independent queries to Ext commute.
4. Any two subsequent independent queries to Ext and S $8\sqrt{2/2^n}$-almost commute.
5. Querying classically the simulator S on the same value multiples times in a row has the same effect on the state of S as making *one* of these queries.
6. Let $x^{\mathsf{e}} \leftarrow \mathsf{Ext}(t)$ for some t, and $t' \leftarrow \mathsf{S}(x^{\mathsf{e}})$ be two subsequent classical queries. Then,
$$\Pr[t \neq t' \wedge x^{\mathsf{e}} \neq \perp] \leq 2/2^n \,.$$
7. Let $t \leftarrow \mathsf{S}(x)$ for some x and $x^{\mathsf{e}} \leftarrow \mathsf{Ext}(t)$ be two subsequent classical queries. Then,
$$\Pr[\widehat{x} = \perp] \leq 2/2^n \,.$$

$\mathsf{COLL}(\mathscr{A})$

1: $(x_1,t_1),\ldots,(x_m,t_m) \leftarrow \mathscr{A}^{\mathsf{S}}$
2: **for** $i \in \{1,\ldots,m\}$: $t_i' \leftarrow \mathsf{S}(x_i)$
3: **for** $i \in \{1,\ldots,m\}$: $x_i^{\mathsf{e}} \leftarrow \mathsf{Ext}(t_i)$
4: **if** $\exists i : x_i^{\mathsf{e}} \neq x_i$ **and** $t_i = t_i'$:
5: **return** 1
6: **return** 0

Figure A.1: Collision game for Property 8 Definition 50.

8. Let COLL be the game defined in Figure A.1. Then, for any $\mathscr{A}$ we have

$$\Pr[\mathsf{COLL}(\mathscr{A}) \Rightarrow 1] \leq \frac{40e^2(q+m+1)^3+2}{2^n},$$

where q is the number of queries $\mathscr{A}$ makes to S and m is the number of tuples output by $\mathscr{A}$ in the game.

Lemma 28 (Early Extraction)**.** Let Γ be a game where an adversary runs on some input, queries a quantum RO H, and outputs two values t and o. Then, the game applies some deterministic function on o to obtain a value x and queries $h \leftarrow H(x)$. The game returns 1 if $h = t$. Now let Γ' be the same as Γ, except the extractor is called on t right after it is returned by $\mathscr{A}$, and the game returns 1 if $h = t$ *and* $x = x^\star$, where $x^\star$ is the value extracted. Then,

$$\Pr[\Gamma \Rightarrow 1] - \Pr[\Gamma' \Rightarrow 1] \leq \frac{2}{2^n} + 8\sqrt{2/2^n} + \frac{40e^2(q_H+2)^3+2}{2^n}.$$

Proof. This follows from Corollary 4.7 in Don et al. and the fact that if $h = t$, where $h = H(x)$, then $\Pr[x^\star = \perp] \leq \frac{2}{2^n}$. □

We also recall the algorithm one way to hiding lemma [HHK17].

Lemma 29 (AOW2H [HHK17])**.** Let $\mathscr{A}$ be a quantum adversary making at most q_H queries to the QRO $H : \{0,1\}^\ell \mapsto \{0,1\}^n$ and outputting 0 or 1. Then, for any algorithm F that does not use H

$$\begin{aligned}&\Big|\Pr[\mathscr{A}^H(x) \Rightarrow 1 : \sigma^* \xleftarrow{\$} \{0,1\}^\ell;\ x \leftarrow \mathsf{F}(\sigma^*, H(\sigma^*))] \\ &- \Pr[\mathscr{A}^H(x) \Rightarrow 1 : (\sigma^*, K) \xleftarrow{\$} \{0,1\}^{\ell+n};\ x \leftarrow \mathsf{F}(\sigma^*, K)]\Big| \\ &\leq 2q_H\sqrt{\Pr\left[\sigma^* \leftarrow \mathsf{E}^{\mathscr{A},H}(x) : \begin{array}{c}(\sigma^*,K) \xleftarrow{\$} \{0,1\}^{\ell+n};\\ x \leftarrow \mathsf{F}(\sigma^*,K)\end{array}\right]}.\end{aligned}$$

where E is an algorithm that runs $\mathscr{A}$, measures the input register of a random query made to H, and outputs the result.

$\mathsf{S}^{H,\mathscr{A}}_{i^*}(\Theta)$

1: $(j,b) \xleftarrow{\$} \big(\{0,\dots q_H-1\} \setminus \{i^*\} \times \{0,1\}\big) \cup \{(q_H,0)\}$
2: $q \leftarrow 1$
3: $(x,z) \xleftarrow{\$} \mathscr{A}^{H'}$ and
   $x' \leftarrow$ measure $\mathscr{A}$'s $j+1$-th query input register
4: **return** (x,z)

$H'(x)$

1: **if** $q = i^*$:
2: **return** Θ
3: **if** $q < j+b+1$:
4: **return** $H(x)$
5: **else**
6: **if** $x = x'$: **return** Θ
7: **else** : **return** $H(x)$

Figure A.2: Algorithm S for Lemma 30.

Finally, we recall the measure-and-reprogram lemma of Jiang et al. [JMZ23].

Lemma 30 (Lemma 3.1 [JMZ23])**.** Let $H : \{0,1\}^m \mapsto \{0,1\}^n$ be a quantum random oracle and $\mathscr{A}^H$ be a quantum algorithm that makes q quantum queries to H and outputs (x,z), where x and z are classical. Furthermore, we assume the i^*-th query of $\mathscr{A}$ to H is classical and equal to x, for some $i^* \in [q_H]$. In addition, let $V(x,y,z)$ be some predicate s.t. $V(x,y,z) = 1$ implies that y was output on $\mathscr{A}$'s i^*-th query to H. Then, there exists an algorithm S_{i^*} (see Figure A.2), that takes some $\Theta \in \{0,1\}^n$ as input and is such that

$$\Pr\left[V(x,H(x),z) = 1 : (x,z) \leftarrow \mathscr{A}^H\right] \leq 2(2q_H+1)^2 \Pr\left[V(x,\Theta,z) = 1 : (x,z) \leftarrow \mathsf{S}^{\mathscr{A}}_{i^*}(\Theta)\right] + \frac{8q_H^2}{2^n}$$

where the probabilities are taken over the randomness of the algorithms, the random oracle H and the sampling of Θ at random.

Informally, the previous lemma states that if some adversary $\mathscr{A}^H$ can satisfy a predicate with probability p, one can build another algorithm $\mathsf{S}^{\mathscr{A}}$ that does not query H on the i^*-th query (but uses its input instead) but that can satisfy with probability $\approx \frac{p}{q_H^2}$.

A.2 Proof of Theorem 5 (Chapter 3)

A.2.1 Proof in the QROM

We proceed with a sequence of games that is detailed in Figure A.3. The proof uses the extractable RO-simulator of Don et al. [DFMS22] (see Definition 50).

Game Γ_0: This is the UNF-1KCA game with $\mathsf{sKEM} := \mathsf{T}_{\mathsf{CH}}(\mathsf{sKEM}_0)$ written explicitly. In addition, the RO used to compute the tag corresponding to ct (i.e. $t = H_1(\mathsf{pk},\mathsf{pk}_\mathsf{B},\mathsf{ct},K_\mathsf{B})$) is different from the one used to compute the tag for ct' (i.e. $t_c = H_2(\mathsf{pk}_\mathsf{A},\mathsf{pk}_\mathsf{B},\mathsf{ct}',\mathscr{K}_\mathsf{A})$). Note that since $(\mathsf{pk},\mathsf{ct}) \neq (\mathsf{pk}_\mathsf{A},\mathsf{ct}')$ for the adversary to win, both oracles can be separated in this way.

Game Γ_1: The game is the same as the previous one, except we use the simulated RO for H_2, and we use the extractor on t' (the tag output by the adversary) at the end. Note that this does not change anything to the probability of success of the game.

Game Γ_2: Now the game outputs 0 if the values extracted are different than $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}', K_\mathsf{A})$. For the game to return 1, t_c must be equal to t', so let's assume it is the case. Hence, Γ^2 and Γ^1 differ only if $\mathsf{S.Ext}(t_c) \neq (\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}', K_\mathsf{A})$ and $H_2(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}', K_\mathsf{A}) = \mathsf{t}_c$. By Lemma 28, this happens with probability at most $\epsilon := \frac{2}{2^n} + 8\sqrt{2/2^n} + \frac{40e^2(q_{H'}+2)^3+2}{2^n}$. Hence, we have

$$\Pr[\Gamma_1] - \Pr[\Gamma_2] \leq \epsilon \,.$$

Game Υ_1: We see that if an adversary $\mathscr{A}$ wins Γ_2, one can build an adversary $\mathscr{B}$ that wins the game Υ_1 defined in Figure A.3. The reduction works simply by $\mathscr{B}$ running $\mathscr{A}$, simulating H_2 with the simulated RO, and running the extractor on t' at the end. Therefore, we have

$$\Pr[\Gamma_2] \leq \Pr[\Upsilon_1] \,.$$

In addition, note that one can consider oracles H and H_1 as *one* oracle $H^* := H_1 \otimes H$ with images in $\{0,1\}^{2n}$ that can be accessed $q_H + q_{H'}$ times by the adversary.

Game Υ_2: We change the game such that (t, K) are picked at random and the oracle used is now $\hat{H}$ instead of $H^* := H_1 \otimes H$. Now, let's consider a game $\mathscr{C}$ that runs Γ^1 and outputs $(x = (\mathsf{pk}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}, K_\mathsf{B}), z = ((t, K), K_\mathsf{A}, K'_\mathsf{A})$. In addition, let $V(x, y, z) := 1_{z_1 = y} \wedge \mathbb{1}[\![z_2 = z_3]\!]$. Clearly, we have that

$$\Pr[\Upsilon_1] \leq \Pr[V(x, H^*(x), z) : (x, z) \xleftarrow{\$} \mathscr{C}^{H^*}]$$

as V is satisfied iff $K_\mathsf{A} = K'_\mathsf{A}$. Also, note that the condition $z_1 = y$ in the predicate is always satisfied by the definition of z_1 itself. Therefore, one can apply Lemma 30 with i^* equal to the query to H^* made by the game (i.e. $(t, K) \leftarrow H^*(\mathsf{pk}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}, K_\mathsf{B})$) and we get

$$\Pr[\Upsilon_1] \leq 2(2(q_H + q_{H'}) + 1)^2 \Pr\left[V(x, (t, K), z) = 1 : (x, z) \leftarrow \mathsf{S}_{i^*}^{\mathscr{A}}((t, K)\right] + \frac{8(q_H + q_{H'})^2}{2^{2n}}$$

where (t, K) is sampled at random and S_{i^*} is the algorithm shown in Figure A.2. By inspection, one can see that if the output of S_{i^*} satisfies the predicate V then Υ_2 would output 1. Therefore, we have

$$\Pr[\Upsilon_1] \leq 2(2(q_H + q_{H'}) + 1)^2 \Pr\left[V(x, (t, K), z) = 1 : (x, z) \leftarrow \mathsf{S}_{i^*}^{\mathscr{A}}((t, K)\right] + \frac{8(q_H + q_{H'})^2}{2^{2n}} \leq \Pr[\Upsilon_2].$$

Finally, one can see that if $\mathscr{A}$ wins Υ_2, one can build an adversary $\mathscr{B}$ s.t. $\mathscr{B}$ wins the decaps-CPA game against sKEM_0. That is, the first phase of $\mathscr{B}$ runs the first phase of $\mathscr{A}$ and outputs the same public key pk. Then, in the second phase, $\mathscr{B}$ runs $\mathscr{A}^{\hat{H}}$ with its own input $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct})$

and random tag and key (t, K). In addition, note that $\mathscr{B}$ can perfectly simulate $\hat{H}$. Finally, $\mathscr{B}$ outputs the same as the adversary $\mathscr{A}$. If $K_{\mathsf{A}} = K'_{\mathsf{A}}$ then $\mathscr{B}$ wins the decaps-OW-CPA game. Hence, we have that

$$\Pr[\Upsilon_2] \leq \Pr[\text{decaps-CPA}_{\mathsf{sKEM}_0}(\mathscr{B}) \Rightarrow 1] .$$

Collecting the probabilities concludes the proof. □

$\Gamma_0(\mathscr{A})$

$\mathsf{pk}_{\mathsf{A}}, \mathsf{sk}_{\mathsf{A}} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
$\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{B}} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
$\mathsf{pk}, \mathsf{st} \xleftarrow{\$} \mathscr{A}^{H,H_1,H_2}(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}})$
$(K_{\mathsf{B}}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}, \mathsf{sk}_{\mathsf{B}})$
$t \leftarrow H_1(\mathsf{pk}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}, K_{\mathsf{B}})$
$K \leftarrow H(\mathsf{pk}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}, K_{\mathsf{B}})$
$(\mathsf{ct}', t') \xleftarrow{\$} \mathscr{A}^{H,H_1,H_2}(\mathsf{pk}_{\mathsf{A}},$
$\quad \mathsf{pk}_{\mathsf{B}}, (\mathsf{ct}, t), K, \mathsf{st})$
if $(\mathsf{pk}_{\mathsf{A}}, \mathsf{ct}') = (\mathsf{pk}, \mathsf{ct})$: **return** 0
$K_{\mathsf{A}} \leftarrow \mathsf{Decaps}(\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{A}}, \mathsf{ct}')$
$t_c \leftarrow H_2(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}', K_{\mathsf{A}})$
if $t_c \neq t'$: **return** 0
return 1

$\Gamma_1(\mathscr{A})$

$\mathsf{pk}_{\mathsf{A}}, \mathsf{sk}_{\mathsf{A}} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
$\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{B}} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
$\mathsf{pk}, \mathsf{st} \xleftarrow{\$} \mathscr{A}^{H,H_1,H_2}(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}})$
$(K_{\mathsf{B}}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}, \mathsf{sk}_{\mathsf{B}})$
$t \leftarrow H_1(\mathsf{pk}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}, K_{\mathsf{B}})$
$K \leftarrow H(\mathsf{pk}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}, K_{\mathsf{B}})$
$(\mathsf{ct}', t') \xleftarrow{\$} \mathscr{A}^{H,H_1,H_2}(\mathsf{pk}_{\mathsf{A}},$
$\quad \mathsf{pk}_{\mathsf{B}}, (\mathsf{ct}, t), K, \mathsf{st})$
if $(\mathsf{pk}_{\mathsf{A}}, \mathsf{ct}') = (\mathsf{pk}, \mathsf{ct})$: **return** 0
$K_{\mathsf{A}} \leftarrow \mathsf{Decaps}(\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{A}}, \mathsf{ct}')$
$t_c \leftarrow H_2(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}', K_{\mathsf{A}})$
if $t_c \neq t'$: **return** 0
$(\mathsf{pk}_1^\star, \mathsf{pk}_2^\star, \mathsf{ct}^\star, K_{\mathsf{A}}^\star) \leftarrow \mathsf{S.Ext}(t')$
return 1

$\Gamma_2(\mathscr{A})$

$\mathsf{pk}_{\mathsf{A}}, \mathsf{sk}_{\mathsf{A}} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
$\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{B}} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
$\mathsf{pk}, \mathsf{st} \xleftarrow{\$} \mathscr{A}^{H,H_1,H_2}(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}})$
$(K_{\mathsf{B}}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}, \mathsf{sk}_{\mathsf{B}})$
$t \leftarrow H_1(\mathsf{pk}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}, K_{\mathsf{B}})$
$K \leftarrow H(\mathsf{pk}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}, K_{\mathsf{B}})$
$(\mathsf{ct}', t') \xleftarrow{\$} \mathscr{A}^{H,H_1,H_2}(\mathsf{pk}_{\mathsf{A}},$
$\quad \mathsf{pk}_{\mathsf{B}}, (\mathsf{ct}, t), K, \mathsf{st})$
$(\mathsf{pk}_0^\star, \mathsf{pk}_1^\star, \mathsf{ct}^\star, K_{\mathsf{A}}^\star) \leftarrow \mathsf{S.Ext}(t')$
if $(\mathsf{pk}_{\mathsf{A}}, \mathsf{ct}') = (\mathsf{pk}, \mathsf{ct})$: **return** 0
$K_{\mathsf{A}} \leftarrow \mathsf{Decaps}(\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{A}}, \mathsf{ct}')$
$t_c \leftarrow H_2(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}', K_{\mathsf{A}})$
if $t_c \neq t'$: **return** 0
if $(\mathsf{pk}_1^\star, \mathsf{pk}_2^\star, \mathsf{ct}^\star, K_{\mathsf{A}}^\star)$
$\quad \neq (\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}', K_{\mathsf{A}})$:
$\quad$ **return** 0
return 1

$\Upsilon_1(\mathscr{A})$

$\mathsf{pk}_{\mathsf{A}}, \mathsf{sk}_{\mathsf{A}} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
$\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{B}} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
$\mathsf{pk}, \mathsf{st} \xleftarrow{\$} \mathscr{A}^{H,H_1}(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}})$
$(K_{\mathsf{B}}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}, \mathsf{sk}_{\mathsf{B}})$
$(t, K) \leftarrow H^*(\mathsf{pk}, \mathsf{pk}_{\mathsf{B}}, \mathsf{ct}, K_{\mathsf{B}})$
$in \leftarrow (\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, (\mathsf{ct}, t), K, \mathsf{st})$
$K'_{\mathsf{A}}, \mathsf{ct}' \xleftarrow{\$} \mathscr{A}^{H,H_1}(in)$
$K_{\mathsf{A}} \leftarrow \mathsf{Decaps}(\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{A}}, \mathsf{ct}')$
if $(\mathsf{pk}_{\mathsf{A}}, \mathsf{ct}') = (\mathsf{pk}, \mathsf{ct})$ **or** $K_{\mathsf{A}} \neq K'_{\mathsf{A}}$:
$\quad$ **return** 0
return 1

$\Upsilon_2(\mathscr{A})$

$(j, b) \xleftarrow{\$} \big(\{0, \ldots q_H - 1\} \times \{0, 1\}\big) \cup \{(q_H, 0)\}$
$x' \leftarrow$ measure $\mathscr{A}$'s $j+1$-th query input register
$q \leftarrow 0$
$\mathsf{pk}_{\mathsf{A}}, \mathsf{sk}_{\mathsf{A}} \xleftarrow{\$} \mathsf{KeyGenA}(1^\lambda)$
$\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{B}} \xleftarrow{\$} \mathsf{KeyGenB}(1^\lambda)$
$(K_{\mathsf{B}}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}_{\mathsf{A}}, \mathsf{sk}_{\mathsf{B}})$
$\mathsf{pk}, \mathsf{st} \xleftarrow{\$} \mathscr{A}^{\hat{H}}(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}})$
$(K_{\mathsf{B}}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encaps}(\mathsf{pk}, \mathsf{sk}_{\mathsf{B}})$
$(t, K) \xleftarrow{\$} \{0, 1\}^{2n}$
$K'_{\mathsf{A}}, \mathsf{ct}' \xleftarrow{\$} \mathscr{A}^{\hat{H}}(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, (\mathsf{ct}, t), K, \mathsf{st})$
$K_{\mathsf{A}} \leftarrow \mathsf{Decaps}(\mathsf{pk}_{\mathsf{B}}, \mathsf{sk}_{\mathsf{A}}, \mathsf{ct}')$
if $K_{\mathsf{A}} \neq K'_{\mathsf{A}}$:
$\quad$ **return** 0
return 1

$\hat{H}(x)$

$q \leftarrow q + 1$
if $q < j + b + 1$:
$\quad$ **return** $H^*(x)$
else
$\quad$ **if** $x = x'$:
$\quad\quad$ **return** (t, K)
$\quad$ **else** :
$\quad\quad$ **return** $H^*(x)$

Figure A.3: Sequence of games for the proof of Theorem 5. H^* is defined as $H_1 \otimes H_1$.

A.2.2 Proof in the ROM

The proof follows a similar approach as the one in the QROM.

Game Γ_0: This is the same as the UNF-1KCA game with $\mathsf{sKEM} = \mathsf{T}_{\mathsf{CH}}(\mathsf{sKEM}_0)$, except we assume there is no collision on H'. Thus, Γ^0 is the same as UNF-1KCA except with probability at most $\frac{q_{H'}^2}{2^n}$.

Game Γ_1: In this game, we return 0 if $\mathscr{A}$ did not query $H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}', K_\mathsf{A})$. As we can assume $(\mathsf{pk}, \mathsf{ct}) \neq (\mathsf{pk}_\mathsf{A}, \mathsf{ct}')$, this changes the probability of $\mathscr{A}$ winning only if $\mathscr{A}$ outputs $t' = H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}', K_\mathsf{A})$ without having made the oracle query. Since the query was not made, one can actually lazy sample the value of $H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}', K_\mathsf{A})$ after $\mathscr{A}$ returns t', and the probability both values are equal is $\frac{1}{2^n}$. Hence,

$$\Pr[\Gamma_0] - \Pr[\Gamma_1] \leq \frac{1}{2^n} .$$

Game Υ_1: If Γ_1 outputs 1, it means $\mathscr{A}$ outputs (ct, t') s.t. $((\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}', K_\mathsf{A}), t')$ is in the list of queries made by the $\mathscr{A}$. Hence, if that happens, one can find ct' and K_A s.t. $\mathsf{Decaps}(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{A}, \mathsf{ct}') = K_\mathsf{A}$ by running $(\mathsf{ct}', t') \xleftarrow{\$} \mathscr{A}$ and looking for t' in the list of queries (note that we assume there is no collision). Therefore, it means one can build an adversary that wins the game Υ_1 in Fig. A.3, and we have

$$\Pr[\Gamma_1] \leq \Pr[\Upsilon_1] .$$

Game Υ_2: We modify the game s.t. the tag t and the key given to the adversary are picked uniformly at random as shown in Figure A.3. Both games are indistinguishable unless $\mathscr{A}$ queries $(\mathsf{pk}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}, K_\mathsf{B})$ to H or H'. Then, an adversary $\mathscr{B}$ playing Υ_2 can perfectly simulate $\mathscr{A}$'s view in Υ_1 if it guesses correctly which query it is going to be and if such a query is going to happen. Overall, $\mathscr{B}$ can make a correct guess with probability $\frac{1}{q_{H'}+q_H+1}$. If that happens though, one can build an OW-CPA adversary $\mathscr{B}$ against sKEM_0 that runs $\mathscr{A}$ and picks a random query made by $\mathscr{A}$ to H or H'. Hence, we have

$$\Pr[\Upsilon_1] \leq (q_{H'} + q_H + 1) \Pr[\Upsilon_2] .$$

Finally, one can see that Υ_2 is the same as the decaps-CPA for sKEM_0 if we omit the random values K and t and the more restrictive winning condition $(\mathsf{pk}_\mathsf{A}, \mathsf{ct}') \neq (\mathsf{pk}, \mathsf{ct})$. Hence, one can build an adversary $\mathscr{C}$ such that

$$\Pr[\Upsilon_2] \leq \Pr[\mathsf{decaps\text{-}CPA}_{\mathsf{sKEM}_0}(\mathscr{C}) \Rightarrow 1] .$$

$\mathcal{O}(\mathcal{L}_{H'}, \{(\mathsf{pk}_i, (\mathsf{ct}_i, t_i))\}_{i=1}^{d})$

1 : **for** $i \in [d]$:
2 : $\quad K'_i \leftarrow \mathsf{Decaps}(\mathsf{pk}_i, \mathsf{sk}_\mathsf{A}, \mathsf{ct}_i)$
3 : $\quad$ **if** $((\mathsf{pk}_i, \mathsf{ct}_i, K'_i), t_i) \notin \mathcal{L}_{H'}$: **return** 0
4 : **return** 1

Figure A.4: Oracle $\mathcal{O}$ used in the proof of Theorem 6.

□

A.3 Proof of Theorem 6 (Chapter 3)

A.3.1 Proof in the ROM

Proof. The idea of the proof is very similar to the IND-qCCA proof of the T_{CH} transform by Huguenin-Dumittan et al. [HV22] and is the following. Either all tags in the decapsulation query are valid and thus they are the form $H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_i, \mathsf{ct}_i, K'_i)$, or the oracle returns $\perp$. Then, if they are valid, with high probability the adversary queried $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_i, \mathsf{ct}_i, K'_i)$ to H' and thus K'_i can be recovered from the list of queries to the RO, i.e. the decapsulation oracle can be simulated without the knowledge of sk_A. In other words, the only information leaked by a query to the decapsulation oracle is whether *all* tags are valid or not, i.e. 1 bit of information, which is not sufficient to break the OW-CPA game. We prove this formally with a sequence of hybrid games.

Game Γ_0: This is the IND-1BatchCCA game with $\mathsf{sKEM} = \mathsf{T}_{\mathsf{CH}}(\mathsf{sKEM}_0)$.

Game Γ_1: We modify the previous game s.t. we abort if the adversary finds any collision when querying H'. We have that

$$\Pr[\Gamma_0] - \Pr[\Gamma_1] \leq \frac{q_{H'}^2}{2^n}$$

where q'_H is the number of queries the adversary makes to H'.

Game Γ_2: We modify the game s.t. it aborts if $\mathsf{BatchDEC}(\{(\mathsf{pk}_i, (\mathsf{ct}_i, t_i))\}_{i=1}^{d})$ does not return $\perp$ but one of the tags t_i was not obtained through an adversary's query to H'. The probability that some tag t_i is valid but $H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_i, \mathsf{ct}_i, K'_i)$ (with $(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_i, \mathsf{ct}_i, t_i) \neq (\mathsf{pk}_\mathsf{A}, \mathsf{pk}_B, \mathsf{ct}^*, t_i^*)$) was

not queried by the adversary is $\frac{1}{2^n}$. Hence, overall we have

$$\Pr[\Gamma_1] - \Pr[\Gamma_2] \leq \frac{d}{2^n}$$

Game Γ_3: We now change the game as follows. We record all queries to H' of the form $(\mathsf{pk}_\mathsf{A}, \cdot, \cdot, \cdot)$ made by the adversary in a list $\mathscr{L}_{H'} = \{((\mathsf{pk}_j, \mathsf{ct}_j, K_j), h_j)\}_{j=1}^{q'_H}$ s.t. $H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_j, \mathsf{ct}_j, K_j) = h_j$ for all $j \in [q_{H'}]$. Then, the BatchDEC oracle is modified as follows. If some tag t_i is s.t. for all $K \in \mathscr{K}$ $((\mathsf{pk}_i, \mathsf{ct}_i, K), t_i) \notin \mathscr{L}_{H'}$ then $\perp$ is returned. Then, $\mathscr{O}(\mathscr{L}_{H'}, \{(\mathsf{pk}_i, (\mathsf{ct}_i, t_i))\}_{i=1}^{d}) \to r$ is queried, where $\mathscr{O}$ is defined in Figure A.4. If $r = 0$ BatchDEC outputs $\perp$, otherwise it outputs $H(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_i, \mathsf{ct}_i, K_i)$ for all $i \in [d]$, where K_i is s.t. $((\mathsf{pk}_i, \mathsf{ct}_i, K_i), t_i) \in \mathscr{L}_{H'}$. Note that all these modifications are only syntactical as $\mathscr{O}$ outputs 1 iff for all $i \in [d]$, K_i is (the unique) value in $\mathscr{L}_{H'}$ s.t. $H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_i, \mathsf{ct}_i, K_i := \mathsf{Decaps}(\mathsf{pk}_i, \mathsf{sk}_\mathsf{A})) = t_i$. Hence, we have

$$\Pr[\Gamma_2] = \Pr[\Gamma_3] .$$

Game Γ_4: We replace the challenge tag t^* and the real key K_0 by random values. This change can only be noticed if the adversary or the BatchDEC oracle queries $H(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}^*, K^*)$ or $H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{ct}^*, K^*)$ at some point in the game. Let QUERY be this event. We show that if QUERY occurs, then one can break the OW-CPA security of sKEM_0 with high probability. The reduction works as follows. The OW-CPA adversary $\mathscr{B}$ receives a challenge ciphertext ct^* and public keys $\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}$ from its own challenger. Next, it samples random values K, t^* and passes all these to the IND-1BatchCCA adversary $\mathscr{A}$. Then, $\mathscr{B}$ can simulate everything in BatchDEC (except the oracle call to $\mathscr{O}$) by recording $\mathscr{A}$'s queries to H'. In order to simulate $\mathscr{O}$, $\mathscr{B}$ samples a bit r at random instead, which succeeds with probability $\frac{1}{2}$. Finally, it samples at random a query made by $\mathscr{A}$ to H or H' or a query made to H by itself, and it outputs the key K that was part of this query. Overall, the simulation is correct with probability $\frac{1}{2}$ and if QUERY occurs $\mathscr{B}$ recovers K^* with probability $\frac{1}{q_H + q_{H'} + d}$. Hence,

$$\Pr[\Gamma_3] - \Pr[\Gamma_4] \leq \Pr[\mathsf{QUERY}] \leq 2(q_H + q_{H'} + d)\mathsf{Adv}^{\text{ow-cpa}}_{\mathsf{sKEM}}(\mathscr{A}) .$$

Finally, we see that the adversary's view is independent of b in Γ_4, therefore $\Pr[\Gamma_4] = \frac{1}{2}$. This concludes the proof. □

A.3.2 Proof in the QROM

Proof. As in the proof of Theorem 5, we use the extractable RO-simulator by Don et al. [DFMS22] and we proceed with a sequence of hybrid games. We note the proof is nearly identical to the QROM IND-qCCA proof of T_{CH} [HV22] and we refer the reader to the latter for

a detailed explanation of the game transitions.

Game Γ_0: This is the IND-1BatchCCA game with $\mathsf{sKEM} = \mathsf{T}_{\mathsf{CH}}(\mathsf{sKEM}_0)$. We also assume that the adversary only makes queries of the form $(\mathsf{pk}_\mathsf{A}, \cdot, \cdot, \cdot)$ to the oracles. This has no consequence on the winning probability of the adversary as other type of queries are independent of the key.

Game Γ_1: We modify the BatchDEC oracle s.t. it returns $\perp$ whenever the list of $(\mathsf{pk}_i, \mathsf{ct}_i, t_i)$ in the query contains $(\mathsf{pk}_i, \mathsf{ct}_i) = (\mathsf{pk}_\mathsf{B}, \mathsf{ct}^*)$ (and thus $t_i \neq t^*$). This change has no impact except if $\mathsf{Decaps}(\mathsf{pk}_\mathsf{B}, \mathsf{sk}_\mathsf{A}, \mathsf{ct}^*) \neq K_0$, where K_0 is the challenge real key. In turn, this would imply that ct^* is an incorrect ciphertext. Hence,

$$\Pr[\Gamma_0] - \Pr[\Gamma_1] \leq \delta \,.$$

Game Γ_2: Now, we split the random oracle H' into two oracles H'_0 and H'_1 s.t.

$$H'(\mathsf{pk}_\mathsf{A}, \mathsf{pk}, \mathsf{ct}, K) := \begin{cases} H'_0(K), & \text{if } (\mathsf{pk}, \mathsf{ct}) = (\mathsf{pk}_\mathsf{B}, \mathsf{ct}^*) \\ H'_1(\mathsf{pk}, \mathsf{ct}, K), & \text{otherwise} \end{cases}$$

and we give the adversary access to H'_0, H'_1 instead of H'. We also switch to the RO simulator instead of using H'_1. In addition, at the end of the game, the challenger calls the extractor on all tags t_i queried as part of the call to the BatchDEC oracle to obtain extracted values $(\mathsf{pk}^\mathsf{e}_i, \mathsf{ct}^\mathsf{e}_i, K^\mathsf{e}_i)$, $i \in [d]$. Note that H'_0 is never called as part of a BatchDEC query due to the modification in the previous game. These changes have no impact on the success of the game and thus

$$\Pr[\Gamma_1] = \Pr[\Gamma_2] \,.$$

Game Γ_3: We abort whenever the decapsulation oracle does not return $\perp$ but the extracted values $(\mathsf{pk}^\mathsf{e}_i, \mathsf{ct}^\mathsf{e}_i, K^\mathsf{e}_i)$ are not equal to $\perp$ or $(\mathsf{pk}_i, \mathsf{ct}_i, K'_i)$, where $K'_i = \mathsf{Decaps}(\mathsf{pk}_i, \mathsf{sk}_\mathsf{A}, \mathsf{ct}_i)$. By Property 8 of the extractable oracle, we have

$$\Pr[\Gamma_2] - \Pr[\Gamma_3] \leq \frac{40e^2(q_{H'} + d + 1)^3 + 2}{2^n} \,.$$

Game Γ_4: We move the extraction to the BatchDEC oracle, right after the corresponding tag verification. By Property 4 of the extractable oracle, we have

$$\Pr[\Gamma_3] - \Pr[\Gamma_4] \leq 8d(d + q_{H'})\sqrt{2/2^n} \,.$$

Game Γ_5: We modify the BatchDEC oracle s.t. we abort if all tag checks pass, i.e. $H'(\mathsf{pk}_i, \mathsf{ct}_i, K'_i) = t_i, \forall i \in [d]$ but some extracted value is equal to $\perp$, i.e. $(\mathsf{pk}^{\mathsf{e}}_i, \mathsf{ct}^{\mathsf{e}}_i, K^{\mathsf{e}}_i) = \perp$ for some $i \in [d]$. By Property 7 of the extractable oracle we have

$$\Pr[\Gamma_4] - \Pr[\Gamma_5] \leq d\frac{2}{2^n} .$$

Game Γ_6: We modify the BatchDEC oracle s.t. the queries to H' made for the tag verification are made after the corresponding extraction. By Property 8 of the extractable oracle we have

$$\Pr[\Gamma_5] - \Pr[\Gamma_6] \leq 8d\sqrt{2/2^n} .$$

Game Γ_7: We modify the previous game as follows. Let r be a bit set to 1 iff for all $i \in [d]$ $(\mathsf{pk}^{\mathsf{e}}_i, \mathsf{ct}^{\mathsf{e}}_i) = (\mathsf{pk}_i, \mathsf{ct}_i)$ and $\mathsf{Decaps}(\mathsf{pk}^{\mathsf{e}}_i, \mathsf{sk}_{\mathsf{A}}, \mathsf{ct}^{\mathsf{e}}_i) = K^{\mathsf{e}}_i$. Then, we change BatchDEC s.t. it returns $\perp$ if $r = 0$, otherwise it returns $H(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_i, \mathsf{ct}_i, K^{\mathsf{e}}_i)$ for all $i \in [d]$. In addition, the tag verification is now skipped. We argue this affects only negligibly the advantage of the adversary compared to the previous game:

- If BatchDEC returns $H(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_i, \mathsf{ct}_i, K'_i)$, $i \in [d]$ in Γ_6, then by the previous changes we know that $(\mathsf{pk}^{\mathsf{e}}_i, \mathsf{ct}^{\mathsf{e}}_i, K^{\mathsf{e}}_i) = (\mathsf{pk}_i, \mathsf{ct}_i, K'_i)$ for all $i \in [d]$, therefore BatchDEC returns $H(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_i, \mathsf{ct}_i, K'_i)$, $i \in [d]$ in Γ_7 as well.

- If BatchDEC returns $H(\mathsf{pk}_{\mathsf{A}}, \mathsf{pk}_i, \mathsf{ct}_i, K^{\mathsf{e}}_i)$, $i \in [d]$ in Γ_7, we know that $(\mathsf{pk}^{\mathsf{e}}_i, \mathsf{ct}^{\mathsf{e}}_i, K^{\mathsf{e}}_i) = (\mathsf{pk}_i, \mathsf{ct}_i, K'_i)$. In addition, for each $i \in [d]$, $t_i = H(\mathsf{pk}^{\mathsf{e}}_i, \mathsf{ct}^{\mathsf{e}}_i, K^{\mathsf{e}}_i)$ with probability $1 - \frac{2}{2^n}$ by Property 6 of the extractable oracle. Therefore, the tag verification would pass in Γ_6 with high probability and BatchDEC would return the same values in that game as well.

Overall, we have

$$\Pr[\Gamma_6] - \Pr[\Gamma_7] \leq d\frac{2}{2^n} .$$

Game Γ_8: Now we move all d queries to H' made in BatchDEC to the end of the game. By Property 8 of the extractable oracle, we have

$$\Pr[\Gamma_7] - \Pr[\Gamma_8] \leq 8d\, q_{H'}\sqrt{2/2^n} .$$

Note that we can now forget about the queries to H' we just moved to the end of the game.

Game Γ_9: We replace the real key K_0 and the challenge tag t^* by random values. We have

$\Pr[\Gamma_9] = \frac{1}{2}$. Applying the OW2H lemma on $H \otimes H'_0$, we get

$$\Pr[\Gamma_8] - \Pr[\Gamma_9] \leq 2(q_{H'} + q_H + d)\sqrt{\Pr[\Upsilon]}$$

where Υ is the same as Γ_9, except the challenger measures a random query made to $H \otimes H'_0$ and outputs 1 iff the query contains K^*, where K^* is the key encapsulated in ct^*. We can build an $\mathsf{OW\text{-}CPA}$ adversary $\mathscr{B}$ against sKEM_0 that wins with high probability when Υ outputs 1. The reduction works nearly as in the ROM proof: $\mathscr{B}$ receives a challenge ciphertext ct^* and two public keys $\mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}$, then it samples t^* and K^* at random and passes all these values to $\mathscr{A}$. Then, $\mathscr{B}$ can perfectly simulate $\mathsf{BatchDEC}$ as in Γ_9, except for the bit r that it can guess correctly with probability $\frac{1}{2}$. Finally, $\mathscr{B}$ measures a random query that was made to H or H' in the execution and outputs the corresponding value K. Overall, we have

$$\Pr[\Upsilon] \leq 2\mathsf{Adv}^{\text{ow-cpa}}_{\mathsf{sKEM}}(\mathscr{A})$$

which concludes the proof. □

A.4 Tables for Sender Keys (Chapter 6)

We provide here two tables (Table A.1 and Table A.2) that serve as a reference for the variables that we use in Chapter 6.

Variable	Description
General	
$\boldsymbol{G}$	sender keys group
γ	state
ID	user
ME	caller
m	application message
T	control message
(e,i)	message epoch counter
H	random oracle
Keys	
SK	sender key
spk	public signature key
ssk	secret signature key
ck	symmetric chain key
mk	message key
$\tau\mathsf{k}$	MAC key
Game Dictionaries	
M	ciphertexts
CH	challenged ciphertexts
T	control messages
E	message epochs
I	message indices
SM	skipped messages

Table A.1: Summary of variables used throughout Chapter 6 (part I).

Variable	Description
	Protocol
add	add command
crt	create command
rem	remove command
upd	update command
acc	acceptance bit
welcome	welcome ciphertext
ep	current epoch
i_{ck}	current index of user's chain key
i_{ME}	current epoch-specific index of caller's chain key
last-kc	last chain key
max-i_{ck}	maximum index of chain key
kc	chain key counter
no-SK	boolean indicator of whether sender key exists
rs	self-sampled randomness (for updates)
r	randomness sent (for updates)
$\vec{C}$	ciphertext vector
M	message
τ	MAC tag
σ	signature
C	ciphertext
γ_{2pc}	two-party channel state
C_{2pc}	two-party channel ciphertext
Upd-Ind	view of update initiator

Table A.2: Summary of variables used throughout throughout Chapter 6 (part II).

Bibliography

[AAB+21] Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part III*, volume 13044 of *Lecture Notes in Computer Science*, pages 222–253, Raleigh, NC, USA, November 8–11, 2021. Springer, Heidelberg, Germany.

[AAN+22a] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. DeCAF: Decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559, 2022. https://eprint.iacr.org/2022/559.

[AAN+22b] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 815–844, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

[ABH+21] Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. Analysing the HPKE standard. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 87–116, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[ACDJ23] Martin R. Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. Practically-exploitable Cryptographic Vulnerabilities in Matrix. In *2023 IEEE Symposium on Security and Privacy*, 2023.

[ACDT19] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. Cryptology ePrint Archive, Report 2019/1189, 2019. https://eprint.iacr.org/2019/1189.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 248–277, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[ACDT21a] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 1463–1483, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[ACDT21b] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. Cryptology ePrint Archive, Report 2021/1083, 2021. https://eprint.iacr.org/2021/1083.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part II*, volume 12551 of *Lecture Notes in Computer Science*, pages 261–290, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany.

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618, Santa Barbara, CA, USA, August 16–20, 2009. Springer, Heidelberg, Germany.

[ADJ24] Martin R. Albrecht, Benjamin Dowling, and Daniel Jones. Device-Oriented Group Messaging: A Formal Cryptographic Analysis of Matrix' Core. In *2024 IEEE Symposium on Security and Privacy (to appear)*, 2024.

[AHKM22] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 69–82, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[AJM20] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. Cryptology ePrint Archive, Report 2020/1327, 2020. https://eprint.iacr.org/2020/1327.

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part II*, volume 13508 of *Lecture Notes in Computer Science*, pages 34–68, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.

[AMPS22] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for Telegram. In *2022 IEEE Symposium on Security and Privacy*, pages 87–106, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press.

[AMT23] Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. Fork-resilient continuous group key agreement. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 396–429, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Heidelberg, Germany.

[ANPPP23] Benedikt Auerbach, Miguel Cueto Noval, Guillermo Pascual-Perez, and Krzysztof Pietrzak. On the cost of post-compromise security in concurrent continuous group-key agreement. In *TCC 2023: 21st Theory of Cryptography Conference, Part I*, Lecture Notes in Computer Science. Springer, Heidelberg, Germany, November 2023.

[ANS23] ANSSI. ANSSI views on the Post-Quantum Cryptography transition (2023 follow up). https://www.ssi.gouv.fr/uploads/2023/09/follow_up_position_paper_on_post_quantum_cryptography.pdf, 2023. Accessed: 28-09-2023.

[AP12] Jacob Alperin-Sheriff and Chris Peikert. Circular and KDM security for identity-based encryption. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012: 15th International Conference on Theory and Practice of Public Key Cryptography*, volume 7293 of *Lecture Notes in Computer Science*, pages 334–352, Darmstadt, Germany, May 21–23, 2012. Springer, Heidelberg, Germany.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. https://eprint.iacr.org/2015/046.

[Bal21] David Balbás. On Secure Administrators for Group Messaging Protocols, 2021. MSc Thesis.

[BBL+22] Olivier Blazy, Ioana Boureanu, Pascal Lafourcade, Cristina Onete, and Léo Robert. How fast do you heal? A taxonomy for post-compromise security in secure-channel establishment. Cryptology ePrint Archive, Report 2022/1090, 2022. https://eprint.iacr.org/2022/1090.

[BBLW22] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. Technical Report 9180, February 2022.

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, May 2018.

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[BCC+23a] Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 362–395, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Heidelberg, Germany.

[BCC+23b] Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. Cryptology ePrint Archive, Report 2023/880, 2023. https://eprint.iacr.org/2023/880.

[BCD+16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1006–1018, Vienna, Austria, October 24–28, 2016. ACM Press.

[BCG22] David Balbás, Daniel Collins, and Phillip Gajland. Analysis and Improvements of the Sender Keys Protocol for Group Messaging. *XVII Reunión española sobre criptología y seguridad de la información. RECSI 2022*, 265:25, 2022.

[BCG23a] David Balbás, Daniel Collins, and Phillip Gajland. Whatsupp with sender keys? analysis, improvements and security proofs. Cryptology ePrint Archive, Report 2023/1385, 2023. https://eprint.iacr.org/2023/1385.

[BCG23b] David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with Sender Keys? Analysis, improvements and security proofs. In *Advances in Cryptology –*

ASIACRYPT 2023, Part I, Lecture Notes in Computer Science. Springer-Verlag, December 7–11, 2023.

[BCK22] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Security analysis of the MLS key derivation. In *2022 IEEE Symposium on Security and Privacy*, pages 2535–2553, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press.

[BCLZL22] Renas Bacho, Daniel Collins, Chen-Da Liu-Zhang, and Julian Loss. Network-agnostic security comes for (almost) free in DKG and MPC. Cryptology ePrint Archive, Report 2022/1369, 2022. https://eprint.iacr.org/2022/1369.

[BCLZL23] Renas Bacho, Daniel Collins, Chen-Da Liu-Zhang, and Julian Loss. Network-agnostic security comes (almost) for free in DKG and MPC. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part I*, volume 14081 of *Lecture Notes in Computer Science*, pages 71–106, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Heidelberg, Germany.

[BCM23a] Subhadeep Banik, Daniel Collins, and Willi Meier. Near collision attack against grain V1. In Mehdi Tibouchi and Xiaofeng Wang, editors, *ACNS 23: 21st International Conference on Applied Cryptography and Network Security, Part I*, volume 13905 of *Lecture Notes in Computer Science*, pages 178–207, Kyoto, Japan, June 19–22, 2023. Springer, Heidelberg, Germany.

[BCM23b] Subhadeep Banik, Daniel Collins, and Willi Meier. Near collision attack against grain v1. Cryptology ePrint Archive, Paper 2023/884, 2023. https://eprint.iacr.org/2023/884.

[BCV22] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. Cryptology ePrint Archive, Report 2022/1411, 2022. https://eprint.iacr.org/2022/1411.

[BCV23] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In *USENIX Security 2023: 32nd USENIX Security Symposium*, pages 1253–1270, Anaheim, CA, USA, August 2023. USENIX Association.

[BDF+11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 41–69, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany.

[BDG+22] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In

Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022: 20th Theory of Cryptography Conference, Part II*, volume 13748 of *Lecture Notes in Computer Science*, pages 213–243, Chicago, IL, USA, November 7–10, 2022. Springer, Heidelberg, Germany.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BDR20] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part II*, volume 12551 of *Lecture Notes in Computer Science*, pages 198–228, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany.

[Bel06] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany.

[Beu20] Ward Beullens. Sigma protocols for MQ, PKP and SIS, and Fishy signature schemes. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 183–211, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany.

[BFG+20] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. Towards post-quantum security for Signal's X3DH handshake. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *SAC 2020: 27th Annual International Workshop on Selected Areas in Cryptography*, volume 12804 of *Lecture Notes in Computer Science*, pages 404–430, Halifax, NS, Canada (Virtual Event), October 21-23, 2020. Springer, Heidelberg, Germany.

[BFG+22a] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 784–813, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.

[BFG+22b] Jacqueline Brendel, Rune Fiedler, Felix Günther, Christian Janson, and Douglas Stebila. Post-quantum asynchronous deniable key exchange and the Signal handshake. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *PKC 2022: 25th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 13178 of *Lecture Notes in Computer Science*, pages 3–34, Virtual Event, March 8–11, 2022. Springer, Heidelberg, Germany.

[BFWW11] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011: 18th Conference on Computer and Communications Security*, pages 51–62, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

[BG21] Colin Boyd and Kai Gellert. A modern view on forward security. *The Computer Journal*, 64(4):639–652, 2021.

[BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84, 2004.

[BHJ+15] Christoph Bader, Dennis Hofheinz, Tibor Jager, Eike Kiltz, and Yong Li. Tightly-secure authenticated key exchange. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015: 12th Theory of Cryptography Conference, Part I*, volume 9014 of *Lecture Notes in Computer Science*, pages 629–658, Warsaw, Poland, March 23–25, 2015. Springer, Heidelberg, Germany.

[BJKS23a] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. An analysis of signal's pqxdh, 2023. https://cryspen.com/post/pqxdh/ Accessed: 23.10.23.

[BJKS23b] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. An analysis of signal's pqxdh, 2023.

[BJRW21] Katharina Boudgoust, Corentin Jeudy, Adeline Roux-Langlois, and Weiqiang Wen. On the hardness of module-LWE with binary secret. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 503–526, Virtual Event, May 17–20, 2021. Springer, Heidelberg, Germany.

[Bla07] Bruno Blanchet. Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, page 156, 2007.

[BLP+13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 575–584, Palo Alto, CA, USA, June 1–4, 2013. ACM Press.

[BLR04] Boaz Barak, Yehuda Lindell, and Tal Rabin. Protocol initialization for the framework of universal composability. Cryptology ePrint Archive, Report 2004/006, 2004. https://eprint.iacr.org/2004/006.

[BM99] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[BR94] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany.

[BRT23] Alexander Bienstock, Paul Rösler, and Yi Tang. Asmesh: Anonymous and secure messaging in mesh networks using stronger, anonymous double ratchet. *Cryptology ePrint Archive*, 2023.

[BRV20] Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 621–650, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.

[BSI23] BSI. Cryptographic Mechanisms: Recommendations and Key Lengths. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.html, 2023. Accessed: 28-09-2023.

[BSJ+15] Richard Barnes, Bruce Schneier, Cullen Fluffy Jennings, Ted Hardie, Brian Trammell, Christian Huitema, and Daniel Borkmann. Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement. RFC 7624, August 2015.

[BSJ+17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 619–650, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[BSSW02] Dirk Balfanz, Diana K. Smetters, Paul Stewart, and H. Chi Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *ISOC Network and Distributed System Security Symposium – NDSS 2002*, San Diego, CA, USA, February 6–8, 2002. The Internet Society.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[CCBR22a] Andrea Caforio, Daniel Collins, Subhadeep Banik, and Francesco Regazzoni. A small GIFT-COFB: Lightweight bit-serial architectures. In Lejla Batina and Joan Daemen, editors, *AFRICACRYPT 22: 13th International Conference on Cryptology in Africa*, volume 2022 of *Lecture Notes in Computer Science*, pages 53–77, Fes, Morocco, July 18–20, 2022. Springer Nature Switzerland.

[CCBR22b] Andrea Caforio, Daniel Collins, Subhadeep Banik, and Francesco Regazzoni. A small GIFT-COFB: Lightweight bit-serial architectures. Cryptology ePrint Archive, Report 2022/955, 2022. https://eprint.iacr.org/2022/955.

[CCD+20] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020.

[CCG16] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In Michael Hicks and Boris Köpf, editors, *CSF 2016: IEEE 29th Computer Security Foundations Symposium*, pages 164–178, Lisbon, Portugal, June 27–1, 2016. IEEE Computer Society Press.

[CCG+18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1802–1819, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[CCG+19] Katriel Cohn-Gordon, Cas Cremers, Kristian Gjøsteen, Håkon Jacobsen, and Tibor Jager. Highly efficient key exchange protocols with optimal tightness. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 767–797, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

[CCGB21a] Andrea Caforio, Daniel Collins, Ognjen Glamočanin, and Subhadeep Banik. Improving first-order threshold implementations of SKINNY. In *Progress in Cryptology–INDOCRYPT 2021: 22nd International Conference on Cryptology in India, Jaipur, India, December 12–15, 2021, Proceedings 22*, pages 246–267. Springer, 2021.

[CCGB21b] Andrea Caforio, Daniel Collins, Ognjen Glamocanin, and Subhadeep Banik. Improving first-order threshold implementations of SKINNY. Cryptology ePrint Archive, Report 2021/1425, 2021. https://eprint.iacr.org/2021/1425.

[CCHD23] Daniel Collins, Simone Colombo, and Loïs Huguenin-Dumittan. Real world deniability in messaging. Cryptology ePrint Archive, Paper 2023/403, 2023. https://eprint.iacr.org/2023/403.

[CD23] Wouter Castryck and Thomas Decru. An efficient key recovery attack on sidh. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 423–447. Springer, 2023.

[CDv+03] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Laih, editor, *Advances in Cryptology – ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 188–207, Taipei, Taiwan, November 30 – December 4, 2003. Springer, Heidelberg, Germany.

[CDV21] Andrea Caforio, F. Betül Durak, and Serge Vaudenay. Beyond security and efficiency: On-demand ratcheting with security awareness. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 649–677, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

[CEST22] Kelong Cong, Karim Eldefrawy, Nigel P. Smart, and Ben Terner. The key lattice framework for concurrent group messaging. Cryptology ePrint Archive, Report 2022/1531, 2022. https://eprint.iacr.org/2022/1531.

[CF11] Cas Cremers and Michele Feltz. One-round strongly secure key exchange with perfect forward secrecy and deniability. Cryptology ePrint Archive, Report 2011/300, 2011. https://eprint.iacr.org/2011/300.

[CGG+22] Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, and Adi Serendinschi. Crime and punishment in distributed byzantine decision tasks. *Cryptology ePrint Archive*, 2022.

[CGK+20a] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting

messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

[CGK+20b] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, Athanasios Xygkis, Matej Pavlovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Online payments by merely broadcasting messages (extended version). *arXiv preprint arXiv:2004.13184*, 2020.

[CHDN+24a] Daniel Collins, Loïs Huguenin-Dumittan, Ngoc Khanh Nguyen, Nicolas Rolin, and Serge Vaudenay. K-Waay: Fast and deniable post-quantum X3DH without ring signatures. In *USENIX Security 2024: 33rd USENIX Security Symposium*. USENIX Association, August 2024.

[CHDN+24b] Daniel Collins, Loïs Huguenin-Dumittan, Ngoc Khanh Nguyen, Nicolas Rolin, and Serge Vaudenay. K-waay: Fast and deniable post-quantum x3dh without ring signatures. Cryptology ePrint Archive, Paper 2024/120, 2024. https://eprint.iacr.org/2024/120.

[CHK21] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021: 30th USENIX Security Symposium*, pages 1847–1864. USENIX Association, August 11–13, 2021.

[CJSV22] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part II*, volume 13508 of *Lecture Notes in Computer Science*, pages 3–33, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.

[CK02] Ran Canetti and Hugo Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 143–161, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany. https://eprint.iacr.org/2002/120/.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 395–427, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany.

[CPZ20] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The Signal private group system and anonymous credentials supporting efficient verifiable encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1445–1459, Virtual Event, USA, November 9–13, 2020. ACM Press.

[CSM+20] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private group messaging with hidden access patterns. In *Annual Computer Security Applications Conference*, pages 84–99, 2020.

[CZ22] Cas Cremers and Mang Zhao. Provably post-quantum secure messaging with strong compromise resilience and immediate decryption. Cryptology ePrint Archive, Report 2022/1481, 2022. https://eprint.iacr.org/2022/1481.

[CZ24] Cas Cremers and Mang Zhao. Secure messaging with strong compromise resilience, temporal privacy, and immediate decryption. In *IEEE S&P*, 2024.

[DDF21] Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. MLS group messaging: How zero-knowledge can secure updates. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021: 26th European Symposium on Research in Computer Security, Part II*, volume 12973 of *Lecture Notes in Computer Science*, pages 587–607, Darmstadt, Germany, October 4–8, 2021. Springer, Heidelberg, Germany.

[DDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. LWE with side information: Attacks and concrete security estimation. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 329–358, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[DFG+23] Gareth T. Davies, Sebastian H. Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the WhatsApp end-to-end encrypted backup protocol. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 330–361, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Heidelberg, Germany.

[DFMS22] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Online-extractability in the quantum random-oracle model. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part III*,

volume 13277 of *Lecture Notes in Computer Science*, pages 677–706, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

[DG22] Samuel Dobson and Steven D. Galbraith. Post-quantum signal key agreement from SIDH. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings*, volume 13512 of *Lecture Notes in Computer Science*, pages 422–450. Springer, 2022.

[DGK06] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Deniable authentication and key exchange. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006: 13th Conference on Computer and Communications Security*, pages 400–409, Alexandria, Virginia, USA, October 30 – November 3, 2006. ACM Press.

[DGP22] Benjamin Dowling, Felix Günther, and Alexandre Poirrier. Continuous authentication in secure messaging. In *ESORICS*, 2022.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[DH20] Benjamin Dowling and Britta Hale. There can be no compromise: The necessity of ratcheted authentication in secure messaging. Cryptology ePrint Archive, Report 2020/541, 2020. https://eprint.iacr.org/2020/541.

[DH21] Benjamin Dowling and Britta Hale. Secure Messaging Authentication against Active Man-in-the-Middle Attacks. In *EuroS&P*, 2021.

[DHRR22] Benjamin Dowling, Eduard Hauck, Doreen Riepel, and Paul Rösler. Strongly anonymous ratcheted key exchange. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part III*, volume 13793 of *Lecture Notes in Computer Science*, pages 119–150, Taipei, Taiwan, December 5–9, 2022. Springer, Heidelberg, Germany.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, 2018. https://tches.iacr.org/index.php/TCHES/article/view/839.

[DNS04] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent zero-knowledge. *Journal of the ACM (JACM)*, 51(6):851–898, 2004.

[DSGHK23] Dana Dachman-Soled, Huijing Gong, Tom Hanson, and Hunter Kippen. Revisiting security estimation for lwe with hints from a geometric perspective. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 748–781, Cham, 2023. Springer Nature Switzerland.

[DV18] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. Cryptology ePrint Archive, Report 2018/889, 2018. https://eprint.iacr.org/2018/889.

[DV19] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *IWSEC 19: 14th International Workshop on Security, Advances in Information and Computer Security*, volume 11689 of *Lecture Notes in Computer Science*, pages 343–362, Tokyo, Japan, August 28–30, 2019. Springer, Heidelberg, Germany.

[EM19] Ksenia Ermoshina and Francesca Musiani. "standardising by running code": the signal protocol and de facto standardisation in end-to-end encrypted messaging. *Internet Histories*, 3(3-4):343–363, 2019.

[ESZ22] Muhammed F. Esgin, Ron Steinfeld, and Raymond K. Zhao. MatRiCT$^+$: More efficient post-quantum private blockchain payments. In *2022 IEEE Symposium on Security and Privacy*, pages 1281–1298, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press.

[fIS23] BSI German Federal Office for Information Security. Bsi tr-01102-1, 2023. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.html.

[FMB+16] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How secure is textsecure? In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 457–472. IEEE, 2016.

[Gal21] Tarek Galal. yowsup, Code Repository, 2021. https://github.com/tgalal/yowsup.

[GHP18] Federico Giacon, Felix Heuer, and Bertram Poettering. KEM combiners. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018: 21st International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 10769 of *Lecture Notes in Computer Science*, pages 190–218, Rio de Janeiro, Brazil, March 25–29, 2018. Springer, Heidelberg, Germany.

[Gie23] Hauke Gierow. Wire welcomes the publication of Messaging Layer Security as RFC 9420. https://wire.com/en/blog/wire-welcomes-the-publication-of-messaging-layer-security-as-rfc-9420/, 2023. Accessed: 28-09-2023.

[GM82] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *14th Annual ACM Symposium on Theory of Computing*, pages 365–377, San Francisco, CA, USA, May 5–7, 1982. ACM Press.

[GPA19] Lachlan J. Gunn, Ricardo Vieitez Parra, and N. Asokan. Circumventing cryptographic deniability with remote attestation. *Proceedings on Privacy Enhancing Technologies*, 2019(3):350–369, July 2019.

[Gua13] The Guardian. The NSA files. https://www.theguardian.com/us-news/the-nsa-files, 2013. Accessed: 28-09-2023.

[Gün90] Christoph G. Günther. An identity-based key-exchange protocol. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology – EUROCRYPT'89*, volume 434 of *Lecture Notes in Computer Science*, pages 29–37, Houthalen, Belgium, April 10–13, 1990. Springer, Heidelberg, Germany.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *SIGOPS*, 41(6):175–188, 2007.

[HKKP21] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An efficient and generic construction for Signal's handshake (X3DH): Post-quantum, state leakage secure, and deniable. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 410–440, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

[HKKP22] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An efficient and generic construction for Signal's handshake (X3DH): Post-quantum, state leakage secure, and deniable. *Journal of Cryptology*, 35(3):17, July 2022.

[HKP+21] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 1441–1462, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[HKP22] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. How to hide MetaData in MLS-like secure group messaging: Simple, modular, and post-quantum. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 1399–1412, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[Hog23] Giles Hogben. An important step towards secure and interoperable messaging. https://security.googleblog.com/2023/07/an-important-step-towards-secure-and.html, 2023. Accessed: 28-09-2023.

[HS20] Martha Norberg Hovd and Martijn Stam. Vetted encryption. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *Progress in Cryptology - INDOCRYPT 2020: 21st International Conference in Cryptology in India*, volume 12578 of *Lecture Notes in Computer Science*, pages 488–507, Bangalore, India, December 13–16, 2020. Springer, Heidelberg, Germany.

[HV22] Loïs Huguenin-Dumittan and Serge Vaudenay. On IND-qCCA security in the ROM and its applications - CPA security is sufficient for TLS 1.3. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part III*, volume 13277 of *Lecture Notes in Computer Science*, pages 613–642, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

[IETF23] (IETF) Internet Engineering Task Force. Messaging layer security, mailing list, 2023. https://mailarchive.ietf.org/arch/browse/mls/.

[IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 369–386, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[Jef20] Kee Jefferys. Session Protocol: Technical implementation details. https://getsession.org/blog/session-protocol-technical-information (accessed July 4th 2023), 2020.

[JLN19] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In Timothy M. Chan, editor, *30th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2439–2447, San Diego, CA, USA, January 6–9, 2019. ACM-SIAM.

[JMM19a] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 159–188, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[JMM19b] Daniel Jost, Ueli Maurer, and Marta Mularczyk. A unified and composable take on ratcheting. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part II*, volume 11892 of *Lecture Notes in Computer Science*, pages 180–210, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany.

[JMZ23] Haodong Jiang, Zhi Ma, and Zhenfeng Zhang. Post-quantum security of key encapsulation mechanism against cca attacks with a single decapsulation query. In *ASIACRYPT 2023*. Springer-Verlag, 2023.

[JS18] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 33–62, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[KBB17] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European symposium on security and privacy (EuroS&P)*, pages 435–450. IEEE, 2017.

[KGP23] Kien Tuong Truong Kenneth G. Paterson, Matteo Scarlata. Three Lessons From Threema: Analysis of a Secure Messenger. In *2023 USENIX Security Symposium*, 2023.

[KP05] Caroline Kudla and Kenneth G. Paterson. Modular security proofs for key agreement protocols. In Bimal K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 549–565, Chennai, India, December 4–8, 2005. Springer, Heidelberg, Germany.

[KPPW+21] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted TreeKEM, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy*, pages 268–284, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[KPRR23] Eike Kiltz, Jiaxin Pan, Doreen Riepel, and Magnus Ringerud. Multi-user cdh problems and the concrete security of naxos and hmqv. In *Cryptographers' Track at the RSA Conference*, pages 645–671. Springer, 2023.

[Kra05] Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 546–566, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.

[KS05] Jonathan Katz and Ji Sun Shin. Modeling insider attacks on group key-exchange protocols. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM CCS 2005: 12th Conference on Computer and Communications Security*, pages 180–189, Alexandria, Virginia, USA, November 7–11, 2005. ACM Press.

[KS23] Ehren Kret and Rolfe Schmidt. The pqxdh key agreement protocol, 2023. https://signal.org/docs/specifications/pqxdh/pqxdh.pdf.

[KT11] Ralf Küsters and Max Tuengerthal. Composition theorems without pre-established session identifiers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011: 18th Conference on Computer and Communications Security*, pages 41–50, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

[Laa16] Thijs Laarhoven. Search problems in cryptography: from fingerprinting to lattice sieving, February 2016. Proefschrift.

[LAZ19] Xingye Lu, Man Ho Au, and Zhenfei Zhang. Raptor: A practical lattice-based (linkable) ring signature. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*, volume 11464 of *Lecture Notes in Computer Science*, pages 110–130, Bogota, Colombia, June 5–7, 2019. Springer, Heidelberg, Germany.

[LLM07] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007: 1st International Conference on Provable Security*, volume 4784 of *Lecture Notes in Computer Science*, pages 1–16, Wollongong, Australia, November 1–2, 2007. Springer, Heidelberg, Germany.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 523–542, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[LN22] Vadim Lyubashevsky and Ngoc Khanh Nguyen. BLOOM: Bimodal lattice one-out-of-many proofs and applications. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part IV*, volume 13794 of *Lecture Notes in Computer Science*, pages 95–125, Taipei, Taiwan, December 5–9, 2022. Springer, Heidelberg, Germany.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[Lun18] Joshua Lund. Technology preview: Sealed sender for signal. https://signal.org/blog/sealed-sender/, 2018. Last visited on 13-09-2023.

[M+16] Moxie Marlinspike et al. Signal protocol repository, 2016.

[Mar14] Moxie Marlinspike. Private Group Messaging. https://signal.org/blog/private-groups/ (accessed Sep 5th 2023), 2014.

[Mar17] Moxie Marlinspike. Safety number updates. https://signal.org/blog/verified-safety-number-updates/, 2017. Accessed: 22-05-2022.

[MKMS22] Jose Maria Bermudo Mera, Angshuman Karmakar, Tilen Marc, and Azam Soleimanian. Efficient lattice-based inner-product functional encryption. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *PKC 2022: 25th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 13178 of *Lecture Notes in Computer Science*, pages 163–193, Virtual Event, March 8–11, 2022. Springer, Heidelberg, Germany.

[MM11] Daniele Micciancio and Petros Mol. Pseudorandom knapsacks and the sample complexity of LWE search-to-decision reductions. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 465–484, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.

[MMM02] Tal Malkin, Daniele Micciancio, and Sara K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 400–417, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[MP16a] Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm, 2016.

[MP16b] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. *Open Whisper Systems*, 283:10, 2016.

[NRS20] Moni Naor, Lior Rotem, and Gil Segev. Out-of-band authenticated group key exchange: From strong authentication to immediate key delivery. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *ITC 2020: 1st Conference on Information-Theoretic Cryptography*, pages 9:1–9:25, Boston, MA, USA, June 17–19, 2020. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Pei14] Chris Peikert. Lattice cryptography for the internet. In Michele Mosca, editor, *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014*, pages 197–219, Waterloo, Ontario, Canada, October 1–3, 2014. Springer, Heidelberg, Germany.

[PM16] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. *GitHub wiki*, 2016.

[PP22] Jeroen Pijnenburg and Bertram Poettering. On secure ratcheting with immediate decryption. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part III*, volume 13793 of *Lecture Notes in Computer Science*, pages 89–118, Taipei, Taiwan, December 5–9, 2022. Springer, Heidelberg, Germany.

[PR18] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 3–32, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[PRSS21] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. SoK: Game-based security models for group key exchange. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 148–176, Virtual Event, May 17–20, 2021. Springer, Heidelberg, Germany.

[PV06] Sylvain Pasini and Serge Vaudenay. An optimal non-interactive message authentication protocol. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 280–294, San Jose, CA, USA, February 13–17, 2006. Springer, Heidelberg, Germany.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93, Baltimore, MA, USA, May 22–24, 2005. ACM Press.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[RMA+23] Nathan Reitinger, Nathan Malkin, Omer Akgul, Michelle L Mazurek, and Ian Miers. Is cryptographic deniability sufficient*f* non-expert perceptions of deniability in secure messaging. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 274–292. IEEE, 2023.

[RMS18] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 415–429, London, UK, 2018. IEEE.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002: 9th Conference on Computer and Communi-*

cations Security, pages 98–107, Washington, DC, USA, November 18–22, 2002. ACM Press.

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 552–565, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany.

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994.

[Sha48] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[Sho94] Peter W. Shor. Polynominal time algorithms for discrete logarithms and factoring on a quantum computer. In Leonard M. Adleman and Ming-Deh A. Huang, editors, *Algorithmic Number Theory, First International Symposium, ANTS-I, Ithaca, NY, USA, May 6-9, 1994, Proceedings*, volume 877 of *Lecture Notes in Computer Science*, page 289. Springer, 1994.

[Sho00] Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.

[SRCM+22] John Scott-Railton, Elies Campo, Bill Marczak, Bahr Abdul Razzak, Siena Anstis, Gözde Böcü, Salvatore Solimano, and Ron Deibert. CatalanGate: Extensive Mercenary Spyware Operation against Catalans Using Pegasus and Candiru. https://citizenlab.ca/2022/04/catalangate-extensive-mercenary-spyware-operation-against-catalans-using-pegasus-candiru/, 2022. Accessed: 22-05-2022.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 617–635, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1461–1480, Virtual Event, USA, November 9–13, 2020. ACM Press.

[Sup22] Signal Support. Twilio Incident: What Signal Users Need to Know. https://support.signal.org/hc/en-us/articles/4850133017242, 2022. Accessed: 03-10-2022.

[SZ21] Alessandra Scafuro and Bihan Zhang. One-time traceable ring signatures. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021: 26th European Symposium on Research in Computer Security, Part II*, volume 12973 of *Lecture Notes in Computer Science*, pages 481–500, Darmstadt, Germany, October 4–8, 2021. Springer, Heidelberg, Germany.

[Tea21] NTRU Prime Risk-Management Team. Risks of lattice KEMs. https://ntruprime.cr.yp.to/latticerisks-20211031.pdf, 2021. Accessed: 28-09-2023.

[Tel] Telegram. Group Chats on Telegram. https://telegram.org/tour/groups.

[TGL+17] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440, 2017.

[UG15] Nik Unger and Ian Goldberg. Deniable key exchanges for secure messaging. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 1211–1223, Denver, CO, USA, October 12–16, 2015. ACM Press.

[UG18] Nik Unger and Ian Goldberg. Improved strongly deniable authenticated key exchanges for secure messaging. *Proceedings on Privacy Enhancing Technologies*, 2018(1):21–66, January 2018.

[US] US National Security Agency. Announcing the commercial national security algorithm suite 2.0. https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF.

[VGIK20] Nihal Vatandas, Rosario Gennaro, Bertrand Ithurburn, and Hugo Krawczyk. On the cryptographic deniability of the Signal protocol. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 20: 18th International Conference on Applied Cryptography and Network Security, Part II*, volume 12147 of *Lecture Notes in Computer Science*, pages 188–209, Rome, Italy, October 19–22, 2020. Springer, Heidelberg, Germany.

[Wei19] Matthew A Weidner. Group messaging for secure asynchronous collaboration, 2019.

[Wha20] WhatsApp. WhatsApp Encryption Overview Technical white paper, v.3, oct 2020. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[Wha23] WhatsApp. How to create and invite into a group. https://faq.whatsapp.com/3242937609289432/?cms_platform=web, 2023. Accessed: 29-09-2023.

[WKHB21] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2024–2045, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[WPBB23] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. Treesync: Authenticated group management for messaging layer security. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1217–1233. USENIX Association, 2023.

[YEL+21] Tsz Hon Yuen, Muhammed F. Esgin, Joseph K. Liu, Man Ho Au, and Zhimin Ding. DualRing: Generic construction of ring signatures with efficient instantiations. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part I*, volume 12825 of *Lecture Notes in Computer Science*, pages 251–281, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

[YGS23] Tarun Kumar Yadav, Devashish Gosain, and Kent Seamons. Cryptographic deniability: A multi-perspective study of user perceptions and expectations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3637–3654, 2023.

[YVCC23] Hailun Yan, Serge Vaudenay, Daniel Collins, and Andrea Caforio. Optimal symmetric ratcheting for secure communication. *The Computer Journal*, 66(4):987–1016, 2023.

[Zhe97] Yuliang Zheng. Digital signcryption or how to achieve cost(signature & encryption) ≪ cost(signature) + cost(encryption). In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 165–179, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Heidelberg, Germany.

Daniel Collins | Curriculum Vitae

Website: https://dcol.me | Nationality: Australian | Email: danielpatcollins@gmail.com

Education

PhD (Cryptography) (EPFL, Lausanne, Switzerland) Sep. 2019-Apr. 2024
- Advisor: Prof. Serge Vaudenay, LASEC (lasec.epfl.ch)

B.Sc. (Advanced Mathematics) (Honours) (University of Sydney, Australia) 2015-2018
Mathematics & Computer Science. Result: Honours Class I and the University Medal. Thesis: Byzantine fault tolerant boardroom elections without synchrony. Advisor: AProf. Vincent Gramoli

Research Visits

- FAU, Erlangen-Nuremberg, Germany: Lab of Paul Rösler. Jul. 2023
- CISPA, Saarbrücken, Germany: Lab of Julian Loss. Sept. 2022, Dec. 2022
- MPI for Security and Privacy, Bochum, Germany: Lab of Giulio Malavolta. Aug. 2022
- IMDEA Software Institute, Madrid, Spain: Lab of Dario Fiore. Apr. 2022

Academic Publications

- **K-Waay: Fast and Deniable Post-Quantum X3DH without Ring Signatures.** *Daniel Collins, Loïs Huguenin-Dumittan, Ngoc Khanh Nguyen, Nicolas Rolin, Serge Vaudenay.* USENIX Security 2024. Preprint: https://eprint.iacr.org/2024/120
- **WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs.** *David Balbás, Daniel Collins, Phillip Gajland.* ASIACRYPT 2023. Preprint: https://eprint.iacr.org/2023/1385. Poster at USENIX Security 2023. Preliminary Version: Best Paper Runner-Up at RECSI 2022.
- **Network-Agnostic Security Comes (Almost) for Free in DKG and MPC.** *Renas Bacho, Daniel Collins, Chen-Da Liu-Zhang, Julian Loss.* CRYPTO 2023. Preprint: eprint.iacr.org/2022/1369
- **On Active Attack Detection in Messaging with Immediate Decryption**. *Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, Serge Vaudenay.* CRYPTO 2023. Preprint: eprint.iacr.org/2023/880
- **Cryptographic Administration for Secure Group Messaging.** *David Balbás, Daniel Collins, Serge Vaudenay.* USENIX Security 2023. Preprint: eprint.iacr.org/2022/1411
- **Near Collision Attack against Grain v1.** *Subhadeep Banik, Daniel Collins, Willi Meier.* ACNS 2023. Preprint: eprint.iacr.org/2023/884
- **A Small GIFT-COFB: Lightweight Bit-Serial Architectures.** *Andrea Caforio, Daniel Collins, Subhadeep Banik, Francesco Regazzoni.* AFRICACRYPT 2022. Preprint: eprint.iacr.org/2022/955
- **Optimal Symmetric Ratcheting for Secure Communication.** *Hailun Yan, Serge Vaudenay, Daniel Collins, Andrea Caforio.* The Computer Journal (2022).
- **Improving First-Order Threshold Implementations of SKINNY.** *Andrea Caforio, Daniel Collins, Ognjen Glamocanin, Subhadeep Banik.* INDOCRYPT 2021. Preprint: eprint.iacr.org/2021/1425
- **Online Payments by Merely Broadcasting Messages.** *Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, Athanasios Xygkis.* DSN 2020 (Best Paper Runner-Up). Preprint: arxiv.org/abs/2004.13184
- **Anonymity Preserving Byzantine Vector Consensus.** *Christian Cachin, Daniel Collins, Tyler Crain, Vincent Gramoli.* ESORICS 2020.

Selected Presentations

- **Cryptographic Administration for Secure Group Messaging.** Swiss Crypto Day 2023 (8 September 2023).
- **On Active Attack Detection in Messaging with Immediate Decryption.** CRYPTO 2023; NYU Crypto Reading Group (16 August 2023).
- **Network-Agnostic Security Comes for Free in DKG and MPC.** CRYPTO 2023.
- **Near Collision Attack against Grain v1.** ACNS 2023.
- **Real World Deniability in Messaging.** *Joint work with Simone Colombo and Loïs Huguenin-Dumittan.* RWC 2023; FAU Erlangen-Nuremberg (26 June 2023). Preprint: eprint.iacr.org/2023/403
- **K-Waay: Fast and Deniable Post-Quantum X3DH without Ring Signatures**. RWPQC 2023.
- **Secure Messaging: Past, Present and Future.** IMDEA Software Institute (March 29 2022).
- **Anonymity Preserving Byzantine Vector Consensus**. ESORICS 2020 (online).

Academic Service

Program committee: AFRICACRYPT 2024

External reviewer: CRYPTO 2024, EUROCRYPT 2024, CANS 2023, CCS 2023, EUROCRYPT 2023, ASIACCS 2023, PKC 2022, CANS 2022, ASIACCS 2022, ANTS XV 2022, ASIACRYPT 2021, ACISP 2021, CANS 2020, SRDS 2018

Master thesis supervision at EPFL (overseen by Prof. Serge Vaudenay):
- Post-Quantum X3DH (*Nicolas Rolin*, Spring 2022; co-supervised with Loïs Huguenin-Dumittan)
- On Secure Administrators for Group Messaging Protocols (*David Balbás*, August 2021)

Master project supervision at EPFL: Andris Suter-Dörig (with Simone Colombo), David Dervishi, Olivier Becker, Oliver Tran Si An, Nathan Duchesne (with Loïs Huguenin-Dumittan), Sina Schaeffler, Alain Gautschi, Nicolas Rolin, Tanguy Rocher (with Andrea Caforio), Maxence Courtet, Dejan Kovac.

Teaching assistant experience:

- Advanced information, computation, communication I (1st year, EPFL)	Autumn 2022-2023
- Advanced cryptography (master, EPFL)	Spring 2021-2022
- Cryptography and security (master, EPFL)	Autumn 2021-2022
- Student seminar: security protocols and applications (master, EPFL)	Spring 2020-2021
- Concurrent algorithms (master, EPFL)	Autumn 2020-2021
- Theory of computation (2nd year, EPFL)	Spring 2019-2020
- Computer Science Project (2nd year, USyd)	2018
- Introduction to Artificial Intelligence (2nd year, USyd)	2018
- Distributed Systems (3rd year, USyd)	2018, 2019

Awards

- EDIC Fellowship, EPFL 2019
-- Semester 1 Supervisor: Prof. Rachid Guerraoui
-- Semester 2 Supervisor: Prof. Serge Vaudenay
- CommBank Cyber Prize for Exceptional Academic Performance in Cyber Security 2018
- University of Sydney Academic Merit Prize 2018
- School of Information Technologies Senior High Honour Roll 2016
- Farrand Science Scholarship 2015
- HSC All-rounder Achievers List member (90+ in all subjects) 2014

Other Employment

Research assistant (University of Sydney) Dec. 2018-Aug. 2019
- First co-author to a paper (Anonymity Preserving Byzantine Consensus) at ESORICS'2020.
- Contributor to the Datalog compiler Soufflé; web development

Summer researcher (University of Sydney) Nov. 2017-Feb. 2018
- Surveyed some verifiable secret sharing schemes; implementation in golang

Major development project (GeoInteractive; University of Sydney) Jul. 2017-Nov. 2017
- Developed a computer vision driven back-end paired with a React/Node web application leveraging AWS for feature detection and classification of pipe imagery

Programming intern (FluidIntel Pty Ltd) Nov. 2016-Feb. 2017
- Fully internationalised a Ruby on Rails web application and a C# application

Private tutor 2015-2018
- Taught students from years 7-12 in math and English and first/second year undergraduate courses

References

Available upon request.