Neutrality-Based Symmetric Cryptanalysis

THÈSE N^{O} 4755 (2010)

PRÉSENTÉE LE 3 SEPTEMBRE 2010

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE CRYPTOLOGIE ALGORITHMIQUE
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Shahram KHAZAEI

acceptée sur proposition du jury:

Prof. J.-P. Hubaux, président du jury
Prof. A. Lenstra, Prof. W. Meier, directeurs de thèse
Prof. A. Canteaut, rapporteur
Prof. A. Joux, rapporteur
Prof. S. Vaudenay, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2010

Abstract

Cryptographic primitives are the basic components of any cryptographic tool. Block ciphers, stream ciphers and hash functions are the fundamental primitives of symmetric cryptography. In symmetric cryptography, the communicating parties perform essentially the same operation and use the same key, if any. This thesis concerns cryptanalysis of *stream ciphers* and *hash functions*. The main contribution of this work is introducing the concept of *probabilistic neutrality* for the arguments of a function, a generalization of the definition of *neutrality*. An input argument of a given function is called neutral if it does not affect the output of the function. This simple idea has already been implicitly used in key recovery cryptanalysis of block ciphers and stream ciphers. However, in 2004, Biham and Chen explicitly used the idea of neutrality to speed up collision finding algorithms for hash functions. We call an input argument of a function *probabilistic neutral* if it does not have a "significant" influence on the output of the function. Simply stated, it means that if the input argument is changed, the output of the function stays the same with a probability "close" to one. We will exploit the idea of probabilistic neutrality to assess the security of several stream ciphers and hash functions. Interestingly, all our cryptanalyses rely on neutrality and/or probabilistic neutrality. In other words, these concepts will appear as a common ingredient in all of our cryptanalytic algorithms. To the best of our knowledge, this is the first time that the probabilistic neutrality has found diverse applications in cryptanalysis.

Keywords: cryptanalysis, cryptography, hash function, stream cipher.

Résumé

Les primitives cryptographiques sont les composants de base de tous les outils cryptographiques. Les Chiffrements par bloc, les chiffrements par flot et les fonctions de hachage sont les primitives fondamentales de la cryptographie symétrique. En cryptographie symétrique, chaque participant effectue essentiellement la même opération et emploie la même clef, s'il y en a. Cette thèse concerne la cryptanalyse des *chiffrements par flot* et des *fonctions de hachage*. La contribution principale de ce travail est de présenter le concept de *neutralité probabiliste* au niveau des arguments d'une fonction, qui est une généralisation de la définition de *neutralité*. Un argument d'entrée d'une fonction donnée est neutre s'il n'affecte pas la sortie de la fonction. Cette idée simple a déjà été implicitement employée dans la cryptanalyse des chiffrements par bloc et des chiffrements par flot pour trouver la clef secrète. Cependant, en 2004, Biham et Chen ont explicitement employé l'idée de la neutralité pour accélérer les algorithmes qui trouvent des collisions sur des fonctions de hachage. Nous qualifions un argument d'entrée d'une fonction de *neutre probabiliste* s'il n'a pas une influence "significative" sur la sortie de la fonction. Plus simplement, ça veut dire que si on change l'argument d'entrée, la sortie de la fonction reste la même avec une probabilité "proche" de 1. Nous exploiterons l'idée de la neutralité probabiliste pour évaluer la sécurité de plusieurs chiffrements par flot et fonctions de hachage. Toutes nos cryptanalyses se fondent sur la neutralité et/ou la neutralité probabiliste. Autrement dit, ces concepts apparaîtront comme des ingrédients communs pour tous nos algorithmes de cryptanalyse. Cela semble être la première fois que la neutralité probabiliste trouve des applications diverses en cryptanalyse.

Mots clés: cryptanalyse, cryptographie, fonction de hachage, chiffrement par flot.

بسی خرسندم که پدر و مادری فداکار و مهربان نصیبم گشته تا در سایه درخت پربار وجودشان بیاسایم، از ریشه آنها شاخ و برگ گیرم و از سایه وجودشان در راه کسب علم و دانش تلاش نمایم. والدینی که بودنشان تاج افتخاری است برسرم و نامشان دلیلی است بر بودنم چراکه این دو وجود همواره مایه هستی ام بوده اند. هم اینان بودند که به من آموختند که در این وادی زندگی پر از فراز و نشیب، با تاریکی جهل در ستیز باشم. من نیز همانگونه که زرتشت می اندیشد «برای نبرد با تاریکی شمشیر از نیام نمی کشم، ؛ بلکه چراغ می افروزم». به سهم خود تلاش کرده ام، همواره بیشتر بدانم و یاد بگیرم تا به نور نزدیکتر شوم. هرچند به گفته گاندی «کمابیش هرآنچه انجام دهیم ناچیز است؛ اما مهم این است که انجام پذیرد». از صمیم قلب و از نهایت جان، مایلم تا این هدیه ناچیز را پیشکش این آموزگاران بزرگ زندگی ام سازم.

Acknowledgements

I am heartily thankful to my supervisors, Arjen K. Lenstra and Willi Meier, whose encouragement, supervision and support from the preliminary to the concluding stages led to the finalization of this dissertation. Arjen created a very pleasant working atmosphere for us at LACAL and gave me great freedom to work on the subject of my interest. Willi accepted me to collaborate with his crypto group at FHNW which enabled me to deepen my understanding of the subject.

I am grateful to all my coauthors for having interesting discussions with. In particular, I would like to thank Simon Fischer. My joint works with Simon were a very encouraging start for my Ph.D.

I would like to thank all my colleagues at LACAL including our secretary Monique Amhof for all the nice moments we passed together on different occasions. In particular, I would like to thank Martijn Stam and Deian Stefan for proofreading my thesis. I appreciate Monique's patience for all the trouble she had in organizing my paperwork. I would also like to thank Nicolas Gama for his correction of my French abstract.

I am also thankful to the members of my jury for accepting to evaluate my work.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the studies. Most of all I would like to thank my parents for everything.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 Cryptography– goals and tenets

The need to protect valuable information goes back to the very old days. As a consequence, *cryptology* has developed over the centuries from an art, in which only few were skillful, into a science. There are several goals which security professionals seek to achieve. These include confidentiality, integrity, authentication, non-repudiation and privacy. Cryptology is known as the science of information protection against unauthorized parties [174] aiming to concretely address some of these goals. Cryptology is further subdivided into *cryptography* and *cryptanalysis*. ‘ Cryptography is the art of making *cryptosystems*, a major part of which concerns *confidentiality*, *authentication*, *integrity* and *non-repudiation*. When constructing cryptosystems, designers (known as *cryptographers*) share the task as follows. Some focus on designing well-established, low-level building blocks called *cryptographic primitives*. Others design *cryptographic protocols* by combining cryptographic primitives. Cryptographic primitives are extremely important; a vast amount of effort is devoted to studying the construction of some cryptographic primitives based on others. *Block ciphers*, *stream ciphers*, *hash functions*, *message authentication codes* and *digital signatures* are among the most fundamental primitives of cryptography. In cryptography, it is attempted to provide concrete definitions for these components. The properties of a cryptographic primitive are inspired by the security threats (known as *attacks*) which cryptographic protocols that use the primitive might be faced with. In other words, a cryptographic protocol imposes some requirements on the cryptographic primitives that it is using. Cryp-

tography is performed in one of the following two fashions: *symmetric cryptography* (or *secret key cryptography*) and *asymmetric cryptography* (or *public key cryptography*). Whereas symmetric key cryptography has been used since antiquity, public key cryptography appeared in the 1970s. Public key cryptography can be used to construct some cryptographic primitives, like digital signatures, which are out of scope of symmetric cryptography.

Cryptanalysis, on the other hand, concerns the analysis and evaluation of cryptosystems. A *cryptanalyst* examines cryptographic primitives and protocols to see if they have any *weakness*. A weakness might be a violation of any requirements which the designer imposed, or a new threat, previously unaddressed. Cryptanalysis is a very difficult task; hence, cryptanalysts usually first attempt to *break* simplified variants of their targets. History has proved that dealing with reduced versions of primitives is a reasonable start for cryptanalysts to understand how to cryptanalyze the target and incrementally reach the analysis of the full version. Moreover, handling simplified instances of primitives gives cryptographers an intuition of the strength of their designs. The struggle between cryptographers and cryptanalysts keeps the field of cryptography a very challenging and lively area.

Despite the elegant features of cryptography and the cryptographers' efforts formalizing security notions, security goals in real life might not be an easy aim to achieve. Real-world security systems are a combination of a complicated series of interactions. Modern systems have so many components and connections – some even unknown to the systems' designers, implementers, or users. Cryptography is not a panacea– you need a lot more than cryptography to have security– but it is essential [217].

This thesis concerns cryptanalysis of stream ciphers and hash functions, two cryptographic primitives which lie in the category of symmetric cryptography. Stream ciphers are deployed as pseudo-random number generators mainly to provide confidentiality, whereas hash functions are used in a wide spectrum of cryptographic applications such as message integrity, authentication and secure timestamping.

1.2 Ciphers and confidentiality

Confidentiality has ever been a main goal of secure communication. This goal attempts to restrict access to data to only those who have a legitimate need for it. A cryptosystem

which provides confidentiality is normally referred to as a *cipher*. However, these two words (cryptosystem and cipher) are often used interchangeably. A cipher is formally described as follows (see [225] for example).

Definition 1. *A cipher is a five tuple* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ *where*

- $\mathcal{P}$ *is the plaintext space,*
- $\mathcal{C}$ *is the ciphertext space,*
- $\mathcal{K}$ *is the key space,*
- *for each key* $k \in \mathcal{K}$*, there is an encryption rule* $E_k \in \mathcal{E}$ *and a corresponding decryption rule* $D_k \in \mathcal{D}$*,*
- *for every key* $k \in \mathcal{K}$*, the function pairs* $E_k : \mathcal{P} \to \mathcal{C}$ *and* $D_k : \mathcal{C} \to \mathcal{P}$ *satisfy* $D_k(E_k(p)) = p$ *for every plaintext* $p \in \mathcal{P}$*.*

The ultimate goal of a cipher is to enable two entities, who have already shared a key through a secure channel, to *securely* communicate over an insecure channel. The above definition of a cipher does not carry any notion of security. Claude Shannon was the first one who, in his seminal work in 1949, established the theory of secrecy by introducing two notions of security, *perfect security* and *computational security*.

1.2.1 Perfect security

Ideally, we would like that no information of the plaintext leaks from the ciphertext. In other words, any adversary, even with unlimited computational power, must not be able to reduce his ambiguity about the plaintext once he observes the corresponding ciphertext. In probability theory, this translates to $\Pr\{P = p|C = c\} = \Pr\{P = p\}$ for any $p \in \mathcal{P}$ and $c \in \mathcal{C}$, where P and C represent the random variables corresponding to the plaintext and ciphertext. From an information theory perspective, Shannon [218] has shown that in order to achieve perfect security the entropy of the plaintext, $H(P)$, must not exceed that of the key, $H(K)$, where $H(X)$ is the Shannon entropy of a random variable X.

Theorem 1 (Shannon, 1949)**.** *Perfect secrecy implies* $H(K) \geq H(P)$*.*

Vernam cipher [230] provides perfect security in which the plaintext is bit-wise XORed with a key of the same length to generate the ciphertext. However, the Vernam cipher is impractical in almost all real-world applications. The main bottleneck is that the key length must be at least as large as the message length itself. This problem, however, is inherent to perfect secrecy according to Theorem 1. In Vernam cipher a key cannot be used more than once, hence bearing the name of *one-time pad* as well. Moreover, the key must be a truly random sequence, a task that is not very easy to achieve in practice.

1.2.2 Computational security

Due to the intrinsic problem of key length in perfect security, in real-life applications people are using encryption schemes with keys much shorter than the message size to encrypt sensitive information. This alternative solution profits from the fact that cryptanalysts in practice have limited computational power. A cryptosystem is said to be *computationally secure* if the best algorithm for breaking it requires at least 2^n *operations*, where n is some specified number, *e.g.*, $n = 128$. Such a cryptosystem is then said to provide n bits of security or to have a security level of n bits. Even under this definition, no known practical cryptosystem can be proved to be computationally secure. In practice, however, there are two approaches. In the *provable security* approach, which is mostly taken in public key cryptography, the evidence of security is provided by means of a reduction. In other words, it is shown that if the cryptosystem can be broken in some specific way, then it would be possible to efficiently solve some well-studied problem that is believed to be difficult. Problems closely related to integer factorization and discrete logarithm, which respectively underlie the RSA [207] and ElGamal [119] public key cryptosystems, are such widely-accepted examples. Shannon suggested that breaking a good cipher should require "as much work as solving a system of simultaneous equations in a large number of unknowns of a complex type". It is known that solving a set of random multivariate polynomial equations over a finite field is an NP-hard problem [120]. Nevertheless, this problem is not a very suitable choice in practice for provable security since it may not be easy to prove that the resulting system of equations is random. However, Shannon's suggestion still leaves us a second approach which might be called *ad-hoc security*, commonly taken in symmetric cryptography. Using this approach, cryptosystems are designed in a way such that the

resulting system of equations seems to be complex. To this end, normally, a security parameter (*e.g.*, the number of rounds) is defined for the system such that increasing it makes the cryptosystem look more complex. The computational security evidence, on the other hand, is provided with respect to certain specific type of threats or the best methods cryptanalysts have managed to discover. Of course, security against one specific type of cryptanalytic algorithm does not guarantee security against some other methods. The astute reader then may ask why bother with symmetric cryptography? The answer comes from practical utilization. Symmetric cryptosystems are not only few hundred times faster than the known asymmetric ones, they also need keys of much shorter length to provide similar security levels. Finally, if enough care is taken, it is possible to design cryptosystems which are practically suitable for several decades.

1.3 Stream ciphers

In symmetric key cryptography, confidentiality is provided by using stream ciphers or block ciphers. Block ciphers are the most well known symmetric primitives and, as the name indicates, operate on fixed-length data blocks. The best-known block ciphers are the data encryption standard (DES) [185] and its replacement, the advanced encryption standard (AES) [186]. AES takes a 128-bit block of plaintext as input together with a secret key, which can be 128, 192 or 256 bits long, to produce a 128-bit ciphertext block. The oldest, simplest, and most natural way of encrypting larger amounts of data using a block cipher is known as the electronic code book (ECB) mode. To use AES in the ECB mode, the message is padded (if necessary) and divided into 128-bit blocks, and then each one is encrypted separately. Unfortunately, this easy way of utilizing a block cipher is insecure as identical plaintext blocks are encrypted into identical ciphertext blocks. In order to securely encrypt larger amounts of data with a block cipher, some other mode of operation such as cipher block chaining (CBC) is used.

Stream ciphers, in general, are preferred to block ciphers in software applications with very high throughput requirements, and in hardware applications with restricted resources such as limited storage, gate count, or power consumption. Very loosely speaking, stream ciphers can be thought of as cryptographically secure pseudo-random number generators with some extra bells and whistles. Modern stream ciphers use a secret key (typically 80 to 256 bits long) and a publicly known initial value (IV)

(typically 64 to 256 bits long) to produce a sequence of random-looking symbols (usually bits), known as the *keystream*. Incorporating a publicly known IV in a stream cipher not only avoids several time-memory trade-off threats [133, 54, 105], but also makes it possible to reuse the same key just by sending a new IV without having to agree on a new key. The sender and receiver must ensure that they are using the same IV. This can be done in a number of ways: by transmitting the IV along with the ciphertext, by agreeing on it in a handshake phase, by calculating it deterministically (usually incrementally), or with the help of a public parameter such as current time, packet number, *etc.* Stream ciphers are meant to imitate the Vernam cipher. Therefore, in order to encrypt a plaintext $\{p_i\}_{i=1}^N$ of length N bits, a random-looking binary keystream $\{z_i\}_{i=1}^N$ of the same length is produced and then XORed with the plaintext to produce the ciphertext $\{c_i\}_{i=1}^N$. That is,

$$c_i = p_i \oplus z_i,\ i = 1, 2, \ldots, N\ . \tag{1.1}$$

Trivially, decryption is performed by XORing the same keystream with the ciphertext in order to get the original plaintext. The keystream bits depend on the secret key K and the publicly known initial value IV. Moreover, in order to keep the keystream bits and ciphertext bits synchronized, the keystream bit at time i, *i.e.*, z_i, depends *either* on the value i *or* on a limited number of the previous ciphertext bits. Depending on either of these cases, stream ciphers are divided in two categories: *synchronous* stream ciphers and *self-synchronizing* stream ciphers.

Synchronous stream ciphers are adopted more widely in practice and, in the literature, are shortly referred to as stream ciphers. In a synchronous stream cipher the keystream is independent of the ciphertext and is simply produced as a mapping of the secret key K and initial value IV, see Figure 1.1. More precisely the ith keystream bit can be expressed as follows

$$z_i = f_i(K, IV),\ i = 1, 2, \ldots, N\ . \tag{1.2}$$

There is no error propagation in the synchronous stream ciphers, making them very suitable for situations where transmission errors are likely to occur. For example, if a single ciphertext bit is received erroneously, the receiver will decrypt only the corresponding plaintext bit wrongly whereas all the remaining plaintext bits will be decrypted correctly. However, the synchronous stream ciphers are highly sensitive to

insertion and deletion errors; the keystream sequence must be kept synchronized with the received ciphertext sequence at the receiver side for correct decryption. In other words, the receiver must know the exact position of the ciphertext bits so that the correct corresponding keystream bits are used in order to decrypt them. As mentioned, the main role of using an IV in a stream cipher is to be able to use the same key several times. Furthermore, in the case of synchronous stream ciphers, using a fresh IV is useful to ensure that the sender and receiver are resynchronized.

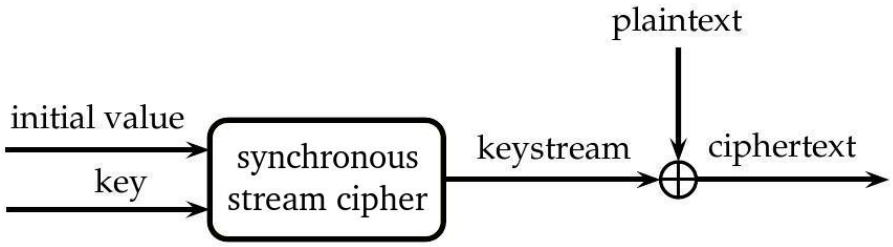

Figure 1.1: Synchronous stream cipher - A high level view of a synchronous stream cipher.

In contrast to synchronous stream ciphers, the keystream bits of the self-synchronizing stream ciphers do not depend on their position in the keystream sequence. In order to make the synchronization still possible, the previous ciphertext bits are used to produce a new keystream bit, see Figure 1.2. More precisely, each keystream bit is computed as a function of the secret key, the publicly known IV and the previous ciphertext bits. In order to limit the error propagation, in practice, the keystream is generated as a function of a limited number of r previous ciphertext bits, and of course the key and IV. That is,

$$z_i = f(K, IV, c_{i-r}, c_{i-r+1}, \ldots, c_{i-1}),\ i = 1, 2, \ldots, N\ , \tag{1.3}$$

where one can assume $c_{-r+1} = \cdots = c_0 = 0$ for example. The value r is known as the *resynchronization memory* of the cipher and should be kept quite small for practical reasons. A direct advantage of this property is that the receiver automatically synchronizes itself with the sender after having received r ciphertext bits correctly. This makes self-synchronizing stream ciphers suitable for the insertion/deletion channels,

that is, those channels which tend to drop message bits or add some garbage bits. However, the self-synchronizing stream ciphers suffer from single bit errors. Though, the effect is limited and only affects up to r keystream bits.

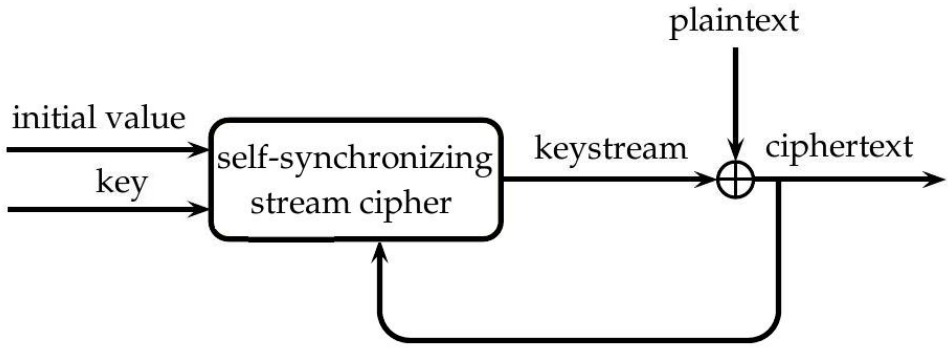

Figure 1.2: Self-synchronizing stream cipher - The figure shows a high level view of a self-synchronizing stream cipher. For practical reasons, each keystream bit must depend on a limited number of the previous cipher text bits.

Dedicated self-synchronizing stream cipher proposals are very rare [89, 215, 11, 213, 91, 152, 92], all of which have shown weaknesses [137, 244, 176, 138, 94, 139, 149, 144]. All the well-known widely used stream ciphers are synchronous stream ciphers. Real-world examples of synchronous stream ciphers include RC4 [3, 216] (used for WEP/WPA, by Bittorrent, and by SSL, to name a few), A5/1 and A5/2 [64] (used in GSM telephony standard) and E0 [61] (used in Bluetooth protocol). Synchronous stream ciphers commonly operate in two phases: *initialization* phase and *keystream generation* phase. The initialization algorithm computes the initial value of the internal state as a function of the key and IV. The key generation algorithm then expands the initial state into the keystream sequence. Traditionally, keystream generators for stream cipher applications are generally realized as autonomous finite state machines whose components, *state update function* and *output filter function*, can be key dependent. Nevertheless, in practice designers avoid using key dependent components for their finite state machines which not only simplifies the design but also avoids unwanted weak key classes, see Figure 1.3. It is important to mention that some block cipher modes of operation turn a block cipher into a stream cipher. Notably, the output

feedback mode (OFB) and the counter mode (CTR) give rise to synchronous stream ciphers whereas the cipher feedback mode (CFB) results in a self-synchronizing stream cipher.

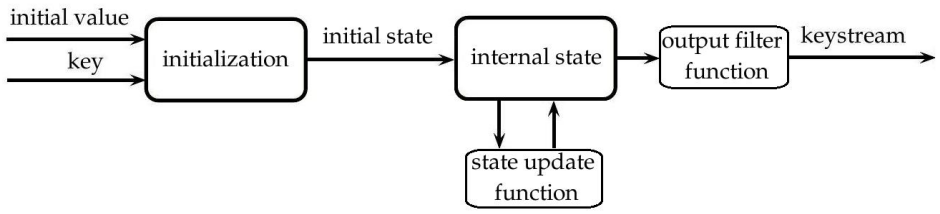

Figure 1.3: Finite state machine for stream cipher applications - The figure shows how stream ciphers can be realized based on finite state machines.

1.3.1 Cryptanalytic model and security of stream ciphers

As stream ciphers are supposed to imitate the Vernam cipher, their ultimate goal is to provide keystreams which *look* like truly random sequences. However, since the only source of the entropy for the keystream is the secret key, one uses the term *pseudo-random* versus truly random.

The security of pseudo-random number generators is then measured in terms of the best algorithm which can *distinguish* their keystreams from truly random sequences. We analyze stream ciphers in a very strong security model used in symmetric cryptography. In this model the cryptanalyst has access to many ciphertext/plaintext pairs which have been encrypted under a fixed key, unknown to the cryptanalyst. The cryptanalyst can choose the ciphertexts (or plaintexts) as well as the IV's (if any). The selection can be done adaptively if necessary. For block ciphers there is no IV and these models are known as *chosen plaintext* and *chosen ciphertext* scenarios. For stream ciphers, however, knowing the ciphertext and plaintext is equivalent to having access to the keystream. Consequently, for synchronous stream ciphers this model is simply referred to as *chosen IV cryptanalysis* scenario. In other words, the cryptanalyst has access to a large amount of the keystream generated under a number of different IV's and a fixed key. In contrast, for self-synchronizing stream ciphers, both *chosen IV chosen plaintext* and *chosen IV chosen ciphertext* cryptanalytic models make sense. As already mentioned, one goal of a cryptanalyst is to distinguish the stream cipher from a truly

random number generator. A more ambitious cryptanalyst might aim at predicting future keystream bits produced by the cipher for the same key, no matter if this happens by recovering the secret key, recovering the internal state of the cipher at some point, or otherwise. The quality of a *cryptanalytic algorithm* (distinguishing, key-recovery, internal state recovery, etc.) is determined according to the following four parameters:

- **Time complexity**: the amount of computation required for the algorithm to terminate.
- **Memory complexity**: the amount of memory needed for the algorithm to run.
- **Data complexity**: the total amount of keystream required to apply the algorithm.
- **Success probability**: the success probability of the algorithm.

Time, memory and data complexities are measured in some specified fixed unit. The amount of memory and data could be expressed in *bits* or *bytes* for example. However, the time unit is more delicate. Some cryptographers vaguely consider the time unit in terms of the required time to perform some simple operations, for example, modular addition or multiplication. Others mention it in terms of the average required time to test the correctness of a guess for the secret key, which sounds more reasonable. In this thesis, we use the later time unit unless otherwise specified. The success probability of a cryptanalytic algorithm is measured over random keys and over any source of randomness which might have been used in the cryptanalytic algorithm. In this thesis we use the following definition for the security of stream ciphers.

Definition 2. *A stream cipher, which uses k-bit long secret keys, is said to provide a security level of n bits, if there is no cryptanalytic algorithm with success probability 2^{-p}, $p \in [0, k]$, requiring time, data and memory complexities limited to 2^{n-p}.*

In symmetric cryptography, it is usually attempted to design k-bit ciphers which provide a security level of k bits.

1.3.2 State of the art of stream ciphers

Stream ciphers are expected to be faster in software or have a smaller implementation footprint than comparable block ciphers. This, in general, forces their structure to be

much simpler than block ciphers, making them not only more difficult to design but also more attractive targets for cryptanalysis. History has seen a noticeable number of weak stream ciphers. The most well-known examples are A5/1 and A5/2 stream ciphers used in the GSM cellular telephone standard. A5/2 is so weak that it can be broken instantly in a ciphertext-only scenario which requires just a few dozen milliseconds of the encrypted conversation [23]. Several cryptanalyses of A5/1 have been published, the most serious of which [57, 184] requires an expensive preprocessing stage to produce a huge table after which the cipher can be broken in a matter of minutes or seconds. Unfortunately the state of two other popular stream ciphers, RC4 and E0, is only a little less dreadful. Although they do not cower in front of cryptanalysts as A5/1 and A5/2 do, they cannot be considered secure according to modern standards due to several revealed weaknesses. We refer to [173, 192, 163] for the latest results on RC4 and E0, and to the references therein for older ones.

The first attempt to create a portfolio of secure stream ciphers was done by the NESSIE project [183], which ran from March 2000 to February 2003. NESSIE did not select any of the proposed stream ciphers for its portfolio, as none of the submissions withstood cryptanalysts' might. The second effort to identify new stream ciphers that may become suitable for widespread adoption was made by the eSTREAM project [106]. This project started in November 2004 and was finalized in April 2008 after three evaluation phases. At the end, four stream ciphers were chosen for software-oriented applications and another four for hardware implementations [19], see Table 1.1. Shortly after the portfolio was revised [20] due to a severe cryptanalytic result on F-FCSR family [10], published in [125]. As of April 2010, the eSTREAM portfolio consists of the remaining seven candidates.

Software	**Hardware**
HC-128 [241]	Grain [128, 127]
Rabbit [62]	Trivium [72]
Salsa20/12 [35]	MICKEY v.2 [21]
SOSEMANUK [28]	~~F-FCSR-H v2~~

Table 1.1: The eSTREAM final portfolio. The table shows the current eSTREAM portfolio. The original one included F-FCSR-H v2 but it was removed later.

We would like to emphasize that unlike the accomplished AES competition [182]

and the current SHA-3 competition [181], both organized by the U.S. National Institute of Standards and Technology (NIST), the goal of eSTREAM was not to develop a new international standard for stream ciphers. But rather merely to act as a focus for academic interest and an attempt to identify the best candidates among the various designs. While the eSTREAM recommended algorithms are still quite new and new weaknesses may yet be found, the portfolio can be considered to represent the current state of academic research on stream ciphers.

1.4 Hash functions

Very generally speaking, a hash function is an efficiently computable algorithm that maps arbitrary-length *messages* to fixed length outputs, called *message digests*. Hashing is traditionally used in non-cryptographic applications such as performance improvement and error checking [130]. In the field of cryptography, hash functions are among the most fundamental primitives and are sometimes referred to as *cryptographic hash functions*. Originally deployed to make digital signatures more efficient, cryptographic hash functions are now used in a broad spectrum of cryptographic applications including message integrity, authentication and secure timestamping. Aforementioned cryptographic applications as well as a host of other uses in various cryptographic protocols often rely on several assumptions about the underlying hash functions. If the hash function fails to be as secure as believed, then the application also fails to be secure in most cases. Ideally, we would like to have a cryptographic hash function which behaves like a *random oracle* [26]. A random oracle is a black box that responds to every query with a truly random response chosen uniformly from its output domain, except that for any specific query, it responds the same way every time it receives that query. Protocols, such as OAEP [27], that use cryptographic hash functions are often proved secure in the random oracle model [26], see also [219]. Unfortunately, random oracles do not actually exist in real life, and therefore proofs in the random oracle model only provide a heuristic for security [67].

1.4.1 Cryptographic properties of hash functions

In cryptographic practice, a hash function is typically a fixed public algorithm $H : \{0,1\}^\star \to \{0,1\}^n$, for some fixed n (typically between 128 and 512). The security of

most practical applications involving hash functions often relies on the following three properties of the underlying hash functions.

- **Collision resistance**: there is no algorithm running in time less than $2^{n/2}$ which can find two distinct messages x and y such that $H(x) = H(y)$.
- **Preimage resistance**: given a digest value $D \in \{0,1\}^n$, there is no algorithm running in time less than 2^n which can find a message x such that $H(x) = D$.
- **Second preimage resistance**: given a message x, there is no algorithm running in time less than 2^n which can find a message y different from x such that $H(x) = H(y)$.

In theory, however, these definitions are not considered to be precise enough. In order to study hash function security on a solid formal footing, one has to consider keyed hash functions [96, 97, 180]. In this case, the hash function is thought of as a collection or family of hash functions $\{H_K : \{0,1\}^\star \to \{0,1\}^n, K \in \mathcal{K}\}$. Some valuable attempts to concretely formalize the above three definitions were done in [208], in which seven notions of security related to these properties are introduced. In this thesis we, however, stick to the informal definitions. We would like to emphasize that we consider the time unit of a cryptanalytic algorithm in terms of hashing a short message, or less precisely the number of required simple operations. The upper-bound complexity $2^{n/2}$ for the collision resistance of an n-bit hash function is due to birthday paradox, see section 1.5.2.

1.4.2 Hash function design

The design of hash functions usually proceeds in two stages. First, one designs a *compression function* with fixed domain. A compression function transforms one short fixed-length input into a shorter fixed-length output. One then applies a *domain extension* method, also known as *mode of operation*, to the compression function in order to construct a hash function for messages of arbitrary length. In practice, the message length is limited to an enormous number, *e.g.*, 2^{64} or 2^{128} bits.

One of the popular ways to create a compression function is to base it on a block cipher. The emphasis on block cipher-based hashing can be understood both historically and practically. Block ciphers have long been the central primitives in symmetric

key cryptography, and there exists some measure of confidence in block cipher designs. From a practical perspective, one might like to reuse optimized code or hardware designs of block ciphers that have already been implemented. Collision resistance and preimage resistance are the minimum requirements one expects from a secure block cipher based compression function. The oldest block cipher based compression functions are commonly referred to as Davies-Meyer [174], Matyas-Meyer-Oseas [169] and Miyaguchi-Preneel [198, 177], see Figure 1.4. The security of block cipher based compression functions has been extensively studied in the literature; see, *e.g.*, [198, 60, 224].

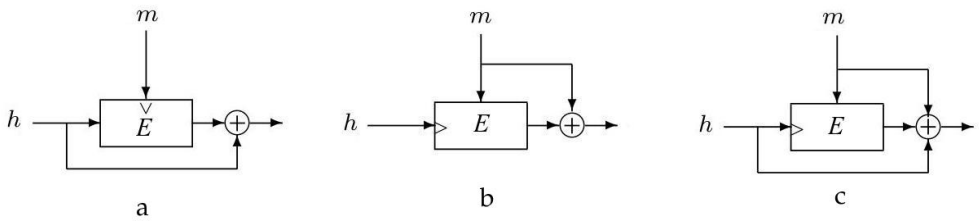

Figure 1.4: The three most famous block cipher based compression functions - a) Davies-Meyer $f(h,m) = E_m(h) \oplus h$, b) Matyas-Meyer-Oseas $f(h,m) = E_h(m) \oplus m$, and c) Miyaguchi-Preneel $f(h,m) = E_h(m) \oplus m \oplus h$. The compression functions $f : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$, mapping $2n$ bits into n bits, are both collision and preimage resistant, where E is an ideal block cipher with equal block and key length of n bits.

A compression function can be turned into a hash function by plugging it in some mode of operation. The most common way, known as the Merkle-Damgård [175, 97] construction, is based on iteratively updating a chaining variable. The standard Merkle-Damgård construction uses a compression function $f : \{0,1\}^b \times \{0,1\}^n \rightarrow \{0,1\}^n$, an initial n-bit chaining value h_0 and some injective padding function $P : \{0,1\}^\star \rightarrow \left(\{0,1\}^b\right)^\star \setminus \emptyset$. The message $M \in \{0,1\}^\star$ to be hashed is first padded (transformed) into a message whose length in bits is a positive multiple of b, *i.e.*, $P(M) = (m_0, \ldots, m_{l-1})$ with $l \geq 1$. The m_i's are called message blocks. Message blocks then recursively update the chaining value according to the relation $h_i = f(m_{i-1}, h_{i-1})$, $i = 1, \ldots, l$. The final chaining value is considered as the message digest, *i.e.*, $H(M) = h_l$; see Figure 1.5.

The security of the Merkle-Damgård construction and its variants has been intensively studied in the literature with respect to several cryptographic properties. The most widely used hash functions, including MD5 [205], SHA-1 [188] and SHA-2 family [189, 190], use a Davies-Meyer compression function with the strengthened Merkle-

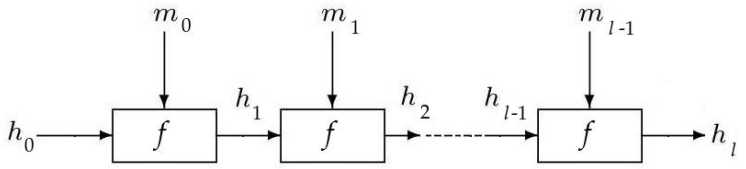

Figure 1.5: The standard Merkle-Damgård construction - The message M is first padded and divided into l message blocks $(m_0, \ldots, m_{l-1})$ where each m_i is b bits long. The message blocks iteratively update the initial chaining value h_0 to produce the message digest h_l.

Damgård mode of operation. The strengthened Merkle-Damgård requires to encode the length of the original message in the padding in order to preclude several trivial weaknesses [174]. It is easy to see that the strengthened Merkle-Damgård construction preserves collision-resistance when instantiated with a collision-resistant compression function [175, 97]. Other variants of the Merkle-Damgård construction can provide secure domain extenders for some specific properties of the corresponding primitive in question such as pseudorandomness [24], MAC unforgeability [6, 172] and randomness extraction [102]. However, the Merkle-Damgård construction does not work to yield a random oracle even if the underlying compression function is modeled as a random oracle. The most notable weaknesses of the Merkle-Damgård construction are due to the length extension [174], multi-collision [135], second preimage [147] and herding [145] attacks. In order to to prevent the weaknesses of the Merkle-Damgård construction, many researchers have considered alternative construction methods, see [59, 166, 39] for example. Building a framework to prove the indifferentiability of domain extenders from a random oracle provided that the underlying compression function is modeled as a random oracle was proposed in [171]. Since then there has been an increased popularity of the indifferentiability framework in the design and analysis of hash functions [80, 25, 76, 103, 40, 104].

1.4.3 History and the state of the art of practical hash functions

Rivest is probably the first to develop a publicly known dedicated cryptographic hash function. His design, called MD2 [142, 161], was developed in 1989. It has a unique non-Merkle-Damgård-based construction, suitable for 8-bit processors. MD2 was soon

superseded by other designs, not because it had been broken, but because its performance on 32-bit processors could not compete with the more modern designs. Nevertheless, MD2 is still part of several (de facto) standards, *e.g.*, PKCS#1 v2.1 [1] and Verisign [2]. Shortly after, Rivest designed MD4 [204] which is a Merkle-Damgård construction with a Davies-Meyer compression function. Despite early cryptanalysis of its simplified variants [98], MD4 inspired most of the hash functions designed afterwards. A newer version of MD4, called MD5 [205], was designed in 1991 to become one of the most widely used hash functions. Two years later, the U.S. National Institute of Standards and Technology (NIST) developed the Secure Hash Standard (SHA) [187], unofficially referred to as SHA-0, based on the same principles of MD4 and MD5. In 1995, NIST revised SHA-0 with a subtle tweak to introduce SHA-1 [188]. NIST published two new hash functions in 2001, called SHA-256 and SHA-512 [189], still following the design principles of MD4, MD5 and SHA-1. These hash functions along with two other variants, SHA-224 and SHA-384, are collectively referred to as the SHA-2 family [189, 190], the number indicating the digest length in bits. The later two members, SHA-224 and SHA-384, are respectively truncated variants of SHA-256 and SHA-512, computed with different initial chaining values. Nevertheless, the SHA-2 family members have not found widespread acceptance. In contrast, the popular hash functions MD5 and SHA-1 (with respective digests of size 128 and 160 bits) are still used almost universally.

Despite the quite fast progress in hash function design, it took cryptanalysts quite some time to gradually grasp and tackle them. Collisions for the compression function of MD5 were found in 1993 [99]; but it was not yet clear how to produce collisions for the hash function itself. The first collision for MD4 was found by Dobbertin in 1996 [101]. In 1998, Chabaud and Joux took a major step in hash function cryptanalysis [75] by introducing the linear differential cryptanalysis of hash functions. However, since the run time of the algorithm was too high to be carried out in practice, the cryptanalysts had to wait a few more years to produce real collisions with further progress in cryptanalysis. In 2004, Biham and Chen [44] introduced neutral bits to speed up the collision finding algorithms on SHA-0. Even though the collisions were not yet achievable, they produced near collisions for SHA-0 (two messages whose digests differ on a few number of bit positions). However, the year after proved to be fruitful for hash function cryptanalysis, mostly thanks to the Chinese cryptographer

Xiaoyun Wang. Improved collision cryptanalyses on SHA-0 were published [45, 236], and the first collision examples on the full MD5 hash function were produced [238]. Real collision examples on SHA-0 were published later the same year [239], and so was the first collision cryptanalysis on the full SHA-1 [237]. As of April 2010 real collision examples of SHA-1 have not been found, simply because the task is still enormous. Surprisingly, despite several attempts, MD2 has still remained quite resistant toward serious cryptanalysis [196, 211, 178, 156, 157].

Although the SHA-2 family of hash functions has not yet succumbed to the new collision-finding cryptanalytic algorithms that have plagued MD5 and SHA-1, the cryptographic community has lost their confidence in them. Although this may be mainly due to their design principles, being similar to those underlying MD5 and SHA-1, some undesirable properties of the Merkle-Damgård construction have also highlighted the situation. These unwanted weaknesses include the length extension, multi-collision, second preimage and herding attacks. Although the length extension property has been folklore knowledge for many years, the other ones were respectively discovered in 2004 by Joux [135], in 2005 by Kelsey and Schneier [147], and in 2006 by Kelsey and Kohno [145]. In response to the shocking cryptanalytic results on MD5 and SHA-1 and in order to improve the current state of the art concerning hash functions, NIST initiated the SHA-3 competition to develop a new set of hash functions. Expected to finish in 2012, the SHA-3 competition is still on-going. Unlike SHA-1 and SHA-2 which were designed internally, the new procedure goes through an open competition similar to that of AES [182]. The call for candidate algorithms was made in November 2007 [181] and 64 candidates were submitted by the call deadline, 31 October 2008. For the first round of the competition, 51 candidates were accepted of which 14 survived to the second round [202]. Table 1.2 shows the 14 second round candidates according to the elimination done by NIST on September 2009. NIST plans to select approximately five finalists for the third round of the competition in the fall of 2010. Due to the weaknesses of the Merkle-Damgård construction, many of the SHA-3 candidates are based on the alternative designs such as permutation-based [59, 210, 223, 209] and wide-pipe [166] constructions.

BLAKE	Grøstl	Shabal
BLUE MIDNIGHT WISH	Hamsi	SHAvite-3
CubeHash	JH	SIMD
ECHO	Keccak	Skein
Fugue	Luffa	

Table 1.2: SHA-3 second round candidates. The table includes the 14 surviving SHA-3 candidates of the second round. The list is going to shrink in the fall 2010 to about five candidates for the next round.

1.5 Techniques for symmetric cryptanalysis

Regardless of how computationally secure a cryptographic algorithm is, there always exist a set of generic cryptanalytic methods. The most trivial generic cryptanalytic methods are brute force search for keyed primitives and birthday paradox for finding hash collisions. These methods can be applied to relevant primitives, independent of the design details. On the other hand, there are many other cryptanalytic tools that, while still enjoying some sort of generality, specifically target the internal structure of the primitives. A well-designed primitive has at least one tunable parameter, such as the number of rounds. There is, however, a trade-off between security and efficiency of the primitive when playing with the tunable parameter. For example, increasing the number of rounds normally decreases efficiency while increasing security. A designer's job is to find a good balance between security and efficiency. In the following we give the intuition behind some of the well-known methods which are applicable to symmetric primitives. The interested reader may refer to [229] for other cryptanalytic techniques.

1.5.1 Brute force method

The brute force method is the most naïve generic cryptanalytic algorithm. Having a few ciphertext/plaintext pairs in hand, the correct key is simply found by exhaustively searching the key space. The complexity to recover the key with certainty is 2^k for k-bit long keys. This method is academically important since it determines the maximum level of security level which a security scheme can provide. In general, if the cryptanalyst restricts the search space to a subset of the keys of size 2^{k-p}, $p \in [0, p]$, he expects to recover the key with success probability 2^{-p}. This justifies our definition of a secure

stream cipher, see Definition 2.

1.5.2 Birthday paradox

In probability theory, the *birthday problem* pertains to the probability that in a set of randomly chosen people some pair of them would have the same birthday. In a group of at least 23 randomly chosen people, this chance is more than 50%. This problem is also known as *birthday paradox* since this result is counter-intuitive to many. In general, the probability that among d independent and identically distributed random variables with uniform distribution over a set of size N, two of them would be equal is about $1 - \exp(-\frac{d(d-1)}{2N})$.

A direct application of this paradox is the collision finding problem for hash functions, first pointed out by Yuval [243] in 1979. For a hash function with n-bit message digests, the number of message digests one needs to observe before a collision is found is approximately $2^{n/2}$. For this application, one can substantially reduce the memory requirements by translating the problem to the detection of a cycle in an iterated mapping [201]. See [228] for an efficient parallel collision search algorithm.

1.5.3 Time-memory trade-offs

Time-memory trade-offs were first introduced by Hellman [129] in 1980 as a generic way to cryptanalyze block ciphers, but can be generalized to the general problem of inverting one-way functions. Hellman's method uses a precomputed table of total size M which allows to recover a block cipher's key in time complexity T. In a precomputation phase several starting points are randomly chosen. Then from each starting point a chain is constructed to get an end point. The chains are built by successively applying a function which is derived from the underlying block cipher. Then the end point/starting point pairs are sorted based on the end point values and stored in a Table of size M. In the on-line phase, which takes time T, the goal is to find the predecessor of a given point. This is done by constructing a long chain of points from the given point. Each time a new point is computed, it is looked up in the precomputed table. If it exists, in the table, from the corresponding starting point, the predecessor of the given point is found. It can be shown that the values of M and T satisfy the relation $TM^2 = 2^{2k}$ (up to logarithmic factors), where k is the key size of the block cipher. A convenient choice

of M and T is $M = T = 2^{\frac{2}{3}k}$. Nonetheless, the precomputation phase still requires a time complexity of 2^k encryptions.

For stream ciphers based on finite state machines whose structures are independent of the key, there is a well-known time-memory-data trade-off cryptanalysis [17, 121]. This threat does not apply to modern stream ciphers since it is worse than exhaustive key search when a large internal state is deployed. This method can be explained as follows. For a stream cipher with an s-bit internal state, the goal is to invert the function which maps an internal state to the first s bits of the keystream generated from that state. To this end, in a preprocessing phase, the cryptanalyst chooses $2^{s/2}$ random inputs (internal states) and applies this function to them to get their corresponding outputs (the prefix of length s of the corresponding keystream). He then sorts the output/input pairs in approximate time $2^{s/2}$ and stores them, based on the output value, in a table of the same size. The cryptanalyst then observes a keystream of length $2^{s/2}$, from which he can construct about $2^{s/2}$ overlapping output values. For each output value he can examine the sorted table and find the corresponding internal state value if it exists in the table. According to the birthday paradox a match between the saved points and the observed output values is quite likely. Since, time, memory and data complexities of this cryptanalytic algorithm is about $2^{s/2}$, it can be easily thwarted by choosing an internal state which is at least twice bigger than the key size. Other variants of time-memory-data trade-offs on stream ciphers can be found in [55, 133, 54, 105].

1.5.4 Differential cryptanalysis

Differential cryptanalysis is a general cryptanalytic method applicable primarily to block ciphers, but also to stream ciphers and cryptographic hash functions. In a very broad sense, it is the study of how specific differences in the input of a particular transformation affect the resulting output. Differential cryptanalysis was first publicized by Biham and Shamir in 1990 [47] to analyze reduced-round variants of DES [47, 48, 52] in a chosen plaintext scenario, followed by the first cryptanalysis on DES, in 1991, which recovers the key faster than exhaustive search [51]. Nevertheless, it turned out that IBM was already aware of this method [79] and so DES was designed to be resistant to differential cryptanalysis.

Differential cryptanalysis studies the differences, usually by means of the XOR, as they evolve through the various rounds and different operations of a symmetric primitive. One first considers a transformation $Z = F(X)$ which is related to the primitive specification. In the case of block ciphers, the input X is the combination of the secret key and the plaintext, *i.e.*, $X = (K, P)$. Moreover, the mapping F is constructed by tracing the network of transformations which gradually convert the plaintext into the ciphertext. Any non-random behaviour of this transformation can be exploited to recover the secret key or part of it. In particular, a cryptanalyst tries to find an input difference Δ_x and an output difference Δ_z. The input difference Δ_x is the combination of the differences in the plaintext and the difference in the key which we assume to be zero, *i.e.*, $\Delta_x = (0, \Delta_p)$. The cryptanalyst then estimates the probability of the *differential* $(\Delta_p \rightarrow \Delta_z)$, *i.e.*, $\Pr\{F(K, P) \oplus F(K, P \oplus \Delta_p) = \Delta_z\}$. The probability of the differential $(\Delta_p \rightarrow \Delta_z)$ can be computed over the whole input domain or a subset of it. Moreover, for block ciphers, and in general for keyed symmetric primitives, the mapping F is constructed in a way that the output difference Δ_z is a function of some part of the key, called *subkey*. For example, when the ciphertext is partially decrypted using the last subkey, one gets the output value of F. Any abnormality in the probability of the differential can be exploited to recover the subkey or get some information about it by means of statistical methods. To this end, the cryptanalyst collects many quadruples of plaintext pairs and their corresponding ciphertext pairs where each plaintext pair has the desired difference Δ_p. It is possible that some of the plaintext pairs, which already satisfy the required input difference Δ_x for the transformation F, provide the desired output difference Δ_z as well. Having at least one such pair, called a *right pair*, is essential to eliminate some of the wrong keys. The exact number of the required right pairs is determined by the best statistical method which the cryptanalyst can apply to recover the subkey.

Several refinements to differential cryptanalysis have attempted to improve the technique for some circumstances. A variant of differential cryptanalysis uses an extended form of differences, in which some of the bits of the output difference are not fixed. Because part of the output difference is left unspecified, this is equivalent to clustering several differentials together. This type of cryptanalysis is called truncated differential cryptanalysis [155]. Another extension of differential cryptanalysis takes advantage of differentials which occur with probability zero [153, 43]; these differentials are called

impossible differentials [43]. There are also non-XOR differential cryptanalysis variants [160], such as modular subtraction, modular division, or a combination of different differences in various places. The generalization of differential cryptanalysis which considers differences between differences is called *higher-order differential cryptanalysis* [155]. Higher-order differences prove to be successful in several cases where ordinary differential cryptanalysis is not applicable. Furthermore, there are also cryptanalytic methods that combine differentials in various ways, the most promising of which are the boomerang [232], amplified boomerang [146], and rectangle [46] cryptanalyses. Lastly, in related key cryptanalysis [42, 154], which is less desirable and highly disputed, a nonzero difference in the keys is also permitted.

Differential cryptanalysis of block ciphers and stream ciphers is closely related to that of hash functions, but also disparate in some aspects. The dissimilarities were recognized in early works [50, 49, 41, 197] and were later developed in hash cryptanalyses [98, 99, 101, 75, 212, 44, 45, 236, 238, 239, 237, 73, 179, 70, 168], still used extensively. A major dissimilarity is that the goal of the cryptanalyst is different. In the first case, the aim is to obtain the secret key while in the later case there is no key to be recovered and the goal is mostly to find a hash collision. Another main difference is the freedom which the cryptanalyst has in playing with the plaintext/message. In differential cryptanalysis of block ciphers, the only restriction on the chosen plaintext pairs is their imposed difference. Remember that the input X is the combination of the secret key and the plaintext, $X = (K, P)$. Although for a plaintext pair $(P, P \oplus \Delta_p)$, the cryptanalyst has still the ability to choose the plaintext value P, this freedom can be hardly exploited in practice (*e.g.*, to increase the chance of getting a right pair). This is mainly because of the fact that the secret key influences the way the input difference propagates through the cipher from the very beginning. For hash functions, however, there is no secret key involved and full control over the input X is available. Any input X for which $F(X \oplus \Delta_x) \oplus F(X) = \Delta_z$, called *conforming message*, yields a collision for the underlying hash function. The recent collision finding algorithms have investigated extensive methods to use this freedom in order to efficiently find such conforming messages by means of satisfying some *conditions*. These methods are referred to as *message modification* techniques which apparently have been used by Xiaoyun Wang as early as 1997 [233, 234]. However, they were brought to the attention of the

international cryptographic community only in 2005 [236, 238, 239, 237]. Message modification techniques use concepts such as neutral bits [44], semi-neutral bits [167, 226] and tunnels [151]. When it comes to implementation, backtracking algorithms [38, 73] are used to find a conforming message.

1.5.5 Integral cryptanalysis

The dual of differential cryptanalysis is *integral cryptanalysis* [158]. There are several cryptanalytic methods related to integral cryptanalysis in the literature, including the square [93], saturation [165], AIDA [231] and Cube [100] cryptanalytic methods. These methods are collectively referred to as multiset cryptanalytic method [56]. Unlike differential cryptanalysis, which uses pairs of chosen inputs with a fixed difference, integral cryptanalysis uses sets or even multisets of chosen inputs. In most of the cases the input is divided into two parts, of which one part is held constant and the other part varies through all possibilities. Furthermore, in integral cryptanalysis one considers the sum of the outputs over the multiset, whereas in differential cryptanalysis the subtraction between the output pair is the center of concentration. For example, if we are considering integral cryptanalysis of the function $F(X)$ where $X = (x_0, x_1, x_2, x_3) \in \mathbb{Z}_{256}^4$, we might choose 256 different inputs that are the same in the last three positions. In other words we compute $\sum_{x \in \mathbb{Z}_{256}} F(x, c_1, c_2, c_3)$ where c_1, c_2, c_3 are some constants. For finite fields with characteristic two, however, the subtraction and addition are the same. Therefore, integral cryptanalysis can be considered as a variant of the previously mentioned higher-order differential cryptanalysis [155].

1.5.6 Algebraic cryptanalysis

The basic principle of algebraic cryptanalysis goes back to Shannon's seminal work [218] in 1949. It consists of expressing the whole cryptographic algorithm as a large system of multivariate algebraic equations, which can then be solved to recover, *e.g.*, the secret key. Despite much research in algebraic cryptanalysis of block ciphers [84, 78, 4, 53], thus far, the proposed methods have had very limited success in targeting modern block ciphers. In fact, there is no modern block cipher (with practical relevance) that has been successfully cryptanalyzed using algebraic cryptanalysis faster than with other techniques. This is also true for the case of algebraic cryptanalysis of hash functions. There is limited work on algebraic cryptanalysis against hash functions

and algebraic techniques have thus far been relatively unexplored for hash function cryptanalysis. In contrast to block ciphers and hash functions, algebraic cryptanalysis has been successfully used in the analysis of several LFSR-based stream ciphers [83, 81, 8, 7, 82]. Algebraic cryptanalysis as a method for stream cipher cryptanalysis was originally introduced by Courtois and Meier [83], and it generally applies to nonlinear combiner generators and nonlinear filter generators. Algebraic cryptanalysis of these keystream generators exploits the fact that each new keystream bit gives rise to a new simple equation on the initial state. Thus, the cryptanalyst can collect a large number of bits from the keystream to construct a system of equations, which can then be solved. The interested reader is referred to [107] for an overview of the main techniques used for solving systems of multivariate polynomial equations, with special focus on methods used in cryptanalysis.

1.6 Contribution of the thesis

This thesis concerns cryptanalysis of stream ciphers and hash functions with emphasis on some of the eSTREAM [106] and NIST SHA-3 [181] candidates.

1.6.1 Target algorithms

In particular, we have contributed to the cryptanalysis of reduced-round variants of the following algorithms.

1. **Salsa20 [31, 35]:** One of the eSTREAM final candidates in the software profile, designed by D. Bernstein in 2005.

2. **ChaCha [30, 33]:** A variant of the Salsa20 stream cipher, designed by D. Bernstein in 2008.

3. **Trivium [71, 68, 72]:** One of the eSTREAM final candidates in the hardware profile, designed by C. De Cannière and B. Preneel in 2005.

4. **Grain-128 [126, 127]:** One of the eSTREAM phase 3 candidates designed by M. Hell, T. Johansson, A. Maximov and W. Meier in 2005. It is a variant of the Grain [128, 127] stream cipher, one of the eSTREAM final candidates in the hardware profile.

5. **Klimov-Shamir SSSC [152]:** A self-synchronizing stream cipher proposed by A. Klimov and A. Shamir in 2005 at the Fast Software Encryption workshop (FSE'05).

6. **MD6 [206]:** One of the first round candidates of the NIST SHA-3 competition, designed by Rivest *et al.* in 2008.

7. **CubeHash [34]:** One of the NIST SHA-3 candidates, designed by D. Bernstein in 2008. It is currently one of the promising second round candidates.

1.6.2 Previous work and main contributions

The main contribution of this work is introducing the concept of *probabilistic neutrality* for the arguments of a function, a generalization of definition of *neutrality*. Interestingly, our cryptanalyses of all the aforementioned algorithms rely on neutrality and/or probabilistic neutrality. An input argument of a given function is called neutral if it does not affect the output of the function. This simple idea has already been used in key recovery cryptanalysis of block ciphers as well as in collision finding algorithms for hash functions.

To explain how neutrality is used in cryptanalysis of block ciphers, consider a block cipher $C = E_K(P)$ which maps the plaintext P into the ciphertext C using the secret key K. As explained in section 1.5.4, a cryptanalyst derives a function $Z = F(K, P)$ from the block cipher such that the output of F can also be computed from the ciphertext, by partially decrypting it. In other words the output of F can be related to the ciphertext and (some part of) the key. Let's assume $Z = G(K, C)$, see Figure 1.6. This method is also known as the *meet-in-the-middle* technique. A cryptanalyst then finds some nonrandom behaviour of F (*e.g.*, using differential cryptanalysis) in a chosen plaintext scenario to get some information about the key with the help of G. The key point is that the function G does *not* depend on the whole key. In other words, only some part of the key, known as *subkey*, suffices to partially decrypt the ciphertext to get the output value of the function F. In other words a large part of the key is neutral for G. Hence, in order to detect the correct subkey, the subkey space is exhaustively searched instead of the whole key space. This application of neutrality is very trivial and has mostly been used implicitly.

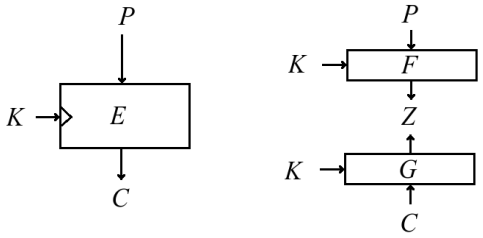

Figure 1.6: Meet-in-the-middle technique - For a block cipher $C = E_K(P)$ one derives two less complex functions F and G such that $F(K, P) = G(K, C)$. The function F has some nonrandom behaviour which can be detected using G. The function G does not depend on the whole key in practice.

However, in 2004 Biham and Chen explicitly used the idea of neutrality to speed up collision finding algorithms for hash functions [44]. The principle of their idea can be roughly explained as follows. As mentioned in section 1.5.4, the collision finding algorithms for hash functions are normally translated into finding a conforming message. That is, one needs to find a message X such that $F(X) \oplus F(X \oplus \Delta_x) = \Delta_z$ where the function F is derived based on differential cryptanalysis of the underlying hash function. If p is the probability that a random message X is a conforming one, a solution can be found after $1/p$ random tries. However, the task can be performed faster in two steps by dividing the message in two parts, let say $X = X_1||X_2$. In the first step the cryptanalyst finds the message part X_1 such that some conditions are satisfied. These conditions are such that they are not violated once choosing the second message parts X_2. In other words, the conditions are neutral with respect to X_2. In the second step, provided that X_1 has been chosen to fulfill those conditions of the first step, X_2 is independently found such that $X_1||X_2$ is a conforming message. The probability to randomly accomplish the first and second phases is p_1 and p_2 respectively where $p \approx p_1 p_2$. Therefore, the effort is reduced from $1/(p_1 p_2)$ to $1/p_1 + 1/p_2$ (p_1 and p_2 are very small numbers). The multi-block technique [45, 236, 238] as well as cryptanalysis based on semi-neutral bits [167, 226] can be seen as applications of the neutrality concept in hash function cryptanalysis.

We call an input argument of a function *probabilistic neutral* if it does not have a "significant" influence on the output of the function. Very loosely speaking, it means that if the input argument is changed, the output of the function stays the same with a probability "close" to one. We exploit the idea of probabilistic neutrality to crypt-

analyze several stream ciphers and hash functions. To the best of our knowledge this is the first time that the probabilistic neutrality has found a concrete application in cryptanalysis.

1.6.3 Publications and thesis outline

Here is a list of articles which I have published, with the help of my coauthors, during my Ph.D. studies.

1. Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. "New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba". In Kaisa Nyberg, editor, *Fast Software Encryption*, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers, volume 5086 of *Lecture Notes in Computer Science*, pages 470–488. Springer, 2008.

2. Simon Fischer, Shahram Khazaei, and Willi Meier. "Chosen IV Statistical Analysis for Key Recovery Attacks on Stream Ciphers". In Serge Vaudenay, editor, Progress in Cryptology – *AFRICACRYPT* 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11–14, 2008. Proceedings, volume 5023 of *Lecture Notes in Computer Science*, pages 236–245. Springer, 2008.

3. Shahram Khazaei and Willi Meier. "New Directions in Cryptanalysis of Self-Synchronizing Stream Ciphers". In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, Progress in Cryptology – *INDOCRYPT* 2008, 9th International Conference on Cryptology in India, Kharagpur, India, December 14–17, 2008. Proceedings, volume 5365 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2008.

4. Eric Brier, Shahram Khazaei, Willi Meier, and Thomas Peyrin. "Linearization Framework for Collision Attacks: Application to CubeHash and MD6". In Mitsuru Matsui, editor, Advances in Cryptology – *ASIACRYPT* 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6–10, 2009. Proceedings, volume 5912 of *Lecture Notes in Computer Science*, pages 560–577. Springer, 2009.

5. Shahram Khazaei, Simon Knellwolf, Willi Meier, and Deian Stefan. "Improved Linear Differential Attacks on CubeHash". In Daniel J. Bernstein and Tanja Lange , editor, Progress in Cryptology – *AFRICACRYPT* 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3–6, 2010. Proceedings, volume 6055 of *Lecture Notes in Computer Science*, pages 407–418. Springer, 2010.

6. Shahram Khazaei and Willi Meier. "On Reconstruction of RC4 Keys from Internal States". In Jacques Calmet, Willi Geiselmann, and Jörn Müller-Quade, editors, *Mathematical Methods in Computer Science*, MMICS 2008, Karlsruhe, Germany, December 17–19, 2008 – Essays in Memory of Thomas Beth, volume 5393 of *Lecture Notes in Computer Science*, pages 179–189. Springer, 2008.

7. Shahram Khazaei, Simon Fischer, and Willi Meier. "Reduced Complexity Attacks on the Alternating Step Generator". In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16–17, 2007, Revised Selected Papers, volume 4876 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2007.

This thesis is based on the first five articles and is organized as follows. Chapter 2 concerns cryptanalysis of Salsa20 and ChaCha stream ciphers and it includes part of the results from the first paper. Chapter 3 deals with cryptanalysis of Trivium and Grain-128 stream ciphers, which is based on the material presented in the second article. Chapter 4 includes cryptanalysis of the Klimov-Shamir self-synchronizing stream cipher which is presented in the third publication. Chapter 5 discusses cryptanalysis of CubeHash and MD6 hash function, based on an extended version of the fourth publication and the fifth. Finally, chapter 6 concludes the dissertation.

The last two articles concern cryptanalysis of two stream ciphers RC4 and Alternating Step Generator. RC4 [3, 216] is the most widely used stream cipher designed by Rivest in 1994. Alternating Step Generator [124] is one of the well-known classical LFSR-based constructions, proposed by Günther in 1987. We do not include these two papers in the thesis since they do not fit in the probabilistic neutral bit cryptanalysis.

2

Cryptanalysis of Salsa20 and ChaCha

In this chapter we cryptanalyze the stream cipher Salsa20 [31], one of the eSTREAM final candidates in the software profile [20], and its variant ChaCha [33]. These primitives have unique designs with a trivial state update function and a complex, round-based output function, similar to that of block ciphers and hash functions. The only operations used in the designs are modular addition, XOR and rotation of 32-bit words. Hence, they are suitable targets for differential cryptanalysis. First, we identify suitable choices of truncated single-bit differentials using common statistical methods. Then we introduce the notion of probabilistic neutral bits, along with a method to find them, which lets us correctly detect part of the key using probabilistic backward computations. The results of this chapter are based on [15] which we published at FSE 2008. In [15], we also presented a cryptanalysis of Rumba [32], a compression function based on Salsa20.

2.1 Introduction

Salsa20 [31] is a stream cipher, introduced by Bernstein in 2005 as a candidate to the eSTREAM project [106], and it was selected as one of the four final candidates in the software profile. Bernstein later introduced ChaCha [33], a variant of Salsa20 that aims at bringing faster diffusion without slowing down encryption. These designs have a total number of 20 rounds. The reduced variants are denoted by Salsa20/R and

ChaCha$R$, when the number of rounds is reduced to R. These ciphers are meant to provide 256-bit security; they also accept 128-bit keys, though.

Before our results, three independent cryptanalyses were published [87, 114, 227], reporting key-recovery cryptanalyses for reduced versions of Salsa20 with up to 7 rounds. These cryptanalyses exploit a truncated differential over 3 or 4 rounds. In 2005, Crowley [87] reported a 3-round differential, and built upon this a cryptanalytic algorithm for Salsa20/5 with time complexity 2^{165}. In 2006, Fischer *et al.* [114] exploited a 4-round differential to cryptanalyze Salsa20/6 with time complexity 2^{177}. In 2007, Tsunoo *et al.* [227] improved the time complexity of the cryptanalysis results for Salsa20/7 to 2^{190}, still exploiting a 4-round differential, and also claiming a break of Salsa20/8. However, the latter cryptanalysis is effectively slower than brute force.

The previous methods use the fact that not a complete knowledge of all key bits is required to detect the bias of the truncated differential from the backward direction. In other words, part of the key is *neutral* and does not need to be guessed. Tsunoo *et al.* notably tried to improve the previous results by reducing the guesses to more relevant bits — rather than guessing the whole relevant key bits — using nonlinear approximation of integer addition. However, their method is not precise enough to give an exact estimation of the cryptanalytic cost. To improve the previous cryptanalyses results of Salsa20, we introduce a novel method, inspired from correlation cryptanalysis [221], and from the notions of *completeness* [110, 143] and *neutrality* [44]. In particular, we introduce the notion of *probabilistic neutral bits* (PNBs) in this chapter. This concept not only allows us to determine the most relevant key bits to be guessed, but also provides a method to give a precise estimation of the cryptanalytic cost. To the best of our knowledge, this is the first time that PNBs are used for the cryptanalysis of keyed primitives. As a result, we present the first key-recovery cryptanalysis for Salsa20/8, with time complexity 2^{251}, and improve the previous cryptanalysis for Salsa20/7 by a factor of 2^{39}, when these ciphers use 256-bit keys. We also consider cryptanalysis of reduced-round variants of Salsa20 with 128-bit keys as well as the ChaCha stream cipher.

2.2 Specification of the primitives

In this section, we give a concise description of the stream ciphers Salsa20 and ChaCha. The R-round variants of these ciphers are denoted by Salsa20/R and ChaCha$R$. The original designs are composed of 20 rounds. These ciphers use a 256-bit key and a 64-bit nonce (or IV) to produce a sequence of 512-bit keystream blocks. They also accept 128-bit keys, in which case an extended key is used by simply concatenating two 128-bit keys to increase the length to 256 bits. Unless mentioned otherwise, we focus on the 256-bit version. These stream ciphers work with 32-bit words and use three operations on words: XOR ($\oplus$), left rotation ($\lll$) and modulo 2^{32} addition ($+$). The eight-word key $k = (k_0, k_1, \ldots, k_7)$ and the two-word nonce $v = (v_0, v_1)$ produce a sequence of 16-word keystream blocks. The ith keystream block, $0 \leq i \leq 2^{64} - 1$, of the stream cipher is a function of the key, the nonce, and the two-word counter $t = (t_0, t_1)$ which corresponds to the integer i ($i = t_0 + t_1 2^{32}$). Both of these ciphers operate on the 4×4 matrix X of words of the following form

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} . \tag{2.1}$$

Once the matrix X is filled with the key, the nonce, the counter and some constants, the ith keystream block Z is defined as

$$Z = X + \mathsf{Round}_R(X) . \tag{2.2}$$

The "+" denotes wordwise modular addition of two matrices. The major differences between Salsa20 and ChaCha are initialization of the matrix X and the definition of Round_R function. We detail the differences below.

2.2.1 Salsa20

For Salsa20 the matrix X is initialized as

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} c_0 & k_0 & k_1 & k_2 \\ k_3 & c_1 & v_0 & v_1 \\ t_0 & t_1 & c_2 & k_4 \\ k_5 & k_6 & k_7 & c_3 \end{pmatrix} . \tag{2.3}$$

where $(c_0, c_1, c_2, c_3) = (\texttt{0x61707865}, \texttt{0x3320646E}, \texttt{0x79622D32}, \texttt{0x6B206574})$ are predefined constants.[1] The function $\mathsf{Round}_R(X)$ is itself based on a permutation called quarterround which maps a four-word vector (a_0, a_1, a_2, a_3) into the four-word vector (b_0, b_1, b_2, b_3) according to the following sequence of operations

$$\begin{array}{lll} b_1 & = a_1 & \oplus\big[(a_3 + a_0) \lll 7\big] \ , \\ b_2 & = a_2 & \oplus\big[(a_0 + b_1) \lll 9\big] \ , \\ b_3 & = a_3 & \oplus\big[(b_1 + b_2) \lll 13\big] \ , \\ b_0 & = a_0 & \oplus\big[(b_2 + b_3) \lll 18\big] \ . \end{array} \tag{2.4}$$

For a matrix X, the value of $\mathsf{Round}_r(X)$, $1 \leq r \leq R$ is computed in r rounds. The rounds are counted from one, and at each round the matrix X is updated. In odd rounds, the quarterround mapping updates the columns (x_0, x_4, x_8, x_{12}), (x_5, x_9, x_{13}, x_1), $(x_{10}, x_{14}, x_2, x_6)$ and $(x_{15}, x_3, x_7, x_{11})$. Whereas, in even rounds, the rows (x_0, x_1, x_2, x_3), (x_5, x_6, x_7, x_4), $(x_{10}, x_{11}, x_8, x_9)$ and $(x_{15}, x_{12}, x_{13}, x_{14})$ are updated by the quarterround function.

2.2.2 ChaCha

ChaCha is similar to Salsa20, with the following modifications.

1. The input words are placed differently in the initial matrix:

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ t_0 & t_1 & v_0 & v_1 \end{pmatrix} \ . \tag{2.5}$$

2. The quarterround permutation transforms a vector (a_0, a_1, a_2, a_3) to (b_0, b_1, b_2, b_3) by sequentially computing

$$\begin{array}{llllllll} u_0 & = a_0 & + & a_1, & u_3 & = (a_3 & \oplus & u_0) \lll 16 \ , \\ u_2 & = a_2 & + & u_3, & u_1 & = (a_1 & \oplus & u_2) \lll 12 \ , \\ b_0 & = u_0 & + & u_1, & b_3 & = (u_3 & \oplus & z_0) \lll 8 \ , \\ b_2 & = u_2 & + & b_3, & z_1 & = (u_1 & \oplus & z_2) \lll 7 \ . \end{array} \tag{2.6}$$

3. The $\mathsf{Round}_r(X)$ function, $1 \leq r \leq R$, is defined differently. In odd rounds, the quarterround function updates the columns (x_0, x_4, x_8, x_{12}), (x_1, x_5, x_9, x_{13}),

[1] A word is denoted by an eight-digit hexadecimal number and, as it is conventional, we prepend the hexadecimal numbers with 0x.

$(x_2, x_6, x_{10}, x_{14})$ and $(x_3, x_7, x_{11}, x_{15})$. Whereas, in even rounds, the quarterround mapping updates the diagonals $(x_0, x_5, x_{10}, x_{15})$, $(x_1, x_6, x_{11}, x_{12})$, (x_2, x_7, x_8, x_{13}) and (x_3, x_4, x_9, x_{14}).

We refer to the Salsa20 [31] and ChaCha [33] original descriptions for more details and design philosophies. Note that we did not use the notation $\mathsf{Round}^r(X)$ as the input matrix is updated differently in the odd and even rounds. This would have brought confusion with r-fold function combination. The $\mathsf{Round}_r(X)$, $1 \leq r \leq R$, function is a permutation and trivially invertible for both Salsa20 and ChaCha. We use the notation $\mathsf{Round}_{R,r}^{-1}$, $1 \leq r \leq R$, for the function which maps $\mathsf{Round}_R(X)$ into $\mathsf{Round}_r(X)$.

2.3 Cryptanalytic model

The most reasonable cryptanalytic model for Salsa20 and ChaCha is a *chosen nonce, chosen counter scenario*. In this model an adversary wants to recover the key k of one of the stream ciphers Salsa20/R or ChaCha$R$. As shown in Figure 2.1, the adversary is allowed to interact with the stream cipher oracle as follows. The adversary can submit as many pairs of nonces and counters (v, t) of his choice as he wants. Then the oracle responds to each request with the corresponding keystream block Z created from a fixed random key k, unknown to the adversary. Once the adversary has collected enough keystream blocks, he tries to guess the secret key k.

We consider differential cryptanalysis of Salsa20 and ChaCha. Specifically, the adversary sends two pairs of nonces and counters (v, t) and (v', t') with a fixed difference $(\Delta v, \Delta t)$ to the oracle. The oracle then sends back their corresponding keystream blocks Z and Z'. In this chapter, the data complexity is given in number of keystream block pairs collected by the adversary. The time complexity unit is the time required to produce one keystream block.

2.4 Differential cryptanalysis with probabilistic neutral bits

This section introduces differential cryptanalysis based on a new technique called *probabilistic neutral bits* (PNBs). To apply it to Salsa20/R and ChaCha$R$, we first search for one-bit truncated differentials in the state matrix after a few rounds. Then we

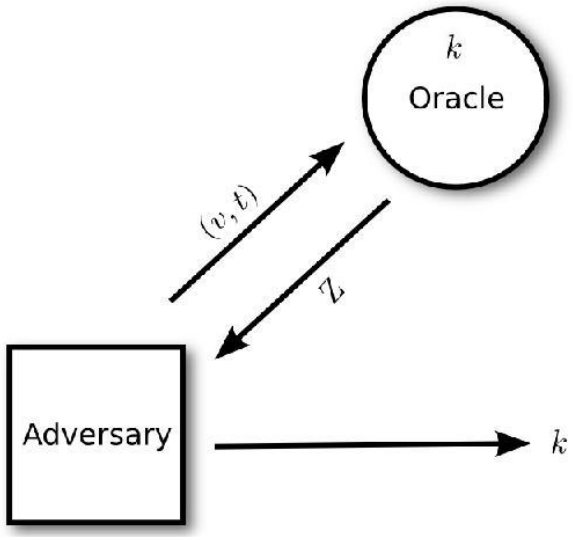

Figure 2.1: Cryptanalytic model for the Salsa20 and ChaCha - Cryptanalysis is done in a chosen nonce, chosen counter scenario. The adversary has access to many keystream blocks for selected pairs of nonces and counters of his choice. The adversary's goal is then to recover the secret key which is unknown to him.

describe a general framework for probabilistic backwards computation, and introduce the notion of PNBs along with a method to find them.

The intuition behind our method can be sketched as follows. Suppose that we have identified a suitable non-zero difference $(\Delta v, \Delta t)$ in the nonce and counter such that after r rounds, $1 \leq r \leq R$, one of the bits in the output of the function Round_r is biased. Recall that $Z = X + \mathsf{Round}_R(X)$ and X depends on the key. If we know the whole key, we can compute backwards from given keystream blocks to observe the biased truncated differential bit of $\mathsf{Round}_r(X)$. However, if we do not know the key, we might have to guess it in order to detect the bias which is quite expensive. The idea of the PNBs is to *partially* guess the key. That is, we guess only those key bits which are extremely important. The bias must still be observable when we compute backwards, even though the other key bits are not correctly guessed.

2.4.1 Truncated differentials

Let X be an initial state matrix filled with the constants, key k, nonce v and counter t, see equations (2.3) and (2.5). Consider another initial state matrix X' filled with the constants and the same key k, but with nonce v' and counter t', where $\Delta v = (v_0 \oplus v'_0, v_1 \oplus v'_1)$ and $\Delta t = (t_0 \oplus t'_0, t_1 \oplus t'_1)$. We are interested in the one-bit truncated differential properties of Round_r function for some small value of r. Let $\Delta^r = \mathsf{Round}_r(X) \oplus \mathsf{Round}_r(X')$ where "$\oplus$" is the wordwise XOR of two matrices. Denote the j-th, $0 \leq j \leq 31$, LSB of the word i, $0 \leq i \leq 15$, of a general 4×4 matrix X by $[X]_{i,j}$. We consider one-bit truncated differentials at the qth LSB of the word number p. Such a differential is denoted by $([\Delta^r]_{p,q} \mid \Delta v, \Delta t)$ where $1 \leq r \leq R$, $0 \leq p \leq 15$, $0 \leq q \leq 31$. In this context, the bias ε_d of the differential for a fixed key is defined by

$$\Pr_{v,t}\{[\Delta^r]_{p,q} = 1\} = \frac{1}{2}(1 + \varepsilon_d) \ , \tag{2.7}$$

where v and t are considered as random variables. In addition, the value $\varepsilon_d^\star$ is defined to be the median value of ε_d when key is considered random. Hence, for half of the keys this differential will have a bias of at least $\varepsilon_d^\star$.

2.4.2 Probabilistic backwards computation

In the following, assume that we are given a differential $([\Delta^r]_{p,q} \mid \Delta v, \Delta t)$ with bias ε_d for an unknown key k. The corresponding keystream blocks to X and X' are respectively denoted by Z and Z'. Recall that X depends on k, v and t, whereas X' depends on k, $v' = v \oplus \Delta v$ and $t' = t \oplus \Delta t$; see equations (2.3) and (2.5). Having Z, Z', k, v and t, one can invert the operations in $Z = X + \mathsf{Round}_R(X)$ and $Z' = X' + \mathsf{Round}_R(X')$ in order to access to the r-round forward differential (with $r \leq R$). This is possible by backward computation thanks to the relations $\mathsf{Round}_r(X) = \mathsf{Round}_{R,r}^{-1}(Z - X)$ and $\mathsf{Round}_r(X') = \mathsf{Round}_{R,r}^{-1}(Z' - X')$. More specifically, define $f(k, v, t, Z, Z')$ as the function which returns the qth LSB of the word number p of the matrix $\mathsf{Round}_{R,r}^{-1}(Z - X) \oplus \mathsf{Round}_{R,r}^{-1}(Z' - X')$, that is,

$$f(k, v, t, Z, Z') = [\mathsf{Round}_{R,r}^{-1}(Z - X) \oplus \mathsf{Round}_{R,r}^{-1}(Z' - X')]_{p,q} \ . \tag{2.8}$$

Therefore, the relation $f(k, v, t, Z, Z') = [\Delta^r]_{p,q}$ holds. Given enough output block pairs with the presumed difference in the input, one can verify the correctness of a

guessed candidate $\hat{k}$ for the key k by evaluating the bias of the function f. This is possible based on the reasonable assumption that we have $\Pr\{f(\hat{k}, v, t, Z, Z') = 1\} = \frac{1}{2}(1 + \varepsilon_d)$ conditioned on $\hat{k} = k$, whereas for (almost all) $\hat{k} \neq k$ we expect f to be unbiased, *i.e.*, $\Pr\{f(\hat{k}, v, t, Z, Z') = 1\} = \frac{1}{2}$. The classical way of finding the correct key requires exhaustive search over all possible 2^{256} guesses $\hat{k}$. However, we can search only over a subkey of $m = 256 - n$ bits, provided that an approximation g of f which effectively depends on m key bits is available. More formally, let k_s correspond to the subkey of m bits of the key k and let f be correlated to g with bias ε_a; that is,

$$\Pr_{v,t,Z,Z'}\{f(k, v, t, Z, Z') = g(k_s, v, t, Z, Z')\} = \frac{1}{2}(1 + \varepsilon_a) \ . \tag{2.9}$$

Note that deterministic backwards computation (*i.e.*, $k_s = k$ with $f = g$) is a special case with $\varepsilon_a = 1$. Denote the bias of g by ε, *i.e.*, $\Pr_{v,t}\{g(k_s, v, t, Z, Z') = 1\} = \frac{1}{2}(1+\varepsilon)$. Under some reasonable independency assumptions, the equality $\varepsilon = \varepsilon_d \cdot \varepsilon_a$ holds. Again, we denote $\varepsilon^\star$ the median bias over all keys (we verified in experiments that $\varepsilon^\star$ can be well estimated by the median of $\varepsilon_d \cdot \varepsilon_a$). Here, one can verify the correctness of a guessed candidate $\hat{k}_s$ for the subkey k_s by evaluating the bias of the function g based on the fact that we have $\Pr\{g(\hat{k}_s, v, t, Z, Z') = 1\} = \frac{1}{2}(1 + \varepsilon)$ for $\hat{k}_s = k_s$, whereas $\Pr\{g(\hat{k}_s, v, t, Z, Z') = 1\} = \frac{1}{2}$ for $\hat{k}_s \neq k_s$. This way we are faced with an exhaustive search over 2^m subkey candidates as opposed to the original 2^{256} key candidates. This can potentially lead to a faster cryptanalysis. We stress that the price which we pay is a higher data complexity as we will see in the following.

In section 2.4.3, we address the problem of finding an approximation function g. For the moment, assume that such an approximation is available. One can shrink the set of possible 2^m subkeys to a smaller subset if he is given some keystream block pairs. The number of required keystream block pairs depends on how small one wants the size of the filtered subset be and the probability that this subset includes the correct subkey. More precisely, for estimating the number N of keystream block pairs, we need to consider the following problem of hypothesis testing [222, 85]. We are given a set of 2^m sequences of random variables where $2^m - 1$ of them verify the null hypothesis H_0; that is, the candidate is not the correct subkey. One of them, *i.e.*, the correct subkey, verifies an alternative hypothesis H_1. For a realization a of the corresponding random variable $\mathcal{A}$, the decision rule $\mathcal{D}(a) = i$ to accept H_i can lead to two types of errors:

1. Non-detection: $\mathcal{D}(a) = 0$ when $\mathcal{A} \in H_1$. The probability of this event is p_{nd}.

2. False alarm: $\mathcal{D}(a) = 1$ when $\mathcal{A} \in H_0$. The probability of this event is p_{fa}.

In our case, the random variable $\mathcal{A}$ is the vector of length N of the values of $g(\hat{k}_s, v, t, Z, Z')$ derived from the N keystream block pairs. Moreover, the optimal distinguisher is simply constructed by comparing the Hamming weight of a realization a of this vector with a certain threshold. The Neyman-Pearson lemma [85] gives us a result to estimate the number N of samples required to get some bounds on the probabilities. Indeed, it can be shown that for

$$N \approx \left(\frac{\sqrt{c \ln 4} + 3\sqrt{1 - \varepsilon^2}}{\varepsilon} \right)^2 , \tag{2.10}$$

and for an appropriate threshold value we have $p_{\text{nd}} = 1.3 \times 10^{-3}$ and $p_{\text{fa}} = 2^{-c}$. Calculus details and more details on the construction of the optimal distinguisher can be found in [222], see also [85, 22] for more general results on the distinguishability of distributions. In our case, the value of ε is key dependent, so we use the median bias $\varepsilon^\star$ in place of ε in equation (2.10), resulting in a success probability of at least $\frac{1}{2}(1 - p_{\text{nd}}) \approx \frac{1}{2}$ for the cryptanalytic algorithm.

We will show in section 2.5 that the time complexity of a cryptanalytic algorithm constructed based on such a distinguisher for Salsa20 and ChaCha is $2^m N + 2^{256-c}$. For a given m and $\varepsilon^\star$, the value of c (or equivalently N or p_{fa}) is chosen such that it minimizes $2^m N + 2^{256-c}$.

Remark 1. *Previous cryptanalysis of Salsa20 use the rough estimate of $N = \varepsilon^{-2}$ samples in order to identify the correct subkey. This estimate, however, is low and incorrect as it ignores the numerator of equation (2.10). This value is the number of samples necessary to distinguish* two *random variables, one of them coming from a uniform source (hypothesis H_0) and the other from a non-uniform source with bias ε (hypothesis H_1). This is clearly a different problem of hypothesis testing.*

2.4.3 Probabilistic neutral bits

Our new view of the problem, described in section 2.4.2, demands efficient ways for finding suitable approximations $g(k_s, W)$ of a given function $f(k, W)$ where k_s is a subvector of k. In our case, we have $W = (v, t, Z, Z')$ which we treat as a uniformly

distributed random variable. Finding such approximations, in general, is an interesting open problem which is related to the *completeness/neutrality* [110, 143] (see also [44]), *correlation immunity* [220] and *avalanche* properties [116, 240] of Boolean functions. In this section we introduce a generalized concept of neutral bits, called *probabilistic neutral bits* (PNBs). This concept helps us to find suitable approximations in the case that the Boolean function f does not properly mix its input bits. Generally speaking, the concept of probabilistic neutrality allows us to divide the key bits into two groups: *significant key bits* (of size m) and *non-significant key bits* (of size $n = 256 - m$). In order to identify these two sets, we focus on the amount of influence which each bit of the key has on the output of f. Definition 3 formally introduces the *neutrality measure*.

Algorithm 1 Estimation of the neutrality measure

Inputs: key bit index l, number of rounds R, the parameters $\Delta v, \Delta t, r, p$ and q of the truncated differential.

Output: the estimated neutrality measure γ_l.

1: Choose the number of samples T and let $\texttt{ctr} = 0$.
2: **for** T' from 1 to T **do**
3: Choose k, v, t, Z and Z' at random.
4: Construct an initial state matrix X with key k, nonce v and counter t (see equations (2.3) and (2.5))
5: Similarly construct X' with key k, nonce $v \oplus \Delta v$ and counter $t \oplus \Delta t$
6: Compute the truncated output difference bit $d_1 = [\mathsf{Round}^{-1}_{R,r}(Z - X) \oplus \mathsf{Round}^{-1}_{R,r}(Z' - X')]_{p,q}$.
7: Flip the lth key bit in X and X'.
8: Compute the truncated output difference bit $d_2 = [\mathsf{Round}^{-1}_{R,r}(Z - X) \oplus \mathsf{Round}^{-1}_{R,r}(Z' - X')]_{p,q}$.
9: Increment $\texttt{ctr}$ if the output differences are equal, *i.e.*, $d_1 = d_2$.
10: **end for**
11: Output $\gamma_l = 2 \cdot \texttt{ctr}/T - 1$.

Definition 3. *The neutrality measure of the lth key bit with respect to the function $f(k, W)$ is defined as γ_l, where $\frac{1}{2}(1 + \gamma_l)$ is the probability (over all k and W) that complementing the lth key bit does not change the output of $f(k, W)$.*

A key bit which has a neutrality measure equal to 1 is called a neutral bit; that is, it has no effect on the output of the function. More generally, for a given threshold value

γ, $-1 \leq \gamma \leq 1$, we call the key bits with $\gamma_l > \gamma$ PNBs; whereas we refer to the other key bits as significant key bits. We also use the term non-significant key bits for PNBs. Intuitively, the bigger γ is, the less effect the PNBs have on the output of the function, and hence the better g approximates f. However, the cryptanalyst must choose an optimal value for the threshold γ to minimize the cryptanalytic time complexity. We discuss this issue in section 2.5.

Remark 2. *Tsunoo et al. [227] used approximations of integer addition to identify the dependency of key bits. In their method, each bit of a modular addition is approximated by a nonlinear Boolean function with a limited algebraic degree. This can be seen as a special case of our method. In particular, our method lets us give a precise estimation of the cryptanalytic time and data complexities (see section 2.5), whereas the estimation of [227] is less reliable.*

We use Algorithm 1 to compute the neutrality measure of a single key bit for Salsa20 or ChaCha. The number of samples T must be adjusted according to the desired confidence level on the estimated neutrality measure. For cryptanalysis of these stream ciphers, a threshold γ is chosen and then the key bits are divided into the set of significant and non-significant key bits according to this threshold value and their neutrality measure. Then the function g is defined to be the same as f except that the non-significant key bits are set to a random value in k.

2.5 Cryptanalytic algorithm and time complexity

Let's assume that we have found a truncated differential $([\Delta^r]_{p,q} \mid \Delta v, \Delta t)$, and for a fixed threshold γ, suppose that we have identified the set of significant key bits for which the function g has median bias $\varepsilon^\star$, see section 2.4.2. The cryptanalytic algorithm is then presented in Algorithm 2. Let us now discuss the time complexity of our cryptanalytic algorithm. Step 2 is repeated for all 2^m subkey candidates. For each subkey, steps (3) and (4) are always executed which requires roughly N keystream block evaluations.[1] The search part of step (6) is performed only with probability $p_{\text{fa}} = 2^{-c}$ which brings an

[1] More precisely the complexity is about $2N(R-r)/R$ times the required time for producing one keystream block. The reason is that R rounds are executed to produce one key stream block, whereas here we only need to invert $R-r$ rounds for each keystream block pair. Recall that in order to evaluate f (see equation (2.8)), two partial inversions are required.

additional cost of 2^n in case a subkey passes the optimal distinguisher filter. Therefore, the complexity of steps (5-8) is $2^n p_{\text{fa}}$, resulting in a total complexity of

$$2^m(N + 2^n p_{\text{fa}}) = 2^m N + 2^{256-c} \ , \tag{2.11}$$

for the cryptanalytic algorithm.

Algorithm 2 : Cryptanalytic algorithm for Salsa20 and ChaCha

Inputs: the parameters $p, q, r, \Delta v$ and Δt of the truncated differential, the set of m significant key bits corresponding to a threshold value γ, number N of keystream block pairs.

Output: the secret key with some probability.

1: Collect N pairs of keystream blocks, produced under an unknown key, where each pair satisfies the relevant input difference Δv for nonce and Δt for counter.
2: **for** each of the 2^m possible assignments k_s to the subkey (*i.e.*, the significant key bits) **do**
3: Set the significant key bits of a key k to k_s and the non-significant key bits to a random value.
4: Evaluate the function g with the subkey k_s for the N keystream block pairs (to this end, evaluate the function f with the key k constructed in step 3 for the N keystream block pairs).
5: **if** the optimal distinguisher, based on the N evaluations of g, legitimates the subkey candidate k_s as a potentially correct one **then**
6: Perform an additional exhaustive search over the $n = 256 - m$ non-significant key bits in order to check the correctness of the filtered subkey k_s, and to find the non-significant key bits in case k_s is indeed the correct subkey.
7: Stop if the right key is found, and output the recovered key.
8: **end if**
9: **end for**

However, there are several issues to be discussed. First, for a given differential $([\Delta^r]_{p,q} \mid \Delta v, \Delta t)$ and a threshold γ, one needs to estimate the median bias $\varepsilon^\star$ of the function g in order to determine the required number N of keystream block pairs. If the bias is big enough (let say $|\varepsilon^\star| > 2^{-15}$), one can experimentally estimate it. Otherwise, some theoretical tools are required. We failed to estimate the bias theoretically. To cope with this problem, in our simulation results to be presented in section 2.6, we not only restricted ourselves to small values of R (up to 8) and r (up to 4), but also

we allowed only one single bit of one of the four words of $(\Delta v, \Delta t)$ to be nonzero. Second, once we have the estimated value $\varepsilon^\star$ for a given threshold γ, we need to choose the false alarm probability $p_{\text{fa}} = 2^{-c}$ such that the time complexity is minimized; see equation (2.10) (with $\varepsilon^\star$ instead of ε) and equation (2.11). Third, the minimized time complexity has still the threshold value γ as a degree of freedom. Bigger values of γ increase the number m of the significant key bits as well as the bias $\varepsilon^\star$. Since these two values have opposite effect on the time complexity, the cryptanalyst must still find the threshold value γ which optimizes the cryptanalytic time complexity.

2.6 Experimental results

As already mentioned, we only consider one bit difference in Δv or Δt. This gives us 128 possible input differentials. In this section, we denote an r-round differential $([\Delta^r]_{p,q} \mid \Delta v, \Delta t)$ by $([\Delta^r]_{p,q} \mid [\Delta^0]_{i,j})$ where i, j indicate the position of the difference in the initial state matrix. We have $0 \leq q, j \leq 31$ and $0 \leq p \leq 15$; moreover, for Salsa20 $6 \leq i \leq 9$ whereas for ChaCha $12 \leq i \leq 15$, see equations (2.3) and (2.5). We used automatized search to identify optimal differentials for the reduced-round versions Salsa20/7, Salsa20/8, ChaCha6, and ChaCha7. This search is based on the following observation. The number n of PNBs for some fixed threshold γ mostly depends on the output difference position (*i.e.*, p and q), but not on the input difference position (*i.e.*, i and j). Consequently, for each of the 512 possible positions for the truncated bit, we can assign the input difference with maximum bias ε_d, and estimate time complexity of the algorithm. Below we only present the differentials leading to the best cryptanalyses. The threshold γ is also an important parameter. Given a fixed differential, time complexity of the algorithm is minimal for some optimal value of γ. However, this optimum may be reached for quite small γ, such that n is large and $|\varepsilon_a^\star|$ small. We use at most 2^{24} random nonces and counters for each of the 2^{10} random keys, so we can only measure a bias of about $|\varepsilon_a^\star| > \beta \cdot 2^{-12}$ (where $\beta \approx 10$ for a reasonable estimation error). In our experiments, the optimum is not reached with these computational possibilities in some cases (see, *e.g.*, Table 2.2), and we note that the described complexities may be improved by choosing a smaller γ. Table 2.1 summarizes our cryptanalytic results on the two target ciphers.

Cipher	Key length	Time	Data
Salsa20/8	256	2^{251}	2^{31}
ChaCha7	256	2^{248}	2^{27}
Salsa20/7	256	2^{151}	2^{26}
ChaCha6	256	2^{139}	2^{30}
Salsa20/7	128	2^{111}	2^{21}
ChaCha6	128	2^{107}	2^{30}

Table 2.1: Summary of our cryptanalytic results on Salsa and ChaCha. The table includes the best cryptanalytic results found for reduced-round variants of Salsa and ChaCha.

Cryptanalysis of the 256-bit Salsa20/7. We use the differential $([\Delta^4]_{1,14} \mid [\Delta^0]_{7,31})$ with $|\varepsilon_d^\star| = 0.131$. The output difference is observed after working three rounds backward from a 7-round keystream block. To illustrate the role of the threshold γ, we present in Table 2.2 complexity estimates along with the number n of PNBs, the values of $|\varepsilon_d^\star|$ and $|\varepsilon^\star|$, and the optimal values of c for several threshold values. For $\gamma = 0.5$, the algorithm runs in time 2^{151} and data 2^{26}. The previous best cryptanalysis in [227] required about 2^{190} trials and 2^{12} data.

γ	n	$\lvert\varepsilon_a^\star\rvert$	$\lvert\varepsilon^\star\rvert$	c	Time	Data
1.00	39	1.000	0.1310	31	2^{230}	2^{13}
0.90	97	0.655	0.0860	88	2^{174}	2^{15}
0.80	103	0.482	0.0634	93	2^{169}	2^{16}
0.70	113	0.202	0.0265	101	2^{162}	2^{19}
0.60	124	0.049	0.0064	108	2^{155}	2^{23}
0.50	131	0.017	0.0022	112	2^{151}	2^{26}

Table 2.2: Different cryptanalytic trade-offs for Salsa20/7. The table includes different parameters which we found for our cryptanalysis of the 256-bit Salsa20/7 with the differential $([\Delta^4]_{1,14} \mid [\Delta^0]_{7,31})$. We cannot find the optimal value for the threshold γ since we measure the bias $\varepsilon^\star$ experimentally.

Cryptanalysis of the 256-bit Salsa20/8. We use again the previous differential $([\Delta^4]_{1,14} \mid [\Delta^0]_{7,31})$ with $|\varepsilon_d^\star| = 0.131$. The output difference is observed after working four rounds backward from an 8-round keystream block. For the threshold $\gamma = 0.12$,

we find $n = 36$, $|\varepsilon_a^\star| = 0.0011$, and $|\varepsilon^\star| = 0.00015$. For $c = 8$, this results in time 2^{251} and data 2^{31}. The list of PNBs is {26-31, 71, 72, 120-122, 148, 165-177, 210-212, 224, 225, 242-247}. Note that our cryptanalytic algorithm is $2^{255-251} = 2^4$ times faster than brute force for the same success probability of about fifty percent. The previous cryptanalysis in [227] claims 2^{255} trials with data 2^{10} for success probability 44% which is worse than exhaustive search, let alone that the required number of samples is underestimated which increase their complexity even further, see Remark 1. Therefore, their algorithm does not constitute a break of Salsa20/8.

Cryptanalysis of the 128-bit Salsa20/7. Our cryptanalytic algorithm can be adapted to the 128-bit version of Salsa20/7. With the differential $([\Delta^4]_{1,14} \mid [\Delta^0]_{7,31})$ and $\gamma = 0.4$, we find $n = 38$, $|\varepsilon_a^\star| = 0.045$, and $|\varepsilon^\star| = 0.0059$. For $c = 21$, this breaks Salsa20/7 within 2^{111} time and 2^{21} data. Our algorithm fails to break 128-bit Salsa20/8 because of the insufficient number of PNBs.

Cryptanalysis of the 256-bit ChaCha6. We use the differential $([\Delta^3]_{11,0} \mid [\Delta^0]_{13,13})$ with $|\varepsilon_d^\star| = 0.026$. The output difference is observed after working three rounds backward from a 6-round keystream block. For the threshold $\gamma = 0.6$, we find $n = 147$, $|\varepsilon_a^\star| = 0.018$, and $|\varepsilon^\star| = 0.00048$. For $c = 123$, this results in time 2^{139} and data 2^{30}.

Cryptanalysis of the 256-bit ChaCha7. We use again the previous differential $([\Delta^3]_{11,0} \mid [\Delta^0]_{13,13})$ with $|\varepsilon_d^\star| = 0.026$. The output difference is observed after working four rounds backward from a 7-round keystream block. For the threshold $\gamma = 0.5$, we find $n = 35$, $|\varepsilon_a^\star| = 0.023$, and $|\varepsilon^\star| = 0.00059$. For $c = 11$, this results in time 2^{248} and data 2^{27}. The list of PNBs is {3, 6, 15, 16, 31, 35, 67, 68, 71, 91-100, 103, 104, 127, 136, 191, 223-225, 248-255}.

Cryptanalysis of the 128-bit ChaCha6. Our cryptanalytic algorithm can be adapted to the 128-bit version of ChaCha6. With the differential $([\Delta^3]_{11,0} \mid [\Delta^0]_{13,13})$ and $\gamma = 0.5$, we find $n = 51$, $|\varepsilon_a^\star| = 0.013$, and $|\varepsilon^\star| = 0.00036$. For $c = 26$, this breaks ChaCha6 within 2^{107} time and 2^{30} data. Our algorithm fails to break 128-bit ChaCha7.

2.7 Summary

We presented a novel method for cryptanalyzing reduced-round Salsa20 and ChaCha, inspired by correlation cryptanalysis and by the notion of neutral bits. In particular, we introduced the notion of probabilistic neutral bits to mount the first cryptanalytic algorithm faster than exhaustive search on the stream cipher Salsa20/8 with a 256-bit key. As of April 2010, this is still the best result on the Salsa20 family of stream ciphers. In [193], Sylvain Pelissier applied our method to reduced-word variants of Salsa20 with words of size 8 and 16 bits. His results show that the probabilistic neutral bits can indeed be successfully applied in practice. Nothing in this chapter affects the security of the full version of Salsa20 and ChaCha. Notably, Salsa20 is widely viewed as a very promising stream cipher for software applications.

3

Chosen IV Cryptanalysis for Synchronous Stream Ciphers

In this chapter we cryptanalyze the initialization procedure of two synchronous stream ciphers Trivium [68] and Grain-128 [126]. Trivium is one of the three stream ciphers in the (updated) eSTREAM final portfolio [20] for hardware applications along with Grain [128, 127] and MICKEY [21]; whereas Grain-128, a variant of Grain, was only among the eSTREAM Phase 3 candidates. Both of these ciphers are constructed using finite state machines with very simple state update functions and output filters, making them highly suitable for restricted environments. The initialization procedures of these ciphers are rather simple and based on iterating a fixed number of rounds a function, either the same as or similar to their state update functions. Our cryptanalysis concerns the general problem of finding an unknown key K from the output of an easily computable keyed function $F(V, K)$, where the adversary has the power to choose the public variable V. In the case of synchronous stream ciphers, we simply define F as the function which maps the key K and IV V into the first keystream bit generated from that key and IV. The goal of the adversary is then to recover the key in a chosen IV scenario. We first derive a function from F, based on a coefficient of its algebraic normal form, which is less complex than F itself. Based on the idea of *probabilistic neutral bits*, introduced in chapter 2, we then examine how much influence each key bit does have on the value of that coefficient. Our cryptanalytic method exploits such information to find the key faster than exhaustive search by filtering some of the wrong candidates. The results of this chapter are based on [113], published at AFRICACRYPT 2008.

3.1 Introduction

Synchronous stream ciphers are symmetric cryptosystems which are suitable in software applications with high throughput requirements, or in hardware applications with restricted resources (such as limited storage, gate count, or power consumption). For synchronization purposes, in many protocols the message is divided into short frames where each frame is encrypted using a different publicly known initial value (IV) and the same secret key. Stream ciphers should be designed to resist cryptanalytic algorithms that exploit many known keystreams generated by the same key but different chosen IVs. In general, the key and the IV are mapped to the initial state of the stream cipher by an initialization function (and the automaton produces then the keystream bits, using an output and update function). The security of the initialization function relies on its mixing (or *diffusion*) properties: each key and IV bit should affect each initial state bit in a complex way. This can be achieved with a round-based approach, where each round consists of some nonlinear operations. On the other hand, using a large number of rounds or highly involved operations is inefficient for applications with frequent resynchronizations. Limited resources of hardware oriented stream ciphers may even preclude the latter, and good mixing should be achieved with simple Boolean functions and a well-chosen number of rounds. In [109, 191, 214, 111], a framework for chosen IV statistical analysis of stream ciphers is suggested to investigate the structure of the initialization function. If mixing is not perfect, then the initialization function has an algebraic normal form (ANF) which can be distinguished from a uniformly random Boolean function. Particularly, the coefficients of high degree monomials in the IV (*i.e.*, the product of many IV bits) may have some biased distribution. This specially happens in the reduced-round variants of the cipher due to the fact that it takes many operations before all of the IV bits meet in the same memory cell. In [109], this question was raised: "*It is an open question how to utilize these weaknesses of state bits to attack the cipher.*". The aim of this chapter is to contribute to this problem and present a framework to mount key-recovery cryptanalytic algorithms. As in [109, 191], one selects a subset of IV bits as variables. Assuming all other IV values as well as the key fixed, one can write a keystream symbol as a Boolean function. By running through all possible values of these bits and generating a keystream output each time,

one can compute the truth table of this Boolean function. Each coefficient in the algebraic normal form of this Boolean function is parametrized by the bits of the secret key. Based on the idea of *probabilistic neutral bits* from [15], we now examine if every key bit in the parametrized expression of a coefficient does occur, or more generally, how much influence each key bit does have on the value of the coefficient. If a coefficient does not depend on all the key bits, this fact can be exploited to filter those keys which do not satisfy the imposed value for the coefficient. It is shown in [231] that for Trivium [68], one of the eSTREAM final candidates in the hardware profile [20], with IV initialization reduced to 576 iterations instead of 1192, linear relations on the key bits can be derived for well chosen sets of variable IV bits. Our framework is more general, as it works with the concept of (probabilistic) neutral key bits, *i.e.*, key bits which have no influence on the value of a coefficient with some (high) probability. This way, we can get information on the key for many more iterations in the IV initialization of Trivium, and similarly for the eSTREAM Phase 3 candidate Grain-128 [126]. On the other hand, extensive experimental evidence indicates clear limits to our approach; with our methods, it is unlikely to get information on the key faster than exhaustive key search for Trivium or Grain-128 with full IV initialization.

3.2 Notations

We use $\mathbb{B} = \{0, 1\}$ for the binary field with two elements. A general t-bit vector in $\mathbb{B}^t$ is denoted by $X = (x_1, x_2, \ldots, x_t)$. By making a partition of X into $U \in \mathbb{B}^l$ and $W \in \mathbb{B}^{t-l}$, we mean dividing the variables set $\{x_1, x_2, \ldots, x_t\}$ into two disjoint subsets $\{u_1, \ldots, u_l\}$ and $\{w_1, \ldots, w_{t-l}\}$ and setting $U = (u_1, \ldots, u_l)$ and $W = (w_1, \ldots, w_{t-l})$. However, whenever we write $(U; W)$ we mean the original vector X. For example, $U = (x_2, x_4)$ and $W = (x_1, x_3, x_5)$ is a partition for the vector $X = (x_1, x_2, x_3, x_4, x_5)$ and $(U; W)$ is equal to X and not to $(x_2, x_4, x_1, x_3, x_5)$. We also use the notation $U = X \setminus W$ and $W = X \setminus U$. A vector of size zero is denoted by $\emptyset$. For a multi-index $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_t) \in \mathbb{B}^t$ and a vector $X \in \mathbb{B}^t$, we have the convenient shorthand $X^\alpha = \prod_{i=1}^t x_i^{\alpha_i}$ for monomial expressions. For example, $(x_1, x_2, x_3, x_4, x_5)^{(1,0,1,1,0)} = x_1 x_3 x_4$. The Hamming weight of a binary vector X (*i.e.*, the number of nonzero elements of the vector) is denoted by $\mathrm{wt}(X)$. For two binary vectors α and β of equal length t, whenever we write $\alpha \leq \beta$ we mean that $\alpha_i \leq \beta_i$, for $1 \leq i \leq t$.

3.3 Problem formalization

Let $F : \mathbb{B}^m \times \mathbb{B}^n \to \mathbb{B}$ be a keyed Boolean function which maps the m-bit public variable V and the n-bit secret variable K into the output bit $F(V, K)$. The function F could stand, *e.g.*, for the Boolean function which maps the key K and IV V of a synchronous stream cipher to the (let say) first keystream bit produced from that key and IV. We consider the following cryptanalytic model which corresponds to the chosen IV scenario in case of synchronous stream ciphers. An oracle chooses a key K uniformly at random over $\mathbb{B}^n$ and returns $F(V, K) \in \mathbb{B}$ to an adversary for any chosen $V \in \mathbb{B}^m$ of adversary's choice. The oracle chooses the key K only once and keeps it fixed and unknown to the adversary. The goal of the adversary is to recover K by dealing with the oracle assuming that he has also the power to evaluate F for all inputs, *i.e.*, all secret and public variables. To this end, the adversary can try all possible 2^n keys and filter the wrong ones by asking enough queries from the oracle. Intuitively each oracle query reveals one bit of information about the secret key if F mixes its input bits well enough to be treated as a random Boolean function with $n + m$ input bits. Therefore, assuming $\log_2 n \ll m$, the n key bits can be recovered by sending $O(n)$ queries to the oracle. More precisely if the adversary asks the oracle $n + \beta$ queries for some integer $\beta \gg 0$, then the probability that only the unknown chosen key by the oracle (*i.e.*, the correct candidate) satisfies these queries while all the remaining $2^n - 1$ keys fail to satisfy all the queries is $(1 - 2^{-(n+\beta)})^{2^n-1} \approx e^{-2^{-\beta}}$ (for $\beta = 10$ it is about $1 - 10^{-3}$). A wrong key is rejected after two query evaluations on average. Therefore, the required time complexity is $O(2^n)$. However, if F extremely deviates from being treated as a random function, the secret key bits may not be determined uniquely. It is easy to argue that F divides $\mathbb{B}^n$ into *equivalence classes*. Two keys K' and K'' belong to the same equivalence class if and only if $F(V, K') = F(V, K'')$ for all $V \in \mathbb{B}^m$. The following lemma is a trivial statement.

Lemma 1. *Let* $F(V, K) = \bigoplus_\kappa \Gamma_\kappa(V) K^\kappa$ *where* $K^\kappa = k_1^{\kappa_1} \cdots k_n^{\kappa_n}$ *for the multi-index* $\kappa = (\kappa_1, \ldots, \kappa_n) \in \mathbb{B}^n$. *No adversary can distinguish between the two keys* K' *and* K'' *for which* $K'^\kappa = K''^\kappa$ *for all* $\kappa \in \mathbb{B}^n$ *such that* $\Gamma_\kappa(V) \neq 0$.

Indeed, given the 2^m values of $\{F(V, K) \mid V \in \mathbb{B}^m\}$ for a key $K \in \mathbb{B}^n$, it is only possible to determine the values of $\{K^\kappa | \forall \kappa, \Gamma_\kappa(V) \neq 0\}$ which does not necessarily have a unique solution for K. As a consequence of Lemma 1, the function F divides

$\mathbb{B}^n$ into $J \leq 2^n$ equivalence classes $\mathcal{K}_1,\ \mathcal{K}_2, \ldots, \mathcal{K}_J$. Let n_i denotes the number of keys which belong to the equivalence class $\mathcal{K}_i$. Note that we have $\sum_{i=1}^{J} n_i = 2^n$. A random key lies in the equivalence class $\mathcal{K}_i$ with probability $n_i/2^n$ in which case $(n - \log_2 n_i)$ bits of information can be achieved about the key. The adversary on average can get $\sum_{i=1}^{J}(n - \log_2 n_i)\frac{n_i}{2^n}$ bits of information about the n key bits by asking enough queries. It is difficult to estimate the minimum number of needed queries due to the statistical dependency between them. It highly depends on the structure of F but we guess that, often, $O(n)$ queries suffice again. However, in the case where F does not properly mix its input bits, there might be faster methods than exhaustive search for key recovery. We are interested in key-recovery algorithms which are faster than exhaustive search in this case. The remaining part of this chapter probes this issue.

3.4 Basic idea and possible scenarios

The intuition behind our idea, to provide key-recovery cryptanalytic algorithms which are faster than exhaustive search, is mainly based on [231, 109, 214, 111, 191] and can be explained as follows. The algebraic description of the function $F(V, K)$ is too complex in general to be amenable to direct analysis. If one derives a weaker keyed function $\Gamma(W, K) : \mathbb{B}^{m-l} \times \mathbb{B}^n \rightarrow \mathbb{B}$ from F which depends on the same key and a part of the public variables, the adversary-oracle interaction can still go on through Γ this time. Our main idea is to derive such functions from the *algebraic expansion* of F by making a partition of the m-bit public variable V into $V = (U; W)$ with l-bit vector U and $(m - l)$-bit vector W. One can evaluate such derived functions with the help of F oracle. In our main example, $\Gamma(W, K)$ is a coefficient of the algebraic normal form of the function deduced from F by varying over the bits in U only. We return to this issue in section 3.5. If the function $\Gamma(W, K)$ does not have a well-distributed algebraic structure, it can be exploited for cryptanalysis. Let us investigate different scenarios:

1. If $\Gamma(W, K)$ is imbalanced for (not necessarily uniformly) random W and many fixed K, then the function F (or equivalently the underlying stream cipher) with unknown K can be distinguished from a random one.

2. If $\Gamma(W, K)$ is evaluated for some fixed W, then $\Gamma(W, K)$ is an expression in the key bits only. The simpler this expression is, the better it can be exploited.

3. If $\Gamma(W, K)$ has many key bits which have (almost) no influence on the values of $\Gamma(W, K)$, a suitable approximation may be identified and exploited for key recovery cryptanalytic algorithms.

Scenario 1 has already been discussed in the introduction. We do not explore it further since it considers distinguishers while we are interested in key recovery algorithms. The interested reader is referred to [109, 191, 214, 111] for more details.

In scenario 2, the underlying idea is to find a relation $\Gamma(W, K)$, evaluated for some fixed W, which depends only on a subset of t ($< n$) key bits. The functional form of this relation can be determined with 2^t evaluations of $\Gamma(W, K)$. By trying all 2^t possibilities for the involved t key bits, one can filter those keys which do not satisfy the imposed relation. The complexity of this precomputation is 2^t times needed to compute $\Gamma(W, K)$, see section 3.5. More precisely, if $p = \Pr\{\Gamma(W, K) = 0\}$ for the fixed W, the key space is filtered by a factor of $p^2+(1-p)^2$. The interesting situation happens when several simple imposed relations on the key bits are available. For example, if a lot of linear relations on the key bits are available, one combines them with Gaussian elimination to reduce the key search space efficiently. This case is essentially, what is done by Vielhaber in the AIDA attack [231].

In scenario 3, *our* main idea is to find a function $A(W, K_s)$ which effectively depends on a subvector K_s of the key of size $t < n$, and which is correlated to $\Gamma(W, K)$ with a nonzero correlation coefficient. Then, by asking the oracle enough queries we get some information about t bits of the secret key by carefully analyzing the underlying hypothesis testing problem. This scenario is the target scenario of this chapter and will be discussed in detail. We will proceed by explaining how to derive such functions Γ from the coefficients of the ANF of F in section 3.5, and how to find such approximation functions A in section 3.6.

3.5 Derived functions from polynomial description

In this section we explain how to derive functions Γ from the coefficients of the ANF of F. Consider the algebraic normal form $F(V, K) = \bigoplus_{\nu,\kappa} \Gamma_{\nu,\kappa} V^\nu K^\kappa$, with binary coefficients $\Gamma_{\nu,\kappa}$ where $\nu \in \mathbb{B}^m$ and $\kappa \in \mathbb{B}^n$. Make the partitionings $V = (U; W)$ for the public variable V and $\nu = (\alpha; \beta)$ for the multi-index ν with l-bit segments U and α, and $(m - l)$-bit segments W and β such that $V^\nu = U^\alpha W^\beta$. Therefore, we

have the expression $F(V,K) = \bigoplus_{\alpha,\beta,\kappa} \Gamma_{(\alpha;\beta),\kappa} U^\alpha W^\beta K^\kappa = \bigoplus_\alpha \Gamma_\alpha(W,K) U^\alpha$, where $\Gamma_\alpha(W,K) = \bigoplus_{\beta,\kappa} \Gamma_{(\alpha;\beta),\kappa} W^\beta K^\kappa$. For every $\alpha \in \mathbb{B}^l$, the function $\Gamma_\alpha(W,K)$ can serve as a function Γ derived from F. Here is a toy example to illustrate the notation.

Example 1. *Let $n = m = 3$ and $F(V,K) = k_1 v_1 \oplus k_2 v_0 v_2 \oplus v_2$. Take $U = (v_0, v_2)$ of $l = 2$ bits and $W = (v_1)$ of $m - l = 1$ bit. Then $\Gamma_0(W,K) = k_1 v_1$, $\Gamma_1(W,K) = 0$, $\Gamma_2(W,K) = 1$, $\Gamma_3(W,K) = k_2$.*[1] □

Note that an adversary with the help of the oracle can evaluate $\Gamma_\alpha(W,K)$ for the unknown key K at any input $W \in \mathbb{B}^{m-l}$ for every $\alpha \in \mathbb{B}^l$ by sending at most 2^l queries to the oracle. In other words, the partitioning of V has helped us to define a computable function $\Gamma_\alpha(W,K)$ (as a candidate for Γ) for small values of l, even though the explicit form of $\Gamma_\alpha(W,K)$ remains unknown. To obtain the values $\Gamma_\alpha(W,K)$ for *all* $\alpha \in \mathbb{B}^l$, an adversary asks for the output values of all 2^l inputs $V = (U;W)$ with the fixed part W. This gives the truth table of a Boolean function in l variables for which the coefficients of its ANF (*i.e.*, the values of $\Gamma_\alpha(W,K)$) can be found in time $l2^l$ and memory 2^l using the so-called binary Möbius transform [86]. Alternatively, a *single* coefficient $\Gamma_\alpha(W,K)$ for a specific $\alpha \in \mathbb{B}^l$ can be computed, without memory requirements, by XORing the output of F for all $2^{\mathrm{wt}(\alpha)}$ inputs $V = (U;W)$ for which each bit of U is at most as large as the corresponding bit of α. In other words, we have the following equation

$$\Gamma_\alpha(W,K) = \bigoplus_{U \le \alpha} F((U;W),K) \ . \tag{3.1}$$

One can expect that a subset of public variable bits are not well mixed with other variables in a not-so-random function F (*e.g.*, in the functions related to the reduced-round variants of the initialization of stream ciphers). We call such a subset (or vector) of public variable bits *a set of weak public variables* in general. In the context of stream ciphers, we call them *a set of weak IV variables*. To avoid existence of any set of weak IV variables, the initialization process of a stream cipher must be long enough such that all of the IV variables are well mixed with themselves as well as with the key variables at the end. It is an open question how to identify a set of weak IV variables by systematic methods.

[1] We also denote a general multi-index $\alpha = (\alpha_1, \ldots, \alpha_t)$ by its integer representation $\sum_{i=1}^t \alpha_t 2^{t-1}$.

3.6 Functions approximation

We are interested in the approximations of a given Boolean function $\Gamma(W, K) : \mathbb{B}^{m-l} \times \mathbb{B}^n \rightarrow \mathbb{B}$ which depend only on a subset of key bits. The astute reader may notice that our probabilistic neutral bit concept in chapter 2 is a useful tool for this goal. Therefore, we make an appropriate partition of the key K according to $K = (K_s; K_n)$, with K_s containing t *significant* key bits and K_n containing the remaining $(n-t)$ *non-significant* key bits, and construct the function $A(W, K_s)$. We also use the term *subkey* to refer to the set of significant key bits. Such a partitioning can be systematically identified, using the definition of neutrality measure, see Definition 3. We bring the definition here for convenience.

Definition 4. *The neutrality measure of the ith key bit with respect to a function* $\Gamma(K, W)$ *is defined as* γ_i*, where* $\frac{1}{2}(1 + \gamma_i)$ *is the probability (over all* K *and* W*) that complementing the ith key bit does not change the output of* $\Gamma(K, W)$*.*

In practice, a threshold $-1 \leq \gamma \leq 1$ is chosen and then the key bits are divided into the set of significant and non-significant key bits according to this threshold value and their neutrality measure. A key bit is recognized as significant if and only if its neutrality measure is not bigger than the threshold value. Then the approximation function $A(W, K_s)$ is defined to be the same as Γ except that the non-significant key bits, K_n, are set to a fixed value (*e.g.* zero) in K. Note that, if K_n consists only of neutral key bits (with $\gamma_i = 1$), then the approximation A is exact, because $\Gamma(W, K)$ does not depend on these key bits.

Example 2. *Let* $n = m = 3$*,* $l = 2$ *and* $\Gamma(W, K) = k_0k_1k_2v_0v_1 \oplus k_0v_1 \oplus k_1v_0$*. For uniformly random* K *and* W*, we find* $\gamma_0 = 1/8$*,* $\gamma_1 = 1/8$*,* $\gamma_2 = 7/8$*. Consequently, it is reasonable to use* $K_s = (k_0, k_1)$ *as the subkey. With fixed* $k_2 = 0$*, we obtain the approximation* $A(W, K_s) = k_0v_1 \oplus k_1v_0$ *which depends on* $t = 2$ *key bits only.* □

3.7 Description and evaluation of the cryptanalytic algorithm

Suppose that we have already a derived function $\Gamma(W, K)$ for a function F as well as an approximation $A(W, K_s)$ of Γ for a partitioning $K = (K_s; K_n)$. For example, Γ can be

constructed from the algebraic coefficients of F by randomly looking for a set of l weak public variables, see section 3.5. As already mentioned in section 3.6, the partitioning of the key bits can be easily done according to the neutrality measure of key bits. In this section we study how to find a small subset of candidates for the subkey K_s by filtering some of the wrong ones with a probabilistic guess-and-determine procedure.

In order to filter the set of all 2^t possible subkeys into a smaller set, we need to distinguish an incorrect guess for the subkey from the correct subkey. Let $\hat{K}_s$ denote a guess for the subkey. Our ability in distinguishing subkeys is related to the correlation coefficient between $A(W, \hat{K}_s)$ and $\Gamma(W, K)$ with $K = (K_s, K_n)$ under the following two hypotheses.

- H_0 : the guessed part $\hat{K}_s$ is correct
- H_1 : the guessed part $\hat{K}_s$ is incorrect.

More precisely, under these two hypotheses, the values of ε_0 and ε_1 defined in the following play a crucial role:

$$\Pr_{W}\{A(W, \hat{K}_s) = \Gamma(W, K) | K = (\hat{K}_s, K_n)\} = \frac{1}{2}(1 + \varepsilon_0) \tag{3.2}$$

$$\Pr_{W, \hat{K}_s}\{A(W, \hat{K}_s) = \Gamma(W, K) | K = (K_s, K_n)\} = \frac{1}{2}(1 + \varepsilon_1)\ . \tag{3.3}$$

In general, both ε_0 and ε_1 are random variables, depending on the key. In the case that the distributions of ε_0 and ε_1 are separated, we can achieve a small non-detection probability p_{mis} and false alarm probability p_{fa} by using enough samples. In the special case where ε_0 and ε_1 are constants with $\varepsilon_0 > \varepsilon_1$, the optimum distinguisher is Neyman-Pearson [85]. Then, N values of $\Gamma(W, K)$ for different W (assuming that the samples $\Gamma(W, K)$ are independent) are sufficient to obtain $p_{\text{fa}} = 2^{-c}$ and $p_{\text{mis}} = 1.3 \times 10^{-3}$, where

$$N \approx \left(\frac{\sqrt{c(1 - \varepsilon_0^2) \ln 4} + 3\sqrt{1 - \varepsilon_1^2}}{\varepsilon_1 - \varepsilon_0} \right)^2 . \tag{3.4}$$

The filtering will be successful with probability $1 - p_{\text{mis}}$. In other words, with probability $1 - p_{\text{mis}}$, the filtered set of subkeys, which has an approximate size of $p_{\text{fa}} 2^t = 2^{t-c}$, includes the correct subkey. The complexity of a cryptanalytic algorithm based on this

filtering is as follows. For each guess $\hat{K}_s$ of the subkey, the correlation coefficient ε of $A(W, \hat{K}_s) \oplus \Gamma(W, K)$ must be computed. This requires computation of the approximation function $A(W, \hat{K}_s)$ by the adversary, and computation of the function $\Gamma(W, K)$ through the oracle, for the same N values of W. The cost of this part is at most $N2^l$ for each guess of the subkey, assuming that Γ is derived from a coefficient of F using a set of weak public variables of size l. This must be repeated for all 2^t possible guesses $\hat{K}_s$. The set of filtered candidates for the subkey K_s has a size of about 2^{t-c}. The whole key can then be verified by an exhaustive search over the non-significant key part K_n with a cost of $2^{t-c}2^{n-t}$ evaluations of F. The total complexity becomes $N2^l2^t + 2^{t-c}2^{n-t} = N2^{l+t} + 2^{n-c}$.

Remark 3. *In practice, the values of ε_0 and ε_1 are key dependent. If the key is considered as a random variable, then ε_0 and ε_1 are also random variables. However, their distribution may not be fully separated, and hence a very small p_{mis} and p_{fa} may not be possible to achieve. We propose the following non-optimal distinguisher: first, we choose a threshold $\varepsilon_0^\star$ such that $p_\epsilon = \Pr\{\varepsilon_0 > \varepsilon_0^\star\}$ has a significant value, e.g., $1/2$. We also identify a threshold $\varepsilon_1^\star$, if possible, such that $\Pr\{\varepsilon_1 < \varepsilon_1^\star\} = 1$. Then, we estimate the sample size using equation (3.4) by replacing ε_0 and ε_1 by $\varepsilon_0^\star$ and $\varepsilon_1^\star$, respectively, to obtain $p_{\text{fa}} \leq 2^{-c}$ and effective success probability $(1 - p_{\text{mis}})p_\epsilon \approx 1/2$. If $\varepsilon_0^\star$ and $\varepsilon_1^\star$ are close, then the estimated number of samples becomes very large. In this case, it is better to choose the number of samples intuitively, and then estimate the related p_{fa}.*

Remark 4. *It is reasonable to assume that a false subkey $\hat{K}_s$, which is close to the correct subkey, may lead to a larger value of ε, and hence a large p_{fa}. Here, the measure for being "close" could be the neutrality measure γ_i and the Hamming weight: if only a few key bits on positions with large γ_i are false, one would expect that ε is large. We believe that the effect of this behaviour on p_{fa} is often negligible because in most of the cases the portion of "close" subkeys is negligible compared to the size of the whole subkey space.*

3.8 Application to Trivium

Trivium [68] is one of the stream ciphers in the final portfolio of the eSTREAM project. It consists of three shift registers of different lengths with an internal state of 288 bits. At each round, a bit is shifted into each of the three shift registers using a non-linear combination of taps from that and one other register; and then one keystream bit is

produced using a linear output function. To initialize the cipher, the $n = 80$ key bits and $m = 80$ IV bits are written into two of the shift registers, with the remaining bits being set to a fixed pattern. The cipher state is then updated $R = 18 \times 64 = 1152$ times without producing output in order to provide a good mixture of the key and IV bits in the initial state. We consider the Boolean function $F(V, K)$ which computes the first keystream bit after r rounds of initialization. In [109], Trivium was analyzed with chosen IV statistical tests and non-randomness was detected for $r = 10 \times 64,\ 10.5 \times 64,\ 11 \times 64,\ 11.5 \times 64$ rounds with $l = 13,\ 18,\ 24$ and 33 IV bits, respectively. In [231], the key recovery cryptanalysis of Trivium was investigated with respect to scenario 2 (see section 3.4) for $r = 9 \times 64$ rounds. Here we provide more examples for key recovery cryptanalysis with respect to scenario 3 for $r = 10 \times 64$ and $r = 10.5 \times 64$. In the following two examples, the sets of weak IV variables have been found by a random search. We first concentrate on equivalence classes of the key.

Example 3. *For $r = 10 \times 64$ rounds, the set of $l = 10$ weak IV variables with the bit positions $\{34, 36, 39, 45, 63, 65, 69, 73, 76, 78\}$ for U, and the coefficient with index $\alpha = 1023$ (corresponding to the monomial of maximum degree), we could experimentally verify that the derived function $\Gamma_\alpha(W, K)$ only depends on $t = 10$ key bits with bit positions $\{15, 16, 17, 18, 19, 22, 35, 64, 65, 66\}$. By assigning all 2^{10} different possible values to these 10 key bits and putting those K_s's which gives the same function $\Gamma_\alpha(W, K)$ (by trying enough samples of W), we could determine the equivalence classes for K_s with respect to Γ_α. Our experiment shows the existence of 65 equivalence classes: one with 512 members for which $k_{15}k_{16} + k_{17} + k_{19} = 0$ and 64 other classes with 8 members for which $k_{15}k_{16} + k_{17} + k_{19} = 1$ and the vector $(k_{18}, k_{22}, k_{35}, k_{64}, k_{65}, k_{66})$ has a fixed value. This shows that Γ_α provides $\frac{1}{2} \times 1 + \frac{1}{2} \times 7 = 4$ bits of information about the key on average.* □

Example 4. *For $r = 10 \times 64$ rounds, the set of $l = 11$ weak IV variables with the bit positions $\{1, 5, 7, 9, 12, 14, 16, 22, 24, 27, 29\}$ for U, and the coefficient with index $\alpha = 2047$ (corresponding to the monomial of maximum degree), the derived function $\Gamma_\alpha(W, K)$ depends on all 80 key bits. A more careful look at the neutrality measure of the key bits reveals that $\max(\gamma_i) \approx 0.35$ and only 7 key bits have a neutrality measure larger than $\gamma = 0.18$, which is not enough to get a useful approximation $A(W, K_s)$ for an efficient cryptanalytic algorithm. However, we observed that $\Gamma_\alpha(W, K)$ is independent of the key for $W = 0$, and more generally the number of significant bits depends on $\mathrm{wt}(W)$.* □

It is difficult to find a good choice of a set of weak IV variables for larger values of r, using a random search. The next example shows how we can go a bit further with some insight on the initialization procedure of the cipher.

Example 5. *Now we consider* $r = 10.5 \times 64 = 10 \times 64 + 32 = 672$ *rounds. The construction of the initialization function of Trivium suggests that shifting the bit positions of* U *in Example 4 may be a good choice for a set of weak IV variables. Hence, we choose* U *with the* $l = 11$ *bit positions* $\{33, 37, 39, 41, 44, 46, 48, 54, 56, 59, 61\}$*, and* $\alpha = 2047$ *(corresponding to the monomial of maximum degree). In this case,* $\Gamma_\alpha(W, K)$ *for* $W = 0$ *is independent of 32 key bits, and* $p = \Pr\{\Gamma_\alpha(0, K) = 1\} \approx 0.42$*. This is already a cryptanalysis which is* $1/0.42 \approx 1.95$ *times faster than exhaustive search.* □

The following example shows how we can connect a bridge between scenarios 2 and 3 and come up with a cryptanalytic algorithm which is much faster than exhaustive search.

Example 6. *Consider the same setup as in Example 5. If we restrict ourselves to* W*'s with* $\mathrm{wt}(W) = 5$ *and compute the value of* γ_i *conditioned over this subset of* W*'s, then* $\max_i(\gamma_i) \approx 0.68$*. Assigning all key bits with* $\gamma_i \leq \gamma = 0.25$ *as significant, we obtain a significant key part* K_s *with the* $t = 29$ *bit positions* $\{1, 3, 10, 14, 20, 22, 23, 24, 25, 26, 27, 28, 31, 32, 34, 37, 39, 41, 46, 49, 50, 51, 52, 57, 59, 61, 63, 68, 74\}$*. Our analysis of the approximation function* $A(W, K_s)$ *shows that for about* 44% *of the keys we have* $\varepsilon_0 > \varepsilon_0^\star = 0.2$ *when the subkey is correctly guessed. If the subkey is not correctly guessed, we observe* $\varepsilon_1 < \varepsilon_1^\star = 0.15$*. Then, according to equation (3.4) the correct subkey of* 29 *bits can be detected using at most* $N \approx 2^{15}$ *samples, with time complexity* $N2^{l+t} \approx 2^{55}$*. Note that* N *must be smaller than the total number of* W*'s with* $\mathrm{wt}(W) = 5$*, i.e.,* $N < \binom{69}{5}$*. This condition is satisfied here.* □

3.9 Application to Grain-128

The stream cipher Grain-128 [126] is one of the eSTREAM phase 3 candidates. It consists of an LFSR, an NFSR and a nonlinear output function. Grain-128 has $n = 128$ key bits, $m = 96$ IV bits and the full initialization function has $R = 256$ rounds. We again consider the Boolean function $F(V, K)$ which computes the first keystream bit of Grain-128 after r rounds of initialization. In [109], Grain-128 was analyzed with chosen IV statistical tests. With $l = 22$ variable IV bits, they observed a non-randomness of the first keystream bit after $r = 192$ rounds. They also observed a non-randomness in

the initial state bits after the full number of rounds. In [191], a non-randomness up to 313 rounds was reported (without justification). In this section we provide key recovery cryptanalysis for up to $r = 180$ rounds with slightly reduced complexity compared with exhaustive search. In the following example, the set of weak IV variables for scenario 2 have been found again by a random search.

Example 7. *Take the set of $l = 7$ weak IV variables with the bit positions $\{2, 6, 8, 55, 58, 78, 90\}$ for U. For the coefficient with index $\alpha = 127$ (corresponding to the monomial of maximum degree), a significant imbalance for up to $r = 180$ rounds can be detected: the monomial of degree 7 appears only with a probability of $p < 0.2$ for 80% of the keys. Note that in [109], the cryptanalytic algorithm with $l = 7$ could be successfully applied only up to $r = 160$ rounds.* □

In the following examples, our goal is to show that there exists a cryptanalytic algorithm for up to $r = 180$ rounds on Grain-128 which is slightly faster than exhaustive key search.

Example 8. *Consider again the $l = 7$ IV bits U with bit positions $\{2, 6, 8, 55, 58, 78, 90\}$. For $r = 150$ rounds we choose the coefficient with index $\alpha = 117$ (corresponding to a monomial of* non-*maximum degree) and include key bits with neutrality measure less than $\gamma = 0.98$ in the list of the significant key bits. This gives a subkey K_s of $t = 99$ bits. Our simulations show that $\varepsilon_0 > \varepsilon_0^\star = 0.95$ for about 95% of the keys, hence $p_{mis} = 0.05$. On the other hand, for 128 wrong guesses of the subkey with $N = 200$ samples, we never observed that $\varepsilon_1 > 0.95$, hence we estimate $p_{fa} < 2^{-7}$. This gives a cryptanalytic algorithm with time complexity $N2^{t+l} + 2^n p_{fa} \approx 2^{121}$ which is an improvement of a factor of (at least) $1/p_{fa} = 2^7$ compared to exhaustive search.* □

Example 9. *With the same choice for U as in Examples 7 and 8, we take $\alpha = 127$ (corresponding to the monomial of maximum degree) for $r = 180$ rounds. We identified $t = 110$ significant key bits for K_s. Our simulations show that $\varepsilon_0 > \varepsilon_0^\star = 0.8$ in about 30% of the runs when the subkey is correctly guessed. For 128 wrong guesses of the subkey with $N = 128$ samples, we never observed that $\varepsilon_1 > 0.8$. Here we have a cryptanalytic algorithm with time complexity $N2^{t+l}+2^n p_{fa} \approx 2^{124}$, i.e., an improvement of a factor of 2^4.* □

3.10 Connection with previous works

In the literature there are a fair number of articles discussing the security of stream ciphers with regards to the initialization procedure [90, 115, 63, 122, 111, 136, 108, 123, 9, 164, 163, 242, 77, 214, 29, 109, 231]. The technique of this chapter is in the line of that of [231, 109, 191, 214, 111], in the sense that all of them consider the algebraic description of the initialization procedure. The method is also closely related to the differential [48], higher-order differential [155], square [93], saturation [165] and integral [158] attacks, also collectively referred to as multiset attack [56], see section 1.5.5. The resemblance comes from the fact that in all of these methods, a specific function is summed over a certain set (or more generally multiset) of inputs, see equation (3.1). However, here we observed a behavior which is less frequent in other methods. In some of the examples in this chapter, the values for W were often chosen adaptively. By adaptively we mean that a stronger deviation from randomness is observed for some restricted choices for W (*e.g.*, low weight W's) or even a particular value for W (*e.g.*, W = 0). Whereas in most applications of differential-related cryptanalysis, specific input values are of no favor.

3.11 Follow-ups of our work

Following our work, Dinur and Shamir published the cube attack [100] which is essentially the same as Vielhaber's AIDA attack [231] that we discussed under scenario 2 in section 3.4. Our method is mainly statistical whereas AIDA/cube attack is totally algebraic. Dinur and Shamir improved the previous results due to the following contributions. They remarked that for a random Boolean function $F(V, K)$ with m-bit public variable V and n-bit secret key K whose degree is limited to d, any subset of IV variables of size $d-1$ is weak. More precisely, if U is any subvector of size $d-1$ of the public variables, the derived function $\Gamma(W, K) = \Gamma_{2^{d-1}-1}(W, K)$ is an affine function where $W = V \setminus U$. In this case, the function Γ corresponds to the coefficient of the highest degree monomial, *i.e.*, the coefficient of $\prod_{i=1}^{d-1} u_i$ when F is described in terms of variables in U. As it was later pointed out by Bernstein [37], this is a trivial corollary of Lai's work [159] published in 1994, which can be explained as follows. The function Γ is a derivative of order $d-1$ of the original function F. Since, each derivative reduces the degree at least by one, Γ trivially has degree at most one, *i.e.*, an affine function.

Interestingly, based on this simple observation, it can be easily argued that AIDA/cube attack has a complexity of $n2^{d-1} + n^2$ under the above assumption for F [100]. Since the degree evolves very slowly in Trivium, this cute observation motivated Dinur and Shamir to look for sets of weak IV variables of larger size. In our examples in this chapter we found the sets of weak IV variables mainly using random search. Most remarkably, Dinur and Shamir developed a *random-walk-based heuristic search* for finding sets of weak public variables (which they call it "appropriate maxterms"). This method helped them to significantly improve the previous results by finding a lot of appropriate maxterms, after few weeks of preprocessing stage. In particular, for Trivium with 672 initialization rounds (for which we require a time complexity of order 2^{55} with a set of 11 weak IV variables, see Example 6), they found 63 appropriate maxterms of size 12, yielding an amazingly low total complexity of around 2^{19}. For Trivium with 767 initialization rounds, they found 35 appropriate maxterms of size 28-31 which gives rise to a cryptanalytic algorithm in time 2^{45}.

Later, Aumasson *et al.* introduced the cube testers [14] which led to demonstrate nonrandom properties for Trivium up to 885 initialization rounds. In [16], cube testers were implemented using an efficient FPGA hardware to find a set of weak IV variables of size 40 using an *evolutionary-based search algorithm* to provide a distinguisher for Grain-128 with 237 initialization rounds in time 2^{40}. By extrapolation, it is conjectured in [16] that a successful distinguisher in time 2^{83} can be constructed for the full Grain-128, *i.e.*, with 256 initialization rounds.

Finding a set of weak public variables is essential for the success of this kind of cryptanalytic algorithms. The random-walk and evolutionary search algorithms turn out to be very good solutions for this crucial problem. In chapter 4, published before [100] and [16] in [149], we use a more systematic method to find a set of weak public variables, rather than random search, to cryptanalyze a self-synchronizing stream cipher based on the ideas of this chapter.

3.12 Summary

The framework for chosen IV statistical distinguishers for stream ciphers has been exploited to provide new methods for key recovery algorithms. This is based on a polynomial description of output bits as a function of the key and the IV. A deviation

of the algebraic normal form (ANF) from random indicates that not every bit of the key or the IV has full influence on the value of certain coefficients in the ANF. It has been demonstrated how this can be exploited to derive information on the key faster than exhaustive key search through approximation of the polynomial description and using the concept of probabilistic neutral key bits. This answers positively the question whether statistical distinguishers based on polynomial descriptions of the IV initialization of a stream cipher can be successfully exploited for key recovery. As a proof, two applications of our methods through extensive experiments have been given: a reduced complexity key recovery for Trivium with IV initialization reduced to 672 of its 1152 iterations, and a reduced complexity key recovery for Grain-128 with IV initialization reduced to 180 of its 256 iterations. Our key recovery results on Trivum has been improved in [100]. Newer cryptanalyses of Grain-128 [16, 69] discuss distinguishers based on cube testers [14] and related-key cryptanalysis based on sliding properties [58] of the initialization procedure of the cipher. Part of our results, still remains the best cryptanalytic results on Grain-128 as of April 2010.

4

Chosen Ciphertext Cryptanalysis for Self-synchronizing Stream Ciphers

In cryptology we commonly face the problem of finding a secret key from the output of an interactable keyed function. The interactable function depends on a secret and a public variable. The public variable is a controllable input, making the function interactable in the following sense. An oracle randomly chooses a key and keeps it unknown to an adversary. He then provides the adversary with the output value of the function (computed under the fixed unknown key) for any chosen value for the public variable of adversary's choice. The goal of the adversary is to recover the secret key as efficiently as possible. In chapter 3, we studied this problem for synchronous stream ciphers. In that case, the public variable were the initial value (IV) and the interactable function were simply the function which maps the key and IV into, for example, the first keystream bit. In this chapter, we focus on self-synchronizing stream ciphers. First we show how to model these primitives in the above-mentioned general problem by relating appropriate interactable functions to the underlying ciphers as it is less trivial than the case of synchronous stream ciphers. Then we apply the framework of chapter 3 for dealing with this kind of problems to the proposed T-function based self-synchronizing stream cipher by Klimov and Shamir [152] at FSE 2005. Our results, originally presented in [149] at INDOCRYPT 2008, show how to deduce some non-trivial information about the key for this cipher.

4.1 Introduction

The area of stream cipher design and analysis has made a lot of progress recently, mostly spurred by the eSTREAM [106] project. Thanks to this project, it is now a common belief that designing elegant strong synchronous stream ciphers is possible. In contrary, there is no such belief about self-synchronizing stream ciphers (SSSCs). There were only two SSSCs among the initial 34 eSTREAM candidates, both of which revealed serious weaknesses. Apparently, SSSCs have been (and maybe still are) widely used in industrial, diplomatic and military communications [88], nevertheless, their future deployment is unclear [18]. However, from a theoretical point of view, it has remained a very challenging issue for almost two decades to come up with suitable designs for SSSCs. There are very few articles in the literature studying SSSCs. Dedicated constructions for SSSCs were first introduced by Maurer [170] in 1991, see [200] for an older work. The remaining articles are devoted to concrete proposals and their cryptanalyses. In 1992, Daemen *et al.* reconsidered the design of dedicated SSSCs from a practical point of view [89] and proposed KNOT as an efficient concrete SSSC. KNOT was later broken by Joux and Muller in 2003 [137]. There are several tweaked versions of KNOT. In his Ph.D. thesis, Daemen detected a statistical imbalance in the output of KNOT and suggested the improved cipher $\Upsilon\Gamma$. In 2005, Daemen and Kitsos proposed the SSSC Mosquito [91] as a candidate to the eSTREAM call for primitives. Soon after in 2006, Joux and Muller [139] proposed successful cryptanalytic algorithms for Mosquito as well as $\Upsilon\Gamma$. As a response, Mosquito was tweaked and Moustique [92] was proposed, which was among the eSTREAM phase-3 candidates. However, it was excluded from the final protfolio [19, 20] due to the successful cryptanalysis of [144]. SSS [213], designed by Rose *et al.*, was the other eSTREAM SSSC candidate which remained a phase 1 candidate due to a devastating cryptanalysis by Daemen *et al.* [94]. HBB (Hiji-bij-bij) is another SSSC designed by Sarkar [215] in 2003 which was defeated by Mitra [176], and Joux and Muller [138]. In 2004, a dedicated SSSC by Arnault and Berger [11] based on Feedback with Carry Shift Registers (FCSR) was broken by Zhang *et al.* [244] as well. To the best of our knowledge, the Klimov-Shamir T-function based SSSC, proposed in 2005 [152], is the only remaining design which has not been subject to cryptanalysis so far. We would like to mention that in addition to the above-mentioned dedicated

designs, a block cipher in a Cipher FeedBack (CFB) mode [112] operates as a SSSC; see also OCFB [5] and SCFB [141], and the related articles [199, 131, 117] as well.

In this work we first show how to model a SSSC by a family of interactable keyed functions $F(C, \mathsf{K_e})$. The input parameter C, called the *public variable*, can be controlled by the adversary while the input $\mathsf{K_e}$ is an unknown parameter to her called the *extended key*. The public variable corresponds to the ciphertext whereas the extended key is a combination of the actual key used in the cipher and the unknown internal state of the cipher. The goal of the adversary would be to recover $\mathsf{K_e}$ or to get some information about it. The problem of finding the unknown $\mathsf{K_e}$, when access is given to the output of an interactable function $F(C, \mathsf{K_e})$ for every C of the adversary's choice, is a very common problem encountered in cryptography. In general, when the keyed function F looks like a random function, the best way to solve the problem is to exhaust the key space. However, there might be more efficient methods if F is far from being a random function. In chapter 3, we studied how to recover the key faster than by exhaustive search in case F does not properly mix its input bits. The idea was to first identify a subset of variable bits from C referred to as *a set of weak public variables* and then to consider the coefficient of a monomial involving this set of weak public variables in the algebraic normal form of F. If this coefficient does not depend on all the unknown bits of $\mathsf{K_e}$, or it weakly depends on some of them, it can be exploited for efficient cryptanalysis. Having modeled the SSSCs as the above-mentioned general problem, we consider the T-function based SSSC proposed by Klimov and Shamir [152] and use the framework from chapter 3 to deduce some information about the key bits through some striking relations. Finding sets of weak public variables was raised as a crucial open question in chapter 3 which was done mostly by random search there. In this chapter we take a more systematic approach to find sets of weak public variables.

4.2 Polynomial approach for key recovery on an interactable keyed function

In this chapter, we use the framework from chapter 3. For the notations please refer to section 3.2. In chapter 3, the secret variable (the key) and public variable (the initial value) were denoted by K and V, respectively. In this chapter, the secret variable (the extended key) and public variable (the ciphertext) are respectively denoted by $\mathsf{K_e}$ and

C. Moreover, we use the terminology *a set of weak ciphertext variables* instead of a set of weak IV variables or a set of weak public variables. Therefore, the interactable Boolean function is denoted by $F : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}$ which maps the m-bit ciphertext C and the n-bit extended key $\mathsf{K_e}$ into the output bit $z = F(C, \mathsf{K_e})$. In addition, for a subvector $U \in \mathbb{B}^l$ of l weak ciphertext variables, a derived function is denoted by $\Gamma^U(W, \mathsf{K_e})$ where $W = C \setminus U \in \mathbb{B}^{m-l}$.

Our cryptanalytic algorithm in this chapter is based on scenario 3 of section 3.4. However, there are two differences here. First, in this chapter, we only consider the derived functions corresponding to the maximum degree monomial. In chapter 3, we also took advantage of other monomial coefficients. Nevertheless, most of our examples were based on the maximum degree monomial coefficient. Other works [109, 231, 100] also suggest that this monomial is usually more suitable. More precisely, we find a subvector U of l weak ciphertext variables and consider the Boolean function $\Gamma^U : \mathbb{B}^{m-l} \times \mathbb{B}^n \rightarrow \mathbb{B}$ where

$$\Gamma^U(W, \mathsf{K_e}) = \bigoplus_{U \in \mathbb{B}^l} F((U; W), \mathsf{K_e}) \ . \tag{4.1}$$

Inspired by the terminology of [100], we refer to Γ^U as the *superpoly* (corresponding to U). The second difference is that we only take advantage of *neutral* bits. In other words, we do not use approximations of the superpoly by identifying *probabilistic neutral bits*. Hence, we directly analyze the dependency of the superpoly on its arguments. In this chapter we will see a lot of examples for which the superpoly Γ^U does not depend on some of its total $m - l + n$ input bits. That is, some of the input bits are neutral. This is a special case of the third scenario in section 3.4 where probabilistic neutral bits were used instead. Suppose that the superpoly effectively depends on $t_{\mathsf{K_e}} \leq n$ extended key bits and $t_W \leq m - l$ ciphertext bits. Assuming the involved $t_{\mathsf{K_e}} + t_W$ bits are mixed reasonably well and provided that $\log_2 t_{\mathsf{K_e}} \ll t_W$ (see section 3.3), the involved $t_{\mathsf{K_e}}$ secret bits can be recovered in time $O(2^{l+t_{\mathsf{K_e}}})$ by sending $O(t_{\mathsf{K_e}} 2^l)$ queries to the F oracle. Recall that each query to Γ^U oracle costs 2^l queries to F according to equation (4.1). However, if the superpoly extremely deviates from being treated as a random function, as already argued in section 3.3 (see Lemma 1), it may even happen that the $t_{\mathsf{K_e}}$ secret bits cannot be determined uniquely. In this case one has to look at the corresponding equivalence classes to see how much information one can achieve

about the involved $t_{\mathsf{K_e}}$ secret bits on average. In sections 4.5 and 4.6 we will provide some examples by considering Klimov-Shamir self-synchronizing stream cipher.

4.3 Self-synchronizing stream ciphers

A self-synchronizing stream cipher (SSSC) is built on an output filter $\mathcal{O} : \mathcal{K} \times \mathcal{S} \to \mathcal{M}$ and a self-synchronizing state update function (see Definition 5) $\mathcal{U} : \mathcal{M} \times \mathcal{K} \times \mathcal{S} \to \mathcal{S}$, where $\mathcal{S}$, $\mathcal{K}$ and $\mathcal{M}$ are the cipher state space, key space and plaintext space. We suppose that the ciphertext space is the same as that of the plaintext. Let $K \in \mathcal{K}$ be the secret key, and $\{S_i\}_{i=0}^{\infty}$, $\{p_i\}_{i=0}^{\infty}$ and $\{c_i\}_{i=0}^{\infty}$ respectively denote the sequences of cipher state, plaintext and ciphertext. We assume that there is no initial value (IV) involved and the initial state is computed through the initialization procedure as $S_0 = \mathcal{I}(K)$ from the secret key K. The ciphertext (in an additive stream cipher) is then computed according to the following relations:

$$c_i = p_i \oplus \mathcal{O}(K, S_i),\ i \geq 0\ , \tag{4.2}$$

$$S_{i+1} = \mathcal{U}(c_i, K, S_i),\ i \geq 0\ . \tag{4.3}$$

Definition 5. *[152] (SSF) Let $\{c_i\}_{i=0}^{\infty}$ and $\{\hat{c}_i\}_{i=0}^{\infty}$ be two input sequences, let S_0 and $\hat{S}_0$ be two initial states, and let K be a common key. Assume that the function $\mathcal{U}$ is used to update the state based on the current input and the key: $S_{i+1} = \mathcal{U}(c_i, K, S_i)$ and $\hat{S}_{i+1} = \mathcal{U}(\hat{c}_i, K, \hat{S}_i)$. The function $\mathcal{U}$ is called a self-synchronizing function (SSF) if there exist a positive integer r such that the equality of any r consecutive inputs implies the equality of the next state, i.e.,*

$$c_i = \hat{c}_i, \ldots, c_{i+r-1} = \hat{c}_{i+r-1} \Rightarrow S_{i+r} = \hat{S}_{i+r}. \tag{4.4}$$

Definition 6. *The "resynchronization memory" of a function $\mathcal{U}$, assuming it is a SSF, is the least positive value of r such that equation (4.4) holds.*

4.3.1 Cryptanalytic model

In this chapter, we consider key recovery algorithms on SSSCs in a chosen ciphertext scenario. Our goal as a cryptanalyst is to efficiently recover the unknown key K by sending to the decryption oracle chosen ciphertexts of our choice. More precisely, we consider

the family of functions $\{\mathcal{H}_i : \mathcal{M}^i \times \mathcal{K} \times \mathcal{S} \to \mathcal{M} | i = 1, 2, \ldots, r\}$, where r is the resynchronization memory of the cipher and $\mathcal{H}_i(c_1, \ldots, c_i, K, S) = \mathcal{O}(K, \mathcal{G}_i(c_1, \ldots, c_i, K, S))$, where $\mathcal{G}_i : \mathcal{M}^i \times \mathcal{K} \times \mathcal{S} \to \mathcal{S}$ is recursively defined as $\mathcal{G}_{i+1}(c_1, \ldots, c_i, c_{i+1}, K, S) = \mathcal{U}(c_{i+1}, K, \mathcal{G}_i(c_1, \ldots, c_i, K, S))$ with initial condition $\mathcal{G}_1 = \mathcal{U}$.

Note that, due to the self-synchronizing property of the cipher, $\mathcal{H}_r(c_1, \ldots, c_r, K, S)$ is actually independent of the last argument S; however, all other $r-1$ functions depend on their last input. The internal state of the cipher is unknown at each step of operation of the cipher but because of the self-synchronizing property of the cipher it only depends on the last r ciphertext inputs and the key. We take advantage of this property and force the cipher to get stuck in a fixed but unknown state $\mathsf{S}^\star$ by sending the decryption oracle ciphertexts with some fixed prefix $(\mathsf{c}^\star_{-r+1}, \ldots, \mathsf{c}^\star_0)$ of our choice. Having forced the cipher to fall in the unknown fixed state $\mathsf{S}^\star$, we can evaluate any of the functions $\mathcal{H}_i$, $i = 1, 2, \ldots, r$, at any point $(c_1, \ldots, c_i, K, \mathsf{S}^\star)$ for any input $(c_1, \ldots, c_i)$ of our choice by dealing with the decryption oracle. To be clearer let $z = \mathcal{H}_i(c_1, \ldots, c_i, K, \mathsf{S}^\star)$. In order to compute z for an arbitrary $(c_1, \ldots, c_i)$, we choose an arbitrary $\mathsf{c}^\star_{i+1} \in \mathcal{M}$ and ask the decryption oracle for $(p_{-r+1}, \ldots, p_{-1}, p_0, \ldots, p_{i+1})$, that is, the decrypted plaintext corresponding to the ciphertext $(\mathsf{c}^\star_{-r+1}, \ldots, \mathsf{c}^\star_0,\ c_1, \ldots, c_i, \mathsf{c}^\star_{i+1})$. We then set $z = p_{i+1} \oplus \mathsf{c}^\star_{i+1}$.

To make notations simpler, we merge the unknown values K and $\mathsf{S}^\star$ in one unknown variable $\mathsf{K_e} = (K, \mathsf{S}^\star) \in \mathcal{K} \times \mathcal{S}$, called *extended key*. We then use the simplified notation $\mathcal{F}_i(C, \mathsf{K_e}) = \mathcal{H}_i(c_1, \ldots, c_i, K, \mathsf{S}^\star) : \mathcal{M}^i \times (\mathcal{K} \times \mathcal{S}) \to \mathcal{M}$ where $C = (c_1, \ldots, c_i)$.

4.4 Description of the Klimov-Shamir T-function based self-synchronizing stream cipher

Shamir and Klimov [152] used the so-called multiword T-functions for a general methodology to construct a variety of cryptographic primitives. No fully specified schemes were given, but in the case of SSSCs, a concrete example construction was outlined. This section recalls its design. Let $\lll$, $+$, $\times$, $\oplus$ and $\vee$ respectively denote left rotation, addition modulo 2^{64}, multiplication modulo 2^{64}, bit-wise XOR and bit-wise OR operations on 64-bit integers. The proposed design works with 64-bit words and has a 3-word internal state $S = (s_0, s_1, s_2)^T$. A 5-word key $K = (k_0, k_1, k_2, k_3, k_4)$ is used to

define the output filter and the state update function as follows:

$$\mathcal{O}(K,S) = \big((s_0 \oplus s_2 \oplus k_3) \lll 32\big) \times \big(((s_1 \oplus k_4) \lll 32) \vee 1\big), \tag{4.5}$$

and

$$\mathcal{U}\big(c,\ K,\ S\big) = \begin{pmatrix} \big(((s'_1 \oplus s'_2) \vee 1) \oplus k_0\big)^2 \\ \big(((s'_2 \oplus s'_0) \vee 1) \oplus k_1\big)^2 \\ \big(((s'_0 \oplus s'_1) \vee 1) \oplus k_2\big)^2 \end{pmatrix}, \tag{4.6}$$

where

$$\begin{array}{c} s'_0 = s_0 + c\ , \\ s'_1 = s_1 - (c \lll 21)\ , \\ s'_2 = s_2 \oplus (c \lll 43). \end{array} \tag{4.7}$$

4.4.1 Reduced word size variants

We also consider generalized versions of this cipher which use ω-bit words (ω even and typically $\omega = 8, 16, 32$ or 64). For ω-bit version, the number of rotations in the output filter, equation (4.5), is $\frac{\omega}{2}$ and those of the state update function, equation (4.7), are $\lfloor \frac{\omega}{3} \rceil$ and $\lfloor \frac{2\omega}{3} \rceil$, $\lfloor x \rceil$ being the closest integer to x.

It can be shown [152] that the update function $\mathcal{U}$ is actually a SSF whose resynchronization memory is limited to ω steps and hence the resulting stream cipher is self-synchronizing indeed. Our analysis of the cipher for $\omega = 8,\ 16,\ 32$ and 64 shows that it resynchronizes after $r = \omega - 1$ steps (using $\omega(\omega - 1)$ input bits). It is an open question if this holds in general.

Remark 5. *In [152] the notation $(k_0, k_1, k_2, k_{\mathcal{O}}, k'_{\mathcal{O}})$ is used for the key instead of the more standard notation $(k_0, k_1, k_2, k_3, k_4)$. The authors possibly meant to use a 3-word key (k_0, k_1, k_2) by deriving the other two key words ($k_{\mathcal{O}}$ and $k'_{\mathcal{O}}$ in their notations corresponding to k_3 and k_4 in ours) from first three key words. However, they do not specify how this must be done if they meant so. Also they did not introduce an initialization procedure for their cipher. In any case, we cryptanalyze a more general situation where the cipher uses a 5-word secret key $K = (k_0, k_1, k_2, k_3, k_4)$ in chosen-ciphertext cryptanalytic scenario. Moreover, for the 64-bit version, the authors mentioned "the best attack we are aware of this particular example [64-bit version] requires $O(2^{96})$ time", without mentioning the attack.*

4.5 Analysis of the Klimov-Shamir T-function based self-synchronizing stream cipher

Let ω ($\omega = 8, 16, 32$ or 64) denote the word size and $r = \omega - 1$ be the resynchronization memory of the ω-bit version of the Klimov-Shamir self-synchronizing stream cipher. Let $\mathbb{B} = \{0, 1\}$ and $\mathbb{B}_\omega$ denote the binary field and the set of ω-bit words respectively. Following the general model of analysis of SSSCs in section 4.3.1, we focus on the family of functions $\mathcal{F}_i(C, \mathsf{K_e}) : \mathbb{B}_\omega^i \times \mathbb{B}_\omega^8 \to \mathbb{B}_\omega$, $i = 1, 2, \ldots, r$ where $C = (c_1, \ldots, c_i)$ and $\mathsf{K_e} = (K, \mathsf{S}^\star) = (k_0, k_1, k_2, k_3, k_4, \mathsf{s}_0^\star, \mathsf{s}_1^\star, \mathsf{s}_2^\star)$. We also look at a word b as an ω-bit vector $b = (b_0, \ldots, b_{\omega-1})$, b_0 being its LSB and $b_{\omega-1}$ its MSB. Therefore, any vector $A = (a_0, a_1, \ldots, a_{t-1}) \in \mathbb{B}_\omega^t$ could be also treated as a vector in $\mathbb{B}^{t\times\omega}$ where the $(i\omega+j)$th bit of A is $a_{i,j}$, the jth LSB of the word a_i, for $i = 0, 1, \ldots, t-1$ and $j = 0, 1, \ldots, \omega - 1$ (we start numbering the bits of vectors from zero).

Now, for any $i = 1, \ldots, r$ and $j = 0, \ldots, \omega - 1$ we consider the family of interactable Boolean functions $\mathcal{F}_{i,j} : \mathbb{B}^{i\omega} \times \mathbb{B}^{8\omega} \to \mathbb{B}$ which maps the $i\omega$-bit ciphertext C and the 8ω-bit extended key $\mathsf{K_e}$ into the jth LSB of the word $\mathcal{F}_i(C, \mathsf{K_e})$. Any of these interactable keyed functions can be put into the framework from [113] explained in section 4.2. The next step is to identify a set of l weak ciphertext variables and make the partitioning $C = (U; W)$ with l-bit subvector U and $(i\omega - l)$-bit subvector W to derive the (hopefully weaker) functions $\Gamma_{i,j}^U : \mathbb{B}^{i\omega-l} \times \mathbb{B}^{8\omega} \to \mathbb{B}$. Recall that $\Gamma_{i,j}^U$ is the superpoly in $\mathcal{F}_{i,j}$ corresponding to U, see equation (4.1). Whenever there is no ambiguity we drop the superscript or the subscripts. We may also use $\Gamma_{i,j}^U[\omega]$ in some cases to emphasize the word size. We are now ready to give our simulation results.

Note 1. *Instead of presenting the variables in the vector U we give the bit numbers as a set. For example, for $\omega = 16$, the set $\{0, 18, 31, 32\}$ stands for the subvector $U = (c_{1,0}, c_{2,2}, c_{2,15}, c_{3,0})$.*

Example 10. *For all possible common word sizes ($\omega = 8, 16, 32$ or 64), we have been able to find some i, j and U such that Γ is independent of W and only depends on three key bits $k_{0,0}$, $k_{1,0}$ and $k_{2,0}$. Table 4.1 shows some of these quite striking relations. We also found relations $\Gamma_{1,0}^{\{3\}}[8] = 1 + k_{2,0}$ and $\Gamma_{1,0}^{\{6,7,8,9,10\}}[16] = 1 + k_{0,0}$ involving only one key bit. In particular, for $\omega = 64$, the three relations in Table 4.1 give 1.75 bits of information about $(k_{0,0}, k_{1,0}, k_{2,0})$.*

ω	i	j	U	the value of $\Gamma^U_{i,j}[\omega]$
8	2	0	$\{2\}$	$1 + k_{0,0}k_{1,0} + k_{0,0}k_{2,0} + k_{1,0}k_{2,0}$
16	3	0	$\{5\}$	$1 + k_{0,0}k_{1,0} + k_{2,0} + k_{0,0}k_{1,0}k_{2,0}$
16	3	0	$\{10\}$	$1 + k_{0,0} + k_{1,0}k_{2,0} + k_{0,0}k_{1,0}k_{2,0}$
32	5	0	$\{11\}$	$1 + k_{0,0}k_{1,0} + k_{2,0} + k_{0,0}k_{1,0}k_{2,0}$
32	16	0	$\{96, 97, 98\}$	$1 + k_{0,0} + k_{2,0} + k_{0,0}k_{2,0}$
64	11	0	$\{21\}$	$1 + k_{0,0}k_{1,0} + k_{2,0} + k_{0,0}k_{1,0}k_{2,0}$
64	11	0	$\{42\}$	$1 + k_{0,0} + k_{1,0}k_{2,0} + k_{0,0}k_{1,0}k_{2,0}$
64	12	0	$\{20\}$	$1 + k_{0,0}k_{1,0} + k_{0,0}k_{2,0} + k_{1,0}k_{2,0}$

Table 4.1: Simple relations on three key bits for the Klimov-Shamir cipher. The table shows some striking relations, which depend on three key bits $(k_{0,0}, k_{1,0}, k_{2,0})$, for the Klimov-Shamir self-synchronizing stream cipher.

A more detailed analysis of the functions $\Gamma^U_{i,j}[\omega](W, \mathsf{K_e})$ for different values of i, j and U reveals that many of these functions depend on only few bits of their $(i\omega - l)$-bit and 8ω-bit arguments. Let t_W and $t_{\mathsf{K_e}}$ respectively denote the number of bits of W and $\mathsf{K_e}$ which Γ effectively depends on. In addition, let t_K out of $t_{\mathsf{K_e}}$ bits come from K and the remaining $t_{\mathsf{S}^\star} = t_{\mathsf{K_e}} - t_K$ bits from $\mathsf{S}^\star$ (remember $\mathsf{K_e} = (K, \mathsf{S}^\star)$). Table 4.2 shows these values for some of these functions.

Having in mind what we mentioned in section 4.2 and being too optimistic, we give the following proposition.

Proposition 1. *If a function $\Gamma^U_{i,j}$ is random-looking enough, recovering the $t_{\mathsf{K_e}}$ unknown bits of the extended key takes expected time $2 \times i \times 2^{l+t_{\mathsf{K_e}}}$.*

The unity of time is processing one ciphertext word of the underlined SSSC. The factors 2, 2^l and i come from the following facts. On average two query evaluations are required to reject a wrong guess for the involved $t_{\mathsf{K_e}}$ unknown extended key bits. Computing Γ from $\mathcal{F}_i$ needs 2^l evaluations of $\mathcal{F}_i$ (remember $\Gamma^U_{i,j}(W, \mathsf{K_e}) = \bigoplus_{U \in \mathbb{B}^l} \mathcal{F}_{i,j}((U; W), \mathsf{K_e})$). Finally, computing $\mathcal{F}_i$ needs i iterations of the cipher.

Even if the ideal condition of Proposition 1 is not satisfied, the only thing which is not guaranteed is that the $t_{\mathsf{K_e}}$ involved unknown bits are uniquely determined. Yet some information about them can be achieved. Refer to the sections 4.2 and 3.2 regarding the equivalence classes.

ω	i	j	U	$t_{\mathsf{K_e}}$	t_W	t_K	$t_{\mathsf{S}^\star}$	comment
8	1	0	$\emptyset$	20	8	9	11	
16	1	0	$\emptyset$	40	16	17	23	
32	1	0	$\emptyset$	80	32	33	47	
64	1	0	$\emptyset$	160	64	65	95	
8	1	0	$\{1\}$	9	5	3	6	
16	1	0	$\{u\}$	18	11	6	12	$8 \leq u \leq 10$
32	1	0	$\{u\}$	42	23	14	28	$16 \leq u \leq 20$
64	1	0	$\{u\}$	90	51	30	60	$32 \leq u \leq 42$
8	3	0	$\{8\}$	4	6	4	0	
8	3	0	$\{18\}$	5	7	5	0	see Example 11
16	7	0	$\{16\}$	17	58	17	0	
16	7	0	$\{34\}$	16	52	16	0	see Example 14
16	7	0	$\{33, 34\}$	12	33	12	0	see Example 12
16	7	0	$\{38, 39\}$	12	30	12	0	see Example 13
32	15	0	$\{32\}$	41	293	41	0	
32	15	0	$\{66\}$	40	279	40	0	
32	15	0	$\{76, 77\}$	36	231	36	0	
64	31	0	$\{64\}$	89	1274	89	0	
64	31	0	$\{130\}$	88	1243	88	0	
64	31	0	$\{129, 130\}$	84	1158	84	0	see Proposition 2
64	31	0	$\{150, 151\}$	84	1155	84	0	see Proposition 2

Table 4.2: Overview of the dependency of the superpolys in their input arguments for the Klimov-Shamir self-synchronizing stream cipher. The table shows the effective number of bits of each argument which the superpoly $\Gamma^U_{i,j}[\omega](W, \mathsf{K_e})$ depends on. The superpoly $\Gamma^U_{i,j}[\omega]$ effectively depends on t_W bits of W and $t_{\mathsf{K_e}}$ bits of $\mathsf{K_e}$. Moreover, t_K bits out of the $t_{\mathsf{K_e}}$ extended key bits are from the original secret key whereas the remaining $t_{\mathsf{S}^\star} = t_{\mathsf{K_e}} - t_K$ bits are due to the unknown internal state. Note that the functions having the same number of effective bits do not necessarily have the same involved variables.

Example 11. *Take the superpoly* $\Gamma_{3,0}^{\{18\}}[8](W, \mathsf{K_e})$ *from Table 4.2. This particular function depends on* $t_{\mathsf{K_e}} = 5$ *bits* $(k_{0,0}, k_{0,1}, k_{1,0}, k_{2,0}, k_{2,1})$ *of the key and on* $t_W = 7$ *bits* $(c_{1,4}, c_{1,5}, c_{1,6}, c_{2,0}, c_{2,1}, c_{2,5}, c_{2,6})$ *of the ciphertext. The ANF of this function is:*

$$\begin{aligned}\Gamma_{3,0}^{\{18\}}[8] \;=\; & 1 + k_{0,0}k_{0,1} + k_{0,0}k_{0,1}k_{2,0} + k_{2,0}k_{2,1} + k_{0,0}c_{1,4} + \\ & k_{0,0}k_{2,0}c_{1,4} + k_{0,0}k_{1,0}c_{1,5} + k_{0,0}k_{1,0}k_{2,0}c_{1,5} + \\ & k_{0,0}c_{1,6} + k_{0,0}k_{2,0}c_{1,6} + k_{2,0}c_{2,0} + k_{0,0}k_{2,0}c_{2,0} + \\ & k_{2,0}c_{2,1} + c_{2,0}c_{2,1} + k_{0,0}c_{2,0}c_{2,1} + k_{2,0}c_{2,0}c_{2,1} + \\ & k_{0,0}k_{2,0}c_{2,0}c_{2,1} + k_{1,0}k_{2,0}c_{2,5} + k_{2,0}c_{2,6}.\end{aligned} \tag{4.8}$$

This equation can be seen as a system of $2^{t_W} = 128$ *equations versus* $t_{\mathsf{K_e}} = 5$ *unknowns. Our analysis of this function shows that only* 48 *of the equations are independent which on average can give* 3.5 *bits of information about the five unknown bits (*4 *bits of information for* 75% *of the keys and* 2 *bits for the remaining* 25% *of the keys).*

Example 12. *Take the superpoly* $\Gamma_{7,0}^{\{33,34\}}[16](W, \mathsf{K_e})$ *from Table 4.2. This particular function depends on* $t_{\mathsf{K_e}} = 12$ *key bits and on* $t_W = 33$ *ciphertext bits. Our analysis of this function shows that on average about* 2.41 *bits of information on the 12 key bits can be achieved (*10 *bits of information for* 12.5% *of the keys,* 3 *bits for* 25% *of the keys and* 0.67 *bits about the remaining* 62.5% *of the keys).*

Example 13. *Take the superpoly* $\Gamma_{7,0}^{\{38,39\}}[16](W, \mathsf{K_e})$ *from Table 4.2. This particular function depends on* $t_{\mathsf{K_e}} = 12$ *key bits and on* $t_W = 30$ *ciphertext bits. Our analysis of this function shows that on average about* 1.94 *bits of information on the 12 key bits can be achieved (*10 *bits of information for* 12.5% *of the keys,* 3 *bits for another* 12.5% *of the keys and* 0.42 *bits for the remaining* 75% *of the keys).*

Example 14. *Take the superpoly* $\Gamma_{7,0}^{\{34\}}[16](W, \mathsf{K_e})$ *from Table 4.2. This particular function depends on* $t_{\mathsf{K_e}} = 16$ *key bits and on* $t_W = 52$ *ciphertext bits. Our analysis of this function shows that on average about* 5.625 *bits of information on the 16 key bits can be achieved (*13 *bits of information for* 25% *of the keys,* 11 *bits for* 12.5% *of the keys,* 4 *bits for another* 12.5% *of the keys, and* 1 *bit for the remaining* 50% *of the keys).*

For larger values of i we expect Γ to fit better the ideal situation of Proposition 1. Therefore, we give the following claim about the security of the 64-bit version of Klimov-Shamir's proposal.

Proposition 2. *We expect each of the functions* $\Gamma_{31,0}^{\{129,130\}}[64]$ *and* $\Gamma_{31,0}^{\{150,151\}}[64]$ *to reveal a large amount of information about the corresponding* $t_{\mathsf{K_e}} = 84$ *involved key bits for a non-negligible fraction of the keys. The required computational time is* $2 \times 31 \times 2^{2+84} \approx 2^{92}$.

In chapter 3, the sets of weak IV variables were mainly found by random search. How to find a set of weak public variables was raised as an open question in chapter 3. In the next section we present a systematic procedure to find a set of weak ciphertext variables, with the consequence of improving Proposition 2. More sophisticated algorithms for finding a set of weak public variables can be found in [100, 16].

4.6 Towards a systematic approach to find weak ciphertext variables

The idea is to start with a set (vector) U and extend it gradually. At each step, we examine all the ciphertext bits which Γ^U depends on. The goal is to determine an extended U for the next step that results in a Γ which depends on the least number of key bits. To this end, we choose a ciphertext bit (to be added to the current set U) which has this property. Table 4.3 shows our simulation results by starting from function $\Gamma_{1,0}^{\{41\}}[64]$ from Table 4.2 which effectively depends on $t_{\mathsf{K_e}} = 90$ extended key bits and $t_W = 51$ ciphertext bits. Similar to Proposition 2, one expects each of the functions $\Gamma_{1,0}^{U}[64]$ in Table 4.3 to reveal a large amount of information about the corresponding $t_{\mathsf{K_e}}$ involved extended key bits (including t_K effective key bits) for a non-negligible fraction of the keys; the time complexity is $2 \times 1 \times 2^{l+t_{\mathsf{K_e}}}$, as indicated in the last column. In particular, by starting from the function in the bottom of Table 4.3, (the promised large amount of information about) the involved $t_K = 12$ key bits and $t_{\mathsf{S}^\star} = 33$ internal state bits can be gained in time 2^{65} (for a non-negligible fraction of the keys). Notice, that once we have the correct value for the unknown extended key for some function in Table 4.3, those of the previous function can be recovered by little effort. Therefore, we present the following proposition.

Proposition 3. *We expect that by starting from* $\Gamma_{1,0}^{\{1-9,32-41\}}[64]$ *and going backwards to* $\Gamma_{1,0}^{\{41\}}[64]$ *as indicated in Table 4.3, a large amount of information about the involved* $t_{\mathsf{K_e}} = 90$ *unknown bits (including* $t_K = 30$ *effective key bits) is revealed for a non-negligible fraction of the keys in time* 2^{65} *.*

Remark 6. *By combining the results of different superpolys* Γ*, one can get better results. Finding an optimal combination demands patience and detailed examination of different* Γ*'s. We make this statement clearer by an example as follows. Detailed analysis of* $\Gamma_{31,0}^{\{129,130\}}[64]$ *and* $\Gamma_{31,0}^{\{150,151\}}[64]$ *shows that the key bits which they depend*

on are $\{0-27, 64-90, 128-156\}$ *and* $\{0-28, 64-90, 128-155\}$*, respectively. These two functions have respectively* 27 *and* 28 *bits in common with the* 30 *key bits* $\{0-19, 21-30\}$ *involved in* $\Gamma_{1,0}^{\{41\}}[64]$*. They also include the key bits* $\{0, 32, 64\}$ *for which* 1.75 *information can be easily gained according to Example 10. Taking it altogether, it can be said that a large amount of information about the* 88 *key bits* $\{0-30, 32, 64-90, 128-156\}$ *can be achieved in time* 2^{65} *with a non-negligible probability.*

4.7 Summary

In this work we proposed a new analysis method for self-synchronizing stream ciphers. We then applied it to the Klimov-Shamir's example of a construction of a T-function based self-synchronizing stream cipher. We did not fully break this proposal, but the strong key leakage demonstrated by our results makes us believe a total break is not out of reach. In future design of self-synchronizing stream ciphers, one has to take into account and counter potential key leakage.

U	$\mathsf{K_e}$	W	$t_{\mathsf{K_e}}$	t_W	t_K	$t_{\mathsf{S}^\star}$	Time
$\{41\}$	$\{0-19, 21-30, 384-414, 449-467, 469-478\}$	$\{0-9, 22-40, 42-63\}$	90	51	30	60	2^{92}
$\{9, 41\}$	$\{0-19, 21-29, 384-413, 449-467, 469-477\}$	$\{0-8, 22-40, 42-63\}$	87	50	29	58	2^{90}
$\{8, 9, 41\}$	$\{0-19, 21-28, 384-412, 449-467, 469-476\}$	$\{0-7, 22-40, 42-63\}$	84	49	28	56	2^{88}
$\{7-9, 41\}$	$\{0-19, 21-27, 384-411, 449-467, 469-475\}$	$\{0-6, 22-40, 42-63\}$	81	48	27	54	2^{86}
$\{6-9, 41\}$	$\{0-19, 21-26, 384-410, 449-467, 469-474\}$	$\{0-5, 22-40, 42-63\}$	78	47	26	52	2^{84}
$\vdots$							$\vdots$
$\{1-9, 41\}$	$\{0-19, 21, 384-405, 449-467, 469\}$	$\{0, 22-40, 42-63\}$	63	42	21	42	2^{74}
$\{1-9, 40, 41\}$	$\{0-18, 21, 384-405, 449-466, 469\}$	$\{0, 22-39, 42-63\}$	61	41	20	41	2^{73}
$\{1-9, 39-41\}$	$\{0-17, 21, 384-405, 449-465, 469\}$	$\{0, 22-38, 42-63\}$	59	40	19	40	2^{72}
$\vdots$							$\vdots$
$\{1-9, 34-41\}$	$\{0-12, 21, 384-405, 449-460, 469\}$	$\{0, 22-33, 42-63\}$	49	35	14	35	2^{67}
$\{1-9, 33-41\}$	$\{0-11, 21, 384-405, 449-459, 469\}$	$\{0, 22-32, 42-63\}$	47	34	13	34	2^{66}
$\{1-9, 32-41\}$	$\{0-10, 21, 384-405, 449-458, 469\}$	$\{0, 22-31, 42-63\}$	45	33	12	33	2^{65}

Table 4.3: Finding weak ciphertext variables in a systematic way for the Klimov-Shamir self-synchronizing stream cipher. The table shows how to start from an initial set of weak ciphertext variables U (first line) and gradually extend it (the following lines). The table shows the effective number of bits of each argument which the superpoly $\Gamma^U_{1,0}[64](W, \mathsf{K_e})$ depends on. The superpoly $\Gamma^U_{1,0}[64]$ effectively depends on t_W bits of W and $t_{\mathsf{K_e}}$ bits of $\mathsf{K_e}$. Moreover, t_K bits out of the $t_{\mathsf{K_e}}$ extended key bits are from the original secret key whereas the remaining $t_{\mathsf{S}^\star} = t_{\mathsf{K_e}} - t_K$ bits are due to the unknown internal state. The last column shows our estimated required time complexity for recovering the involved $t_{\mathsf{K_e}}$ extended key bits, including the t_K original key bits.

5

Linearization Framework for Finding hash Collisions

In the previous three chapters, we cryptanalyzed stream ciphers which are among keyed cryptographic primitives. In this chapter, however, we study hash functions which are often unkeyed. There is no secret key to be recovered in this case. Instead, a major goal in hash function cryptanalysis is to find a hash collision. In this chapter, an improved differential cryptanalysis framework for finding collisions in hash functions is provided. Its principle is based on linearization of hash functions in order to find low weight differential characteristics as initiated by Chabaud and Joux [75]. This is, however, formalized and refined in several ways: for the problem of finding conforming message pairs whose differential trail follows a linear trail, a condition function is introduced so that finding collisions is equivalent to finding preimages of the zero vector under the condition function. Then, the dependency table concept shows how much influence every input bit of the condition function has on each output bit. Careful analysis of the dependency table reveals degrees of freedom that can be exploited in accelerated preimage reconstruction under the condition function. It turns out that the degrees of freedom are highly related to the neutrality and probabilistic neutrality concepts, exploited to accelerate key recovery cryptanalysis on keyed primitives in the previous chapters. These concepts are applied to an in-depth collision analysis of reduced-round variants of two SHA-3 candidates CubeHash and MD6. This chapter is mainly due to the contributions of [65], published at ASIACRYPT 2009. We also include the result of [148], published at AFRICACRYPT 2010, yet largely based on [65].

5.1 Introduction

Hash functions are important cryptographic primitives that find applications in many areas including digital signatures, commitment schemes and authentication codes. To this end, hash functions are expected to possess several security properties, one of which is *collision resistance*. Informally, a hash function is collision resistant if it is *practically infeasible* to find two distinct messages that produce the same output.

The goal of this work is to *revisit* collision-finding methods using linearization of the hash function. This method was initiated by Chabaud and Joux on SHA–0 [75] and was later extended and applied to SHA–1 by Rijmen and Oswald [203]. In particular, in [203] it was observed that the codewords of a linear code, which are defined through a linearized version of the hash function, can be used to identify high probability differential paths. This method was later extended by Pramstaller *et al.* [195] with the general conclusion that finding high probability differential paths is related to low weight codewords of the attributed linear code. In this chapter we further investigate this issue.

The first contribution of our work is to present a more concrete and tangible relation between the linearization and differential paths. In the case that modular addition is the only involved nonlinear operation, our results can be stated as follows. Given the parity check matrix $\mathcal{H}$ of a linear code, and two matrices $\mathcal{A}$ and $\mathcal{B}$, find a codeword Δ such that $\mathcal{A}\Delta \vee \mathcal{B}\Delta$ is of low weight. This is clearly different from the problem of finding a low weight codeword Δ.

We then consider the problem of finding a conforming message pair for a given differential trail for a certain linear approximation of the hash function. The recent collision finding algorithms on hash functions [44, 45, 236, 238, 239, 237, 73, 179, 70, 168, 134] have investigated extensive methods to identify degrees of freedom to be used to efficiently find conforming message pairs by means of satisfying some *conditions*. These methods are referred to as *message modification* techniques which apparently have been used by Xiaoyun Wang as early as 1997 [233, 234]. However, they were brought to the attention of the international cryptographic community only in 2004 [235]; see also [236, 238, 239, 237]. Message modification techniques use concepts such as neutral bits [44], semi-neutral bits [167, 226] and tunnels [151]. When it comes to implementation, backtracking algorithms [38, 73] are used to find a conforming message.

Our second contribution in this chapter is to present a unified framework to exploit degrees of freedom and evaluate the complexity of the corresponding backtracking algorithm. Our framework, similar to the work by De Cannière and Rechberger [73], has the flexibility to be applied to a large number of differential paths to identify the best one. In particular, we show that the problem of finding conforming pairs can be reformulated as finding preimages of zero under a function which we call the Condition function. We then define the concept of *dependency table* which shows how much influence every input bit of the condition function has on each output bit. By carefully analyzing the dependency table, we are able to profit not only from neutral bits [44] but also from probabilistic neutral bits [15] in a backtracking search algorithm, similar to [73, 38, 194, 118]. This contributes to a better understanding of freedom degrees uses.

We consider hash functions working with n-bit words. In particular, we focus on those using modular addition of n-bit words as the only nonlinear operation. The incorporated linear operations are XOR, shift and rotation of n-bit words in practice. We present our framework in detail for these constructions by approximating modular addition with XOR. We demonstrate its validity by applying it on reduced-round variants of CubeHash [34] (one of the 14 second round NIST SHA-3 [181] competitors) which uses modular addition, XOR and rotation. CubeHash instances are parametrized by two parameters r and b and are denoted by `CubeHash`-r/b which process b message bytes per iteration; each iteration is composed of r rounds. Although we cannot break the original submission `CubeHash`-8/1 nor the current tweaked official proposal `CubeHash`-16/32, we provide real collisions for the much weaker variants `CubeHash`-3/64, `CubeHash`-4/48 and `CubeHash`-5/96. Interestingly, we show that neither the more secure variants `CubeHash`-6/16, `CubeHash`-7/64 and `CubeHash`-8/96 do provide the desired collision security by providing theoretical attacks with complexities $2^{219.9}$, $2^{203.0}$ and $2^{80.0}$ respectively; nor that `CubeHash`-6/4 with 512-bit digests is second-preimage resistant, as with probability 2^{-478} a second preimage can be produced by only one hash evaluation. Our theory can be easily generalized to arbitrary nonlinear operations. We discuss this issue and as an application we provide collision cryptanalyses on 16 rounds of MD6 [206]. MD6 was a first round SHA-3 candidate whose original number of rounds varies from 80 to 168 when the digest size ranges from 160 to 512 bits.

5.2 Linear differential cryptanalysis of hash functions

5.2.1 Attributing compression functions to hash functions

Hash functions transform a variable-length input to a fixed-size output, called message *digest*. In practice, hash functions are mostly built from a fixed input size *compression* function, *e.g.*, the Davies-Meyer [174] block cipher based construction; see 1.4.2. One then applies a *domain extension* method, such as the renowned Merkle-Damgård construction, to the compression function in order to construct a hash function that accepts messages of arbitrary length. To any hash function, no matter how it has been designed, we can always attribute fixed input size compression functions, such that a collision for a derived compression function results in a direct collision for the hash function itself. This way, firstly we are working with fixed input size compression functions rather than varying input size ones, secondly we can attribute compression functions to those hash functions which are not explicitly based on a fixed input size compression function, and thirdly we can derive different compression functions from a hash function. For example, multi-block collision cryptanalysis [238] benefits from the third point. Our task is to find two messages for an attributed compression function such that their digests are preferably equal (a collision) or differ in only a few bits (a near-collision). Collisions for a compression function are directly translated into collisions for the hash function provided that the initial value condition of the hash function is satisfied. The relevance of near collisions, however, depends on the hash function structure and the way the compression function has been defined. In most of the cases the near collisions of the compression function provide near collisions for the underlying hash function as well. But in some other cases, such as sponge constructions [39] with a strong filtering at the end or a Merkle-Damgård construction with a strong final transformation [166], they are of little interest.

5.2.2 Linearization of compression functions

Let's consider a compression function $H = \mathsf{Compress}(M, V)$ which works with n-bit words and maps an m-bit message M and a v-bit initial value V into an h-bit output H. Our aim is to find a collision for such compression functions with a randomly given initial value V. In this section we consider *modular-addition-based* Compress

functions, that is, they use only modulo 2^n additions in addition to linear transformations. This includes the family of AXR (Addition-XOR-Rotation) hash functions which are based on these three operations. In section 5.5 we generalize our framework to other family of compression functions. For these Compress functions, we are looking for two messages with a difference Δ that result in a collision. In particular, we are interested in a Δ for which two randomly chosen messages with this difference lead to a collision with a high probability for a randomly chosen initial value. For modular-addition-based Compress functions, we consider a linearized version for which all modular additions are replaced by XOR. This is a common linear approximation of modular addition. Other possible linear approximations of modular addition, which are less addressed in literature, can be considered according to our generalization of section 5.5. As modular addition was the only nonlinear operation, we now have a linear function which we call $\mathsf{Compress}_{\text{lin}}$. Since $\mathsf{Compress}_{\text{lin}}(M,V) \oplus \mathsf{Compress}_{\text{lin}}(M \oplus \Delta, V) = \mathsf{Compress}_{\text{lin}}(\Delta, 0)$ is independent of the value of V, we adopt the notation $\mathsf{Compress}_{\text{lin}}(M) = \mathsf{Compress}_{\text{lin}}(M, 0)$ instead. Let Δ be an element of the kernel of the linearized compression function, *i.e.*, $\mathsf{Compress}_{\text{lin}}(\Delta) = 0$. We are interested in the probability $\Pr\{\mathsf{Compress}(M,V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = 0\}$ for a random M and V. In the following we present an algorithm which computes this probability, called the *raw* (or *bulk*) *probability*.

5.2.3 Computing the raw probability

We consider a general n-bit vector $x = (x_0, \ldots, x_{n-1})$ as an n-bit integer denoted by the same variable, *i.e.*, $x = \sum_{i=0}^{n-1} x_i 2^i$. The Hamming weight of a binary vector or an integer x, $\mathrm{wt}(x)$, is the number of its nonzero elements, *i.e.*, $\mathrm{wt}(x) = \sum_{i=0}^{n-1} x_i$. We use $+$ for modular addition of words and $\oplus, \vee$ and $\wedge$ for bit-wise XOR, OR and AND logical operations between words as well as vectors. We use the following lemma which is a special case of the problem of computing $\Pr\{\big((A \oplus \alpha) + (B \oplus \beta)\big) \oplus (A + B) = \gamma\}$ where α, β and γ are constants and A and B are independent and uniform random variables, all of them being n-bit words. Lipmaa and Moriai have presented an efficient algorithm for computing this probability [162]. We are interested in the case $\gamma = \alpha \oplus \beta$ for which the desired probability has a simple closed form.

Lemma 2. $\Pr\{\big((A \oplus \alpha) + (B \oplus \beta)\big) \oplus (A + B) = \alpha \oplus \beta\} = 2^{-\mathrm{wt}\left((\alpha \vee \beta) \wedge (2^{n-1}-1)\right)}$.

Lemma 2 gives us the probability that modular addition behaves like the XOR operation. As $\mathsf{Compress}_{\mathrm{lin}}$ approximates $\mathsf{Compress}$ by replacing modular addition with XOR, we can then devise a simple algorithm to compute (estimate) the raw probability $\Pr\{\mathsf{Compress}(M,V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = \mathsf{Compress}_{\mathrm{lin}}(\Delta)\}$. Let's first introduce some notation.

Notation. Let n_{add} denote the number of modular additions which $\mathsf{Compress}$ uses in total. In the course of evaluation of $\mathsf{Compress}(M,V)$, let the two addends of the ith modular addition ($1 \leq i \leq n_{\mathrm{add}}$) be denoted by $A^i(M,V)$ and $B^i(M,V)$, for which the ordering is not important. The value $C^i(M,V) = \big(A^i(M,V) + B^i(M,V)\big) \oplus A^i(M,V) \oplus B^i(M,V)$ is then called the carry word of the ith modular addition. Similarly, in the course of evaluation of $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$, denote the two inputs of the ith linearized modular addition by $\alpha^i(\Delta)$ and $\beta^i(\Delta)$ in which the ordering is the same as that for A^i and B^i. We define five more functions $\mathbf{A}(M,V)$, $\mathbf{B}(M,V)$, $\mathbf{C}(M,V)$, $\boldsymbol{\alpha}(\Delta)$ and $\boldsymbol{\beta}(\Delta)$ with $(n-1)n_{\mathrm{add}}$-bit outputs. These functions are respectively defined as the concatenation of the first $(n-1)$ bits of the words $A^i(M,V), B^i(M,V), C^i(M,V), \alpha^i(M,V)$ and $\beta^i(M,V)$, $1 \leq i \leq n_{\mathrm{add}}$. For example, $\mathbf{A}(M,V)$ and $\boldsymbol{\alpha}(\Delta)$ are respectively the concatenation of the n_{add} words $\big(A^1(M,V), \ldots, A^{n_{\mathrm{add}}}(M,V)\big)$ and $\big(\alpha^1(\Delta), \ldots, \alpha^{n_{\mathrm{add}}}(\Delta)\big)$ where the MSBs of the words are excluded. To be more precise, we have $\mathbf{A}(M,V) = \big(A^1(M,V) \mod 2^{n-1}, \ldots, A^{n_{\mathrm{add}}}(M,V) \mod 2^{n-1}\big)$. Similar relations can be written for the other four functions.

Table 5.1 is a reference for all the main symbols used in sections 5.2, 5.3.1 and 5.5. Using this notation, the raw probability can be simply estimated as follows.

Lemma 3. *Let* $\mathsf{Compress}$ *be a modular-addition-based compression function. Then for any message difference* Δ *and for random values* M *and* V*,* $p_\Delta = 2^{-\mathrm{wt}\left(\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta)\right)}$ *is a lower bound for* $\Pr\{\mathsf{Compress}(M,V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = \mathsf{Compress}_{\mathrm{lin}}(\Delta)\}$.

Proof. We start with the following definition.

Definition 7. *We say that a message* M *(for a given* V*) conforms to (or follows) the trail of* Δ *iff* [1]

$$\big((A^i \oplus \alpha^i) + (B^i \oplus \beta^i)\big) \oplus (A^i + B^i) = \alpha^i \oplus \beta^i, \text{ for } 1 \leq i \leq n_{\mathrm{add}}, \tag{5.1}$$

where A^i*,* B^i*,* α^i *and* β^i *are shortened forms for* $A^i(M,V)$*,* $B^i(M,V)$*,* $\alpha^i(\Delta)$ *and* $\beta^i(\Delta)$*, respectively.*

[1] if and only if

symbol	description
$H = \mathsf{Compress}(M, V)$	a modular-addition-based or binary-FSM-based compression function from $\{0,1\}^m \times \{0,1\}^v$ to $\{0,1\}^h$
$\mathsf{Compress}_{\mathrm{lin}}(\Delta, 0)$ or $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$	the linearized version of Compress
$\mathsf{Condition}_{\Delta}(M, V)$ or $\mathsf{Condition}(M, V)$	condition function associated with a differential trail Δ for a compression function Compress and its linearized version $\mathsf{Compress}_{\mathrm{lin}}$
Y	the output of the condition function, $Y = \mathsf{Condition}_{\Delta}(M, V)$
Y_j	the jth condition bit
y	the total number of condition bits, *i.e.*, length of Y
n_{add}	the total number of modular additions of a modular-addition-based Compress function
n_{nl}	the total number of NUBFs of a binary-FSM-based Compress function
(A^i, B^i) or $\big(A^i(M, V), B^i(M, V)\big)$	the two addend words of the ith modular addition in a modular-addition-based Compress function
C^i or $C^i(M, V)$	the carry word of the ith modular addition in a modular-addition-based Compress function
(α^i, β^i) or $\big(\alpha^i(\Delta), \beta^i(\Delta)\big)$	the two input words of the ith linearized modular addition (*i.e.*, XOR) in a modular-addition-based linearized compression function $\mathsf{Compress}_{\mathrm{lin}}$
$\mathbf{A}(M, V)$	the concatenation of all n_{add} addend words $A^1, \ldots, A^{n_{\mathrm{add}}}$ excluding their MSBs
$\mathbf{B}(M, V)$	the concatenation of all n_{add} addend words $B^1, \ldots, B^{n_{\mathrm{add}}}$ excluding their MSBs
$\mathbf{C}(M, V)$	the concatenation of all n_{add} carry words $C^1, \ldots, C^{n_{\mathrm{add}}}$ excluding their MSBs
$\boldsymbol{\alpha}(\Delta)$	the concatenation of all n_{add} words $\alpha^1, \ldots, \alpha^{n_{\mathrm{add}}}$ excluding their MSBs
$\boldsymbol{\beta}(\Delta)$	the concatenation of all n_{add} words $\beta^1, \ldots, \beta^{n_{\mathrm{add}}}$ excluding their MSBs
$\boldsymbol{\alpha}_k, \boldsymbol{\beta}_k, \mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k$	the kth bit of the vectors $\boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{A}, \mathbf{B}, \mathbf{C}$
g^k	the kth NUBF of a binary-FSM-based Compress function
g^k_{lin}	the linear approximation of g^k as the one in $\mathsf{Compress}_{\mathrm{lin}}$ for a binary-FSM-based Compress function
x^k	the input vector of the kth NUBF of a binary-FSM-based compression function (*i.e.*, g^k) which computes $\mathsf{Compress}(M, V)$
δ^k	the input vector of the kth linearized NUBF of a linearized binary-FSM-based compression function (*i.e.*, g^k_{lin}) which computes $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$
$\Lambda(M, V)$	the output of all the n_{nl} NUBFs of a binary-FSM-based Compress function
$\Phi(\Delta)$	the output of all the n_{nl} linearized NUBFs of a linearized binary-FSM-based compression function $\mathsf{Compress}_{\mathrm{lin}}$
$\Lambda^{\Delta}(M, V)$	see section 5.5
$\Gamma(\Delta)$	see section 5.5
$\Lambda_k(M, V)$	the kth bit of the vector $\Lambda(M, V)$ which equals $g^k(x^k)$
$\Phi_k(\Delta)$	the kth bit of the vector $\Phi(\Delta)$ which equals $g^k_{\mathrm{lin}}(\delta^k)$
$\Lambda^{\Delta}_k(M, V)$	the kth bit of the vector $\Lambda^{\Delta}(M, V)$ which equals $g^k(x^k \oplus \delta^k)$
$\Gamma_k(\Delta)$	the kth bit of the vector $\Gamma(\Delta)$; $\Gamma_k(\Delta) = 1$ iff $\delta^k \neq 0$

Table 5.1: Main symbols used in this chapter. This table includes the main symbols used in sections 5.2, 5.3.1 and 5.5 along with their description.

It is not difficult to prove that under some reasonable independence assumptions p_Δ, which we call conforming probability, is the probability that a random message M follows the trail of Δ. This is a direct corollary of Lemma 2 and Definition 7. The exact proof can be done by induction on n_{add}, the number of modular additions in the compression function. Due to other possible non-conforming pairs that start from message difference Δ and lead to output difference $\mathsf{Compress}_{\text{lin}}(\Delta)$, p_Δ is a lower bound for the desired probability in the lemma. □

If $\mathsf{Compress}_{\text{lin}}(\Delta)$ is of low Hamming weight, a conforming message pair leads to a near collision in the output. The interesting Δ's for collision search are those which belong to the kernel of $\mathsf{Compress}_{\text{lin}}$, *i.e.*, those that satisfy $\mathsf{Compress}_{\text{lin}}(\Delta) = 0$. From now on, we assume that $\Delta \neq 0$ is in the kernel of $\mathsf{Compress}_{\text{lin}}$, hence looking for collisions. According to Lemma 3, one needs to try around $1/p_\Delta$ random message pairs in order to find a collision which conforms to the trail of Δ. However, in a random search it is better not to restrict oneself to the conforming messages as a collision at the end is all we want. Since p_Δ is a lower bound for the probability of getting a collision for a message pair with difference Δ, we might get a collision sooner. In section 5.3 we explain a method which might find a *conforming* message by avoiding random search.

5.2.4 Link with coding theory

We would like to conclude this section with a note on the relation between the following two problems:

I) finding low-weight codewords of a linear code,

II) finding a high probability linear differential path.

Since the functions $\mathsf{Compress}_{\text{lin}}(\Delta)$, $\boldsymbol{\alpha}(\Delta)$ and $\boldsymbol{\beta}(\Delta)$ are linear, we consider Δ as a column vector and attribute three matrices $\mathcal{H}$, $\mathcal{A}$ and $\mathcal{B}$ to these three transformations, respectively. In other words we have $\mathsf{Compress}_{\text{lin}}(\Delta) = \mathcal{H}\Delta$, $\boldsymbol{\alpha}(\Delta) = \mathcal{A}\Delta$ and $\boldsymbol{\beta}(\Delta) = \mathcal{B}\Delta$. We then call $\mathcal{H}$ the *parity check* matrix of the compression function.

Based on an initial work by Chabaud and Joux [75], the link between these two problems has been discussed by Rijmen and Oswald in [203] and by Pramstaller *et al.* in [195] with the general conclusion that finding highly probable differential paths is related to low weight codewords of the attributed linear code. In fact the relation

between these two problems is more delicate. For problem (I), we are provided with the parity check matrix $\mathcal{H}$ of a linear code for which a codeword Δ satisfies the relation $\mathcal{H}\Delta = 0$. Then, we are supposed to find a low-weight nonzero codeword Δ. This problem is known to be NP-hard but there are some non-optimal heuristic approaches for it, see [74] for example. For problem (II), however, we are given three matrices $\mathcal{H}$, $\mathcal{A}$ and $\mathcal{B}$ and need to find a nonzero Δ such that $\mathcal{H}\Delta = 0$ and $\mathcal{A}\Delta \vee \mathcal{B}\Delta$ is of low-weight, see Lemma 3. Nevertheless, low-weight codewords Δ's of the parity check matrix $\mathcal{H}$ might be good candidates for providing low-weight $\mathcal{A}\Delta \vee \mathcal{B}\Delta$, *i.e.*, differential paths with high probability p_Δ. In particular, this approach is promising if these three matrices are sparse.

5.3 Finding a conforming message pair efficiently

The methods that are used to accelerate the finding of a message which satisfies some requirements are referred to as *freedom degrees use* in the literature. This includes message modifications [238], neutral bits [44], semi-neutral bits [167, 226], tunnels [151], submarine modifications [179] and boomerang attacks [140, 168]. In this section we show that the problem of finding conforming message pairs can be reformulated as finding preimages of zero under a function which we call the *condition* function. One can carefully analyze the condition function to see how freedom degrees might be used in efficient preimage reconstruction. Our method is based on measuring the amount of influence which every input bit has on each output bit of the condition function. We introduce the dependency tables to distinguish the influential bits, from those which have no influence or are less influential. In other words, in case the condition function does not mix its input bits well, we profit not only from neutral bits [44] but also from probabilistic neutral bits [15]. This is achieved by devising a backtracking search algorithm, similar to [73, 38, 194, 118], based on the dependency table.

5.3.1 Condition function

Let's assume that we have a differential path for the message difference Δ which holds with probability $p_\Delta = 2^{-y}$. According to Lemma 3 we have $y = \mathrm{wt}\big(\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta)\big)$. In this section we show that, given an initial value V, the problem of finding a conforming message pair such that $\mathsf{Compress}(M, V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = 0$ can be translated

into finding a message M such that $\mathsf{Condition}_\Delta(M,V)=0$. Here $Y=\mathsf{Condition}_\Delta(M,V)$ is a function which maps m-bit message M and v-bit initial value V into y-bit output Y. In other words, the problem is reduced to finding a preimage of zero under the $\mathsf{Condition}_\Delta$ function. As we will see it is quite probable that not every output bit of the Condition function depends on all the message input bits. By taking a good strategy, this property enables us to find the preimages under this function more efficiently than random search. But of course, we are only interested in preimages of zero. In order to explain how we derive the function Condition from Compress, we first present a quite easy-to-prove lemma. We recall that the *carry word* of two words A and B is defined as $C=(A+B)\oplus A\oplus B$.

Lemma 4. *Let A and B be two n-bit words and C represent their carry word. Let $\delta=2^i$ for $0\leq i\leq n-2$. Then,*

$$\big((A\oplus\delta)+(B\oplus\delta)\big)=(A+B)\Leftrightarrow A_i\oplus B_i\oplus 1=0\ , \tag{5.2}$$

$$\big(A+(B\oplus\delta)\big)=(A+B)\oplus\delta\Leftrightarrow A_i\oplus C_i=0\ , \tag{5.3}$$

and similarly

$$\big((A\oplus\delta)+B\big)=(A+B)\oplus\delta\Leftrightarrow B_i\oplus C_i=0\ . \tag{5.4}$$

For a given difference Δ, a message M and an initial value V, let $\mathbf{A}_k$, $\mathbf{B}_k$, $\mathbf{C}_k$, $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$, $0\leq k<(n-1)n_{\text{add}}$, respectively denote the kth bit of the output vectors of the functions $\mathbf{A}(M,V)$, $\mathbf{B}(M,V)$, $\mathbf{C}(M,V)$, $\boldsymbol{\alpha}(\Delta)$ and $\boldsymbol{\beta}(\Delta)$, as defined in section 5.2.3. Let $\{i_0,\ldots,i_{y-1}\}$, $0\leq i_0<i_1<\cdots<i_{y-1}<(n-1)n_{\text{add}}$ be the positions of 1's in the vector $\boldsymbol{\alpha}(\Delta)\vee\boldsymbol{\beta}(\Delta)$. We define the function $Y=\mathsf{Condition}_\Delta(M,V)$ as:

$$Y_j=\begin{cases}\mathbf{A}_{i_j}\oplus\mathbf{B}_{i_j}\oplus 1 & \text{if}\quad(\boldsymbol{\alpha}_{i_j},\boldsymbol{\beta}_{i_j})=(1,1),\\ \mathbf{A}_{i_j}\oplus\mathbf{C}_{i_j} & \text{if}\quad(\boldsymbol{\alpha}_{i_j},\boldsymbol{\beta}_{i_j})=(0,1),\\ \mathbf{B}_{i_j}\oplus\mathbf{C}_{i_j} & \text{if}\quad(\boldsymbol{\alpha}_{i_j},\boldsymbol{\beta}_{i_j})=(1,0),\end{cases} \tag{5.5}$$

for $j=0,1,\ldots,y-1$. We sill later see that this equation can be equivalently written as equation (5.8).

Proposition 4. *For a given V and Δ, a message M conforms to the trail of Δ iff* $\mathsf{Condition}_\Delta(M,V)=0$.

Proof. The proof is straightforward from Definition 7, Lemma 4 and the definition of the Condition function in equation (5.5). □

5.3.2 Dependency table for freedom degrees use

For simplicity and generality, let's adopt the notation $F(M,V) = \mathsf{Condition}_{\Delta}(M,V)$ in this section. Assume that we are given a general function $Y = F(M,V)$ which maps m message bits and v initial value bits into y output bits. Our goal is to reconstruct preimages of a particular output (*e.g.*, the zero vector) efficiently. More precisely, we want to find V and M such that $F(M,V) = 0$. If F mixes its input bits very well, one needs to try about 2^y random inputs in order to find one mapping to zero. However, in some special cases due to neutrality [44] or semi-neutrality [151, 167, 226] of some input bits, not every input bit of F affects every output bit. Consider an ideal situation where message bits and output bits can be divided into ℓ and $\ell+1$ disjoint subsets respectively as $\bigcup_{i=1}^{\ell} \mathcal{M}_i$ and $\bigcup_{i=0}^{\ell} \mathcal{Y}_i$ such that the output bits $\mathcal{Y}_j$ $(0 \le j \le \ell)$ only depend on the input bits $\bigcup_{i=1}^{j} \mathcal{M}_i$ and the initial value V. In other words, once we know the initial value V, we can determine the output part $\mathcal{Y}_0$. If we know the initial value V and the input portion $\mathcal{M}_1$, the output part $\mathcal{Y}_1$ is then known and so on. Refer to Tables A.1 and A.2 to see the partitioning of condition functions related to CubeHash and MD6. This property of F suggests Algorithm 3 for finding a preimage of zero. Algorithm 3 is a backtracking search algorithm in essence, similar to [73, 38, 194, 118], and in practice is implemented recursively with a tree-based search to avoid memory requirements. The values $q_0, q_1, \ldots, q_\ell$ are the parameters of the algorithm to be determined later. To discuss the complexity of the algorithm, let $|\mathcal{M}_i|$ and $|\mathcal{Y}_i|$ denote the cardinality of $\mathcal{M}_i$ and $\mathcal{Y}_i$ respectively, where $|\mathcal{Y}_0| \ge 0$ and $|\mathcal{Y}_i| \ge 1$ for $1 \le i \le \ell$. We consider an *ideal behavior* of F for which each output part depends in a complex way on all the variables that it depends on. Thus, the output segment changes independently and uniformly at random if we change any part of the relevant input bits.

To analyze the algorithm, we need to compute the optimal values for $q_0, \ldots, q_\ell$. The time complexity of the algorithm is $\sum_{i=0}^{\ell} 2^{q_i}$ as at each step 2^{q_i} values are examined. The algorithm is successful if we have at least one candidate left at the end, *i.e.*, $q'_{\ell+1} \ge 0$. We have $q'_{i+1} \approx q_i - |\mathcal{Y}_i|$, coming from the fact that at the ith step 2^{q_i} values are examined each of which makes the portion $\mathcal{Y}_i$ of the output null with probability $2^{-|\mathcal{Y}_i|}$. Note that we have the restrictions $q_i - q'_i \le |\mathcal{M}_i|$ and $0 \le q'_i$ since we have $|\mathcal{M}_i|$ bits of freedom degree at the ith step and we require at least one surviving candidate

Algorithm 3 : Preimage finding

Inputs: Function F with the corresponding message partitions $\mathcal{M}_1, \ldots, \mathcal{M}_\ell$ and output partitions $\mathcal{Y}_0, \ldots, \mathcal{Y}_\ell$, and the (optimised) parameters $q_0, q_1, \ldots, q_\ell$ for the algorithm.

Output: some preimage of zero under F.

0: Choose 2^{q_0} initial values at random and keep those $2^{q'_1}$ candidates which make $\mathcal{Y}_0$ part null.

1: For each candidate, choose $2^{q_1 - q'_1}$ values for $\mathcal{M}_1$ and keep those $2^{q'_2}$ ones making $\mathcal{Y}_1$ null.

2: For each candidate, choose $2^{q_2 - q'_2}$ values for $\mathcal{M}_2$ and keep those $2^{q'_3}$ ones making $\mathcal{Y}_2$ null.

$\vdots$

i: For each candidate, choose $2^{q_i - q'_i}$ values for $\mathcal{M}_i$ and keep those $2^{q'_{i+1}}$ ones making $\mathcal{Y}_i$ null.

$\vdots$

ℓ: For each candidate, choose $2^{q_\ell - q'_\ell}$ values for $\mathcal{M}_\ell$ and keep those $2^{q'_{\ell+1}}$ final candidates making $\mathcal{Y}_\ell$ null.

after each step. Hence, the optimal values for q_i's can be recursively computed as $q_{i-1} = |\mathcal{Y}_{i-1}| + \max(0, q_i - |\mathcal{M}_i|)$ for $i = \ell, \ell - 1, \ldots, 1$ with $q_\ell = |\mathcal{Y}_\ell|$.

How can we determine the partitions $\mathcal{M}_i$ and $\mathcal{Y}_i$ for a given function F? Based on the idea of *probabilistic neutrality* and *neutrality measure* introduced in chapter 2, we propose the following heuristic method for determining the message and output partitions in practice. We first construct a $y \times m$ binary valued table T called *dependency table*. The entry $T_{j,i}$, $0 \leq i \leq m-1$ and $0 \leq j \leq y-1$, is set to one iff the jth output bit is highly affected by the ith message bit. To this end, we empirically measure the probability that changing the ith message bit changes the jth output bit. The probability is computed over random initial values and messages. We then set $T_{j,i}$ to one iff this probability is greater than a threshold $0 \leq th < 0.5$, *e.g.*, $th \approx 0.3$. We then call Algorithm 4.

In practice, once we make a partitioning for a given function using the above method, there are two issues which may cause the ideal behavior assumption to be violated:

1. The message segments $\mathcal{M}_1, \ldots, \mathcal{M}_i$ do not have full influence on $\mathcal{Y}_i$,
2. The message segments $\mathcal{M}_{i+1}, \ldots, \mathcal{M}_\ell$ have influence on $\mathcal{Y}_0, \ldots, \mathcal{Y}_i$.

Algorithm 4 : Message and output partitioning

Input: Dependency table T.

Outputs: ℓ, message partitions $\mathcal{M}_1, \ldots, \mathcal{M}_\ell$, and output partitions $\mathcal{Y}_0, \ldots, \mathcal{Y}_\ell$.

1: Put all the output bits j in $\mathcal{Y}_0$ for which the row j of T is all-zero.
2: Delete all the all-zero rows from T.
3: $\ell := 0$;
4: **while** T is not empty **do**
5: $\quad \ell := \ell + 1$;
6: $\quad$ **repeat**
7: $\quad\quad$ Determine the column i in T which has the highest number of 1's and delete it from T.
8: $\quad\quad$ Put the message bit which corresponds to the deleted column i into the set $\mathcal{M}_\ell$.
9: $\quad$ **until** There is at least one all-zero row in T OR T becomes empty
10: $\quad$ If T is empty set $\mathcal{Y}_\ell$ to those output bits which are not in $\bigcup_{i=0}^{\ell-1} \mathcal{Y}_i$ and stop.
11: $\quad$ Put all the output bits j in $\mathcal{Y}_\ell$ for which the corresponding row of T is all-zero.
12: $\quad$ Delete all the all-zero rows from T.
13: **end while**

With regard to the first issue, we ideally would like that all the message segments $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_i$ as well as the initial value V have full influence on the output part $\mathcal{Y}_i$. In practice the effect of the last few message segments $\mathcal{M}_{i-d_i}, \ldots, \mathcal{M}_i$ (for some small integer d_i) is more important, though. Theoretical analysis of deviation from this requirement may not be easy. However, with some tweaks on the tree-based (backtracking) search algorithm, we may overcome this effect in practice. For example, if the message segment $\mathcal{M}_{i-1}$ does not have a great influence on the output segment $\mathcal{Y}_i$, we may decide to backtrack two steps at depth i, instead of one (the default value). The reason is as follows. Imagine that you are at depth i of the tree and you are trying to adjust the ith message segment $\mathcal{M}_i$, to make the output segment $\mathcal{Y}_i$ null. If, after trying about $2^{\min(|\mathcal{M}_i|,|\mathcal{Y}_i|)}$ choices for the ith message block, you do not find an appropriate one, you will go one step backward and choose another choice for the $(i-1)$-st message segment $\mathcal{M}_{i-1}$; you will then go one step forward once you have successfully adjusted the $(i-1)$-st message segment. If $\mathcal{M}_{i-1}$ has no effect on $\mathcal{Y}_i$, this would be useless and increase our search cost at this node. Hence it would be appropriate if we backtrack two steps at this depth. In general, we may tweak our tree-based search by setting the number of steps which we want to backtrack at each depth.

In contrast, the theoretical analysis of the second issue is easy. Ideally, we would like that the message segments $\mathcal{M}_i, \ldots, \mathcal{M}_\ell$ have no influence on the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$. The smaller the threshold value th is chosen, the less the influence would be. Let 2^{-p_i}, $1 \leq i \leq \ell$, denote the probability that changing the message segment $\mathcal{M}_i$ does not change any bit from the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$. The probability is computed over random initial values and messages, and a random non-zero difference in the message segment $\mathcal{M}_i$. Algorithm 3 must be reanalyzed in order to recompute the optimal values for $q_0, \ldots, q_\ell$. Algorithm 3 also needs to be slightly changed by reassuring that at step i, all the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$ remain null. The time complexity of the algorithm is still $\sum_{i=0}^{\ell} 2^{q_i}$ and it is successful if at least one surviving candidate is left at the end, *i.e.*, $q_{\ell+1} \geq 0$. However, here we set $q'_{i+1} \approx q_i - |\mathcal{Y}_i| - p_i$. This comes from the fact that at the ith step 2^{q_i} values are examined each of which makes the portion $\mathcal{Y}_i$ of the output null with probability $2^{-|\mathcal{Y}_i|}$ and keeping the previously set output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$ null with probability 2^{-p_i} (we assume these two events are independent). Here, our restrictions are again $0 \leq q'_i$ and $q_i - q'_i \leq |\mathcal{M}_i|$. Hence, the optimal values

for q_i's can be recursively computed as $q_{i-1} = p_{i-1} + |\mathcal{Y}_{i-1}| + \max(0, q_i - |\mathcal{M}_i|)$ for $i = \ell, \ell - 1, \ldots, 1$ with $q_\ell = |\mathcal{Y}_\ell|$.

Remark 7. *When working with functions with a huge number of input bits, it might be appropriate to consider the m-bit message M as a string of u-bit units instead of bits. For example, one can take $u = 8$ and work with bytes. We then use the notation $M = (M[0], \ldots, M[m/u - 1])$ (assuming u divides m) where $M[i] = (M_{iu}, \ldots, M_{iu+u-1})$. In this case the dependency table must be constructed according to the probability that changing every message unit changes each output bit.*

5.4 Application to CubeHash

CubeHash [34] is Bernstein's proposal for the NIST SHA-3 competition [181]. CubeHash variants, denoted by `CubeHash`-r/b, are parametrized by r and b which at each iteration process b bytes in r rounds. Although `CubeHash`-8/1 was the original official submission, later the designer proposed the tweak `CubeHash`-16/32 which is almost 16 times faster than the initial proposal [36]. Nevertheless, the author has encouraged cryptanalysis of `CubeHash`-r/b variants for smaller r's and bigger b's.

5.4.1 CubeHash description

CubeHash works with 32-bit words ($n = 32$) and uses three simple operations: XOR, rotation and modular addition. It has an internal state $S = (S_0, S_1, \ldots, S_{31})$ of 32 words and its variants, denoted by `CubeHash`-r/b, are identified by two parameters $r \in \{1, 2, \ldots\}$ and $b \in \{1, 2, \ldots, 128\}$. The internal state S is set to a specified value which depends on the digest length (limited to 512 bits) and parameters r and b. The message to be hashed is appropriately padded and divided into b-byte message blocks. At each iteration one message block is processed as follows. The 32-word internal state S is considered as a 128-byte value and the message block is XORed into the first b bytes of the internal state.[1]Then, the following fixed permutation is applied r times to the internal state to prepare it for the next iteration.

[1] The first message byte into the least significant byte of S_0, the second one into the second least significant byte of S_0, the third one into the third least significant byte of S_0, the fourth one into the most significant byte of S_0, the fifth one into the least significant byte of S_1, and so forth until all b message bytes have been exhausted.

1. Add S_i into $S_{i\oplus 16}$, for $0 \leq i \leq 15$.
2. Rotate S_i to the left by seven bits, for $0 \leq i \leq 15$.
3. Swap S_i and $S_{i\oplus 8}$, for $0 \leq i \leq 7$.
4. XOR $S_{i\oplus 16}$ into S_i, for $0 \leq i \leq 15$.
5. Swap S_i and $S_{i\oplus 2}$, for $i \in \{16, 17, 20, 21, 24, 25, 28, 29\}$.
6. Add S_i into $S_{i\oplus 16}$, for $0 \leq i \leq 15$.
7. Rotate S_i to the left by eleven bits, for $0 \leq i \leq 15$.
8. Swap S_i and $S_{i\oplus 4}$, for $i \in \{0, 1, 2, 3, 8, 9, 10, 11\}$.
9. XOR $S_{i\oplus 16}$ into S_i, for $0 \leq i \leq 15$.
10. Swap S_i and $S_{i\oplus 1}$, for $i \in \{16, 18, 20, 22, 24, 26, 28, 30\}$.

Having processed all message blocks, a fixed transformation is applied to the final internal state to extract the hash value as follows. First, the last state word S_{31} is ORed with integer 1 and then the above permutation is applied $10 \times r$ times to the resulting internal state. Finally, the internal state is truncated to produce the message digest of desired hash length. Refer to [34] for the full specification.

Remark 8. *For* `CubeHash-`r`/`b *there is a generic collision cryptanalysis with complexity of about* 2^{512-4b}*. For* $b > 64$ *this is faster than the generic birthday attack on hash functions with digest length of 512 bits. Specifically, for* $b = 96$ *the generic attack has a complexity of about* 2^{128}*.*

5.4.2 Defining the compression function

To be in the line of our general method, we need to deal with fixed-size input compression functions. To this end, we consider t ($t \geq 1$) consecutive iterations of CubeHash. We define the function $H = \mathsf{Compress}(M, V)$ with an $8bt$-bit message $M = M^0||\ldots||M^{t-1}$, a 1024-bit initial value V and a $(1024 - 8b)$-bit output H. The initial value V is used to initialize the 32-word internal state of CubeHash. Each M^i is a b-byte message block. We start from the initialized internal state and update it in t iterations. That is, in t iterations the t message blocks $M^0, \ldots, M^{t-1}$ are sequentially

processed in order to transform the internal state into a final value. The output H is then the last $128-b$ bytes of the final internal state value which is ready to absorb the $(t+1)$-st message block (the 32-word internal state is interpreted as a 128-byte vector).

Our goal is to find collisions for this Compress function. In the next section we explain how collisions can be constructed for CubeHash itself.

5.4.3 Collision construction

We are planning to construct collision pairs (M', M'') for `CubeHash-`r/b which are of the form $M' = M^{\text{pre}}||M||M^t||M^{\text{suf}}$ and $M'' = M^{\text{pre}}||M \oplus \Delta||M^t \oplus \Delta^t||M^{\text{suf}}$. Here, M^{pre} is the common prefix of the colliding pairs whose length in bytes is a multiple of b, M^t is one message block of b bytes and M^{suf} is the common suffix of the colliding pairs whose length is arbitrary. The message prefix M^{pre} is chosen for randomizing the initial value V. More precisely, V is the content of the internal state after processing the message prefix M^{pre}. For this value of V, $(M, M \oplus \Delta)$ is a collision pair for the compression function, *i.e.*, $\mathsf{Compress}(M, V) = \mathsf{Compress}(M \oplus \Delta, V)$. Remember that a collision for the Compress indicates collision over the last $128 - b$ bytes of the internal state. The message blocks M^t and $M^t \oplus \Delta^t$ are used to get rid of the difference in the first b bytes of the internal state. The difference Δ^t is called the *erasing block difference* and is computed as follows. When we evaluate the Compress with inputs (M, V) and $(M \oplus \Delta, V)$, Δ^t is the difference in the first b bytes of the final internal state values.

Once we find message prefix M^{pre}, message M and difference Δ, any message pairs (M', M'') of the above-mentioned form is a collision for CubeHash for *any* message block M^t and *any* message suffix M^{suf}. We find the difference Δ using the linearization method of section 5.2, to be applied to CubeHash in the next section. Then, M^{pre} and M are found by finding a preimage of zero under the Condition function as explained in section 5.3. Algorithm 6 in appendix A.7 shows how CubeHash Condition function can be implemented in practice for a given differential path.

5.4.4 Constructing linear differentials

We linearize the compression function of CubeHash to find message differences that can be used for efficient collision search. In particular, we are interested in finding differences which result in a low theoretical collision complexity when used with the dependency table and backtracking algorithm. As a first approach one can search for

differences with a high raw probability. Another motivation is that, as we will see at the end of this section, a differential path with higher raw probability also corresponds to a better second preimage cryptanalysis. Table 5.2 indicates the $-\log_2$ probability (*i.e.*, number of bit conditions) of the differential paths with the highest raw probability which we found for $b \in \{1, 2, 3, 4, 8, 16, 32, 48, 64, 96\}$ and $r \in \{1, 2, 3, 4, 5, 6, 7, 8\}$. We did not consider the dash entries "–" since the large probability of their differential paths makes them less interesting. The corresponding differential paths can be found in appendix A.1. We would like to emphasize that since we are using linear differentials, the erasing block difference Δ^t only depends on the difference Δ, see section 5.4.3. Appendix A.1 also includes the erasing block for convenience.

The differential paths have been found as follows. As explained in section 5.2, the linear transformation $\mathsf{Compress}_{\text{lin}}$ can be identified with a matrix $\mathcal{H}$. A linear differential path is a member of the kernel of this matrix, *i.e.*, those Δ's such that $\mathcal{H}\Delta = 0$. For CubeHash-r/b with t iterations, $\Delta = \Delta^0 || \dots || \Delta^{t-1}$ and $\mathcal{H}$ has size $(1024 - 8b) \times 8bt$, see section 5.4.2. Let τ be the dimension of the kernel of $\mathcal{H}$. The matrix $\mathcal{H}$ does not have full rank for many parameters r/b and t. Therefore, one can find differences with a high raw probability in the set of linear combinations of at most λ kernel basis vectors, where $\lambda \geq 1$ is chosen such that the set can be searched exhaustively. The results heavily depend on the choice of the kernel vectors, see [148]. The kernel of $\mathcal{H}$ contains 2^τ different elements. The above method finds the best difference out of a subset of $\sum_{i=1}^{\lambda} \binom{\tau}{i}$ elements. We may find better results by increasing λ or by repeating the search for another choice of the basis. Using ideas from [74, 195] we propose an alternative search algorithm, that works well for many variants of CubeHash and does not decisively depend on the choice of the kernel basis.

Let $\Delta_0, \dots, \Delta_{\tau-1}$ be a kernel basis of $\mathsf{Compress}_{\text{lin}}$ and denote $\mathcal{G}$ the matrix whose τ rows consist of the binary vectors $\Delta_i \| \boldsymbol{\alpha}(\Delta_i) \| \boldsymbol{\beta}(\Delta_i)$ for $i = 0, \dots, \tau - 1$; refer to section 5.2.3 for the notation. Elementary row operations on $\mathcal{G}$ preserve this structure. That is, the rows always have the form $\Delta \| \boldsymbol{\alpha}(\Delta) \| \boldsymbol{\beta}(\Delta)$ where Δ lies in the kernel of $\mathsf{Compress}_{\text{lin}}$ and its raw probability is given by the Hamming weight of $\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta)$. For convenience, we call this the raw probability of the row.

Our proposed algorithm works as follows. Determine $i_{\max}$, the index of the row with the highest raw probability. Then repeat the following steps a number of $\mathcal{N}$ times:

1. Randomly choose a column index j and let i be the smallest row index such that $\mathcal{G}_{i,j} = 1$ (choose a new j if no such i exists).

2. For all row indices $k = i + 1, \ldots, \tau - 1$ such that $\mathcal{G}_{k,j} = 1$:

   - add row i to row k,
   - set $i_{\max} = k$ if row k has higher raw probability than row $i_{\max}$.

3. Move row i to the bottom of $\mathcal{G}$, shifting up rows $i + 1, \ldots, \tau - 1$ by one.

Remark 9. *Table 5.2 shows the best differential paths with regards to the raw probability found after 200 trials of the above randomized algorithm with* $\mathcal{N} = 600$ *repetitions. A more specific choice of the column index* j *in the first step does not lead to better results. In particular, we tried to prioritize choosing columns towards the end, or for every chosen column in* $\boldsymbol{\alpha}(\Delta)$ *to also eliminate the corresponding column in* $\boldsymbol{\beta}(\Delta)$*.*

$r \setminus b$	1	2	3	4	8	12	16	32	48	64	96
1	1225	221	46	32	32	–	–	–	–	–	–
2	1225	221	46	32	32	–	–	–	–	–	–
3	4238	1881	798	478	478	400	400	400	364⋆	65	–
4	2614	964	195	189	189	156	156	156	130⋆	130⋆	38
5	10221	4579	2433	1517	1517	1244⋆	1244⋆	1244⋆	1244⋆	205	127⋆
6	4238	1881	798	478	478	394	394	394	309	309	90
7	13365	5820	3028	2124	2124	1748	1748	1748	1748	447⋆	251⋆
8	2614	2614	1022	1009	1009	830	830	830	637⋆	637⋆	151

Table 5.2: Differential paths for CubeHash with the highest probability. The table includes the values of y (*i.e.*, the $-\log_2$ probability or the number of condition bits) for the differential path with the highest raw probability which we found. Any entry which is less than 512 indicates a theoretical second preimage cryptanalysis for 512-bit digest values. For $b \leq 64$, any entry which is less than 256 (already even without message modification) indicates a theoretical collision cryptanalysis for 512-bit digest values. For CubeHash-r/96, any entry which is less than 128 (already even without message modification) indicates a theoretical collision cryptanalysis for digest values of size bigger than 256 bits, see Remark 8. Some of the corresponding differential paths can be found in appendix A.1. The entries without a star are optimal with respect to both second preimage and collision cryptanalyses when message modification is exploited. However, the starred entries are only optimal with respect to second preimage cryptanalysis, see Table 5.3.

Second preimage cryptanalysis on CubeHash. Any differential path with raw probability greater than 2^{-512} can be considered as a (theoretical) second preimage

cryptanalysis on CubeHash with 512-bit digest size. In Table 5.2 the entries which do not correspond to a successful second preimage cryptanalysis (*i.e.*, $y > 512$) are shown in gray, whereas the others have been highlighted. For example, our differential path for `CubeHash`-6/4 with raw probability 2^{-478} indicates that by only one hash evaluation we can produce a second preimage with probability 2^{-478}. Alternatively, it can be stated that for a fraction of 2^{-478} messages we can easily provide a second preimage.

5.4.5 Collision cryptanalysis on CubeHash variants

Table 5.2 includes our best differential paths found with respect to raw probability or equivalently second preimage cryptanalysis. Nevertheless, when it comes to freedom degrees use for collision cryptanalysis, these trails might not be the optimal ones. In other words, for a specific r and b, there might be another differential path which is worse in terms of raw probability but is better regarding the collision cryptanalysis complexity if we use some freedom degrees speedup. As an example, for `CubeHash`-5/96 with the path which has raw probability 2^{-127} (given in equation (A.2)), using our method of section 5.3 the time complexity can be theoretically reduced to about $2^{68.7}$ (partial) evaluation of its condition function. However, there is another path with raw probability 2^{-134} (given in equation (A.4)) which has theoretical time complexity of about $2^{31.9}$ (partial) evaluation of its condition function. This behavior can be explained as follows. As previously observed in [74, 44, 238, 195, 179], conditions in early steps of the computation can be more easily satisfied than those in later steps. This is due to message modifications, (probabilistic) neutral bits, submarine modifications and other freedom degrees use. Similar techniques are used implicitly when using a dependency table to find a preimage of the condition function (and thus a collision for the compression function). This motivates to search for differences Δ such that $\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta)$ is sparse at the end. However, in general, this is not the case for trails of Table 5.2 since most of them are sparse in the beginning and dense at the end. This is due to the diffusion of the linearized compression function in the forward direction. In order to find differential trails which are sparse at the end, one can work with the inverse linearized round transformations, refer to [148] for more details. Table 5.3 shows the best paths we found regarding the reduced complexity of the collision cryptanalysis using our method of section 5.3. While most of the paths are still the optimal ones with respect to the raw probability, the starred entries indicate the ones which invalidate

this property. Some of the interesting differential paths for starred entries in Table 5.3 are given in appendix A.2. The second entries in Table 5.3 show the reduced time complexities of the collision cryptanalysis using our method of section 5.3 whereas the first entries are the complexities without message modification, that is, the $-\log_2$ values of the probability of the differential paths. To construct the dependency table, we have analyzed the Condition function at byte level, see Remark 7. Therefore, the time complexities might be improved if the dependency table is analyzed at a bit level instead. The complexity unit is (partial) evaluation of their respective Condition function. We remind that the full evaluation of a Condition function corresponding to a t-iteration differential path is almost the same as application of t iterations (rt rounds) of CubeHash. We emphasize that the complexities are independent of digest size. All the complexities which are less than $\min\left(2^{512-4b}, 2^{c/2}\right)$, see Remark 8, can be considered as a successful collision cryptanalysis for `CubeHash`-r/b if the hash size is bigger than c bits. In particular, the complexities bigger than 2^{256} for $b \leq 64$ (considering 512-bit digests) and bigger than 2^{128} for $b = 96$ (considering 256-bit digests) are worse than generic attacks, hence shown in gray. The successfully cryptanalyzed instances have been highlighted.

Effect of threshold value on cryptanalysis. Recall that a threshold value th was used to construct the dependency table, see section 5.3.2. The astute reader should realize that the complexities of Table 5.3 correspond to the optimal threshold value. To see the effect of the threshold value on the complexity, we focus on seven instances of CubeHash: `CubeHash`-3/64, -4/48, -5/96, -6/96, -4/32, -3/48, and -3/32. The first three instances are the ones for which the theoretical complexities are practically reachable and we have managed to find their real collision examples, see the next subsection. The other four instances are the ones whose theoretical complexities are just above the practically reachable values and are most probably the ones for which real collisions will be found in near future either using more advanced methods or utilizing a huge cluster of computers. Table 5.4 shows the effect of the threshold value on the complexity for these seven instances.

Real collisions for CubeHash-3/64, -4/48 and -5/96. For `CubeHash`-3/64, we use the 2-iteration message difference $\Delta = \Delta^0||\Delta^1$ of equation (A.1). Equation (A.1)

$r \setminus b$	1	2	3	4	8	12	16	32	48	64	96
1	1225 1121.0	221 135.1	46 24.0	32 15.0	32 7.6	– –	– –	– –	– –	– –	– –
2	1225 1177.0	221 179.1	46 27.0	32 17.0	32 7.9	– –	– –	– –	– –	– –	– –
3	4238 4214.0	1881 1793.0	798 720.0	478 380.1	478 292.6	400 153.5	400 102.0	400 55.6	368⋆ 53.3	65 9.4	– –
4	2614 2598.0	964 924.0	195 163.0	189 138.4	189 105.3	156 67.5	156 60.7	156 54.7	134⋆ 30.7	134⋆ 28.8	38 7.0
5	10221 10085.0	4579 4460.0	2433 2345.0	1517 1397.0	1517 1286.0	1250⋆ 946.0	1250⋆ 868.0	1250⋆ 588.2	1250⋆ 425.0	205 71.7	134⋆ 31.9
6	4238 4230.0	1881 1841.0	798 760.6	478 422.1	478 374.4	394 256.1	394 219.9	394 180.0	309 135.7	309 132.0	90 51.0
7	13365 13261.0	5820 5709.0	3028 2940.0	2124 2004.0	2124 1892.0	1748 1423.0	1748 1323.0	1748 978.0	1748 706.0	455⋆ 203.0	260⋆ 101.0
8	2614 2606.0	2614 2590.0	1022 982.0	1009 953.0	1009 889.0	830 699.0	830 662.0	830 524.3	655⋆ 313.0	655⋆ 304.4	151 80.0

Table 5.3: Differential paths for CubeHash with the least collision complexity. The table corresponds to the differential paths which provide the best collision cryptanalyses when message modification is exploited using our method of section 5.3. The first number is the $\log_2$ value of the collision complexity without message modification (*i.e.*, the $-\log_2$ probability of the path or the number of condition bits) whereas the second number is the $\log_2$ value of the theoretical reduced collision complexity using message modification. For $b \leq 64$, any entry with $\log_2$ complexity less than 256 indicates a theoretical collision cryptanalysis for 512-bit digest values. For CubeHash-r/96, any entry with $\log_2$ complexity less than 128 indicates a theoretical collision cryptanalysis for digest values of size bigger than 256 bits, see Remark 8. Some of the corresponding differential paths can be found in appendices A.1 (for entries without a star) and A.2 (for starred entries). The entries without a star are optimal with respect to both second preimage and collision cryptanalyses when message modification is exploited. However, the starred entries are only optimal with respect to the collision cryptanalysis, see Table 5.2.

Variant	$y \setminus th$	0.0	0.025	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45
3/64	65	16.3	11.3	11.4	10.6	10.2	10.4	9.9	**9.4**	10.4	10.4	11.4
4/48	134	50.8	38.3	34.6	33.6	32.3	38.4	35.0	32.2	**30.7**	32.3	37.7
5/96	134	48.0	35.2	34.1	32.1	32.2	32.6	32.0	**31.9**	31.9	42.0	42.0
6/96	90	58.0	55.1	55.0	55.0	54.3	53.4	52.8	**51.0**	51.0	52.7	54.6
4/32	156	75.2	67.4	67.4	63.7	69.0	62.0	54.8	**54.7**	60.7	66.0	63.9
3/48	368	99.7	90.6	87.2	80.3	67.6	61.2	65.3	**53.3**	65.1	83.4	102.6
3/32	400	91.0	63.3	61.5	57.1	61.8	**55.6**	70.9	70.7	79.7	82.4	98.9

Table 5.4: Effect of the threshold value on the collision cryptanalysis complexity for CubeHash. This table shows the theoretical $\log_2$ complexities of the improved collision cryptanalysis versus the threshold parameter (which is used for constructing the dependency table, see section 5.3.2) when freedom degrees are exploited using our method of section 5.3 for some CubeHash instances.

also includes the erasing block difference Δ^2, required for collision construction. Appendix A.3 includes the values of M^{pre} and $M = M^0||M^1$ for collision on `CubeHash`-3/64 with 512-bit digest size. In other words, $\mathsf{Condition}_\Delta(M, V) = 0$ holds for the corresponding condition function where V is the content of the internal state after processing the message prefix M^{pre}. According to section 5.4.3, the pair M' and M'' where

$$M' = M^{\text{pre}}||M^0||M^1||M^2||M^{\text{suf}},$$

$$M'' = M^{\text{pre}}||(M^0 \oplus \Delta^0)||(M^1 \oplus \Delta^1)||(M^2 \oplus \Delta^2)||M^{\text{suf}},$$

is a collision for `CubeHash`-3/64 for *any* message block M^2 and *any* message suffix M^{suf} of arbitrary length.

For `CubeHash`-4/48, we use the 2-iteration message difference $\Delta = \Delta^0||\Delta^1$ of equation (A.3). It also includes the erasing block difference Δ^2, required for collision construction. The values of M^{pre} and $M = M^0||M^1$, for collision on `CubeHash`-4/48 with 512-bit digest size, are provided in appendix A.4. Based on these values, collisions for `CubeHash`-4/48, similar to the way we explained for `CubeHash`-3/64, can be constructed.

For `CubeHash`-5/96, we use the 1-iteration message difference $\Delta = \Delta^0$ of equation (A.4). It also includes the erasing block difference Δ^1, required for collision construction. The values of M^{pre} and $M = M^0$, for collision on `CubeHash`-5/96 with 512-bit digest size, are provided in appendix A.5. Therefore, the pair $M' = M^{\text{pre}}||M^0||M^1||M^{\text{suf}}$ and $M'' = M^{\text{pre}}||(M^0 \oplus \Delta^0)||(M^1 \oplus \Delta^1)||M^{\text{suf}}$ is a collision for

CubeHash-5/96 for *any* message block M^1 and *any* message suffix M^{suf} of arbitrary length.

Note that a collision pair for a given r and b can be easily transformed to a collision pair for the same r and bigger b's by appending enough zeros to each message block.

Practice versus theory. We provided a framework which is handy in order to analyze many hash functions in a generic way. In practice, the optimal threshold value may be a little different from the theoretical one. Moreover, by slightly playing with the neighboring bits in the suggested partitioning corresponding to a given threshold value (Algorithm 4), we may achieve a partitioning which is more suitable for applying the attacks. In particular, Tables 5.3 and 5.4 contain the theoretical complexities for different CubeHash instances under the assumption that the Condition function behaves ideally with respect to the first issue discussed in section 5.3.2. In practice, deviation from this assumption increases the effective complexity. For particular instances, more simulations need to be done to analyze the potential non-randomness effects in order to give a more exact estimation of the practical complexity.

In the following we compare the practical complexities with the theoretical values for some cases for which their complexities are practically reachable. Moreover, for some CubeHash instances for which their complexities are unreachable in practice we try to give a more precise estimation of their effective complexities.

Our tree-based search implementation for the CubeHash-3/64 case with $th \approx 0.3$ has median complexity 2^{21} instead of the $2^{9.4}$ of Table 5.4. The median decreases to 2^{17} by backtracking three steps at each depth instead of one, see section 5.3.2. We expect the practical complexities for other instances of CubeHash with three rounds to be slightly bigger than the theoretical numbers in Table 5.3. These cases need to be further investigated.

Our detailed analysis of CubeHash-4/32, CubeHash-4/48 and CubeHash-4/64 shows that these cases perfectly match with theory. According to Table 5.3, for CubeHash-4/64 (with $th \approx 0.33$) and CubeHash-4/48 (with $th \approx 0.30$) we have the theoretical complexities $2^{28.8}$ and $2^{30.7}$, respectively. We experimentally achieve median complexities $2^{28.3}$ and $2^{30.4}$ respectively. For CubeHash-4/32 (with $th \approx 0.30$) the theoretical complexity is $2^{54.7}$. In the tree-based search algorithm, we need to satisfy 44 bit conditions at step 18, *i.e.*, $|\mathcal{Y}_{18}| = 44$. This is the node which has the highest cost and if

it is successfully passed, the remaining steps will easily be followed. In other words, a single surviving candidate at this node (which means $2^{q'_{19}} = 1$ referring to Algorithm 3) suffices to make the remaining condition bits null with little cost. Our simulations show that on average we need about 2^{10} (partial) evaluations of the condition function per one surviving candidate which arrives at depth 18. Hence, our estimation of the practical complexity is $2^{10} \times 2^{44} = 2^{54}$ which agrees with theory.

Our detailed analysis of `CubeHash`-5/96 shows that it perfectly matches with theory. According to Table 5.3, for `CubeHash`-5/96 (with $th \approx 0.30$) we have the theoretical complexities $2^{31.9}$. See also Table A.1, the corresponding dependency table. We experimentally find collisions in almost the same time. In `CubeHash`-5/64 case (with $th \approx 0.24$), the costliest node is at depth 20 for which 70 bit conditions must be satisfied, *i.e.*, $|\mathcal{Y}_{20}| = 70$. A single surviving candidate at this node suffices to make the remaining condition bits null with little cost. Our simulation shows that on average about $2^{7.0}$ (partial) evaluations of the condition function are required per one surviving candidate which arrives at depth 20. Hence, our estimation of the practical complexity is about $2^{7.0+70} = 2^{77.0}$, versus theoretical value $2^{71.7}$.

In `CubeHash`-6/16 case (with $th \approx 0.12$), the costliest node is at depth 9 for which 204 bit conditions must be satisfied, *i.e.*, $|\mathcal{Y}_9| = 204$. We need 2^{12} candidates to successfully pass this node, *i.e.*, $q'_{10} = 12$; see Algorithm 3. Our simulation shows that on average about 2^5 (partial) evaluations of the condition function are required per one surviving candidate which arrives at depth 9. Hence, our estimation of the practical complexity is about $2^{12+5+204} = 2^{221}$, versus theoretical value $2^{219.9}$.

In `CubeHash`-7/64 case (with $th \approx 0.25$), the costliest node is at depth 24 for which 201 bit conditions must be satisfied, *i.e.*, $|\mathcal{Y}_{24}| = 201$. Only one surviving candidate at this node suffices to make the remaining condition bits null with much less cost. Our simulation shows that on average about 2^7 (partial) evaluations of the condition function are required per one surviving candidate which arrives at depth 24. Hence, our estimation of the practical complexity is about $2^{7+201} = 2^{208}$, versus theoretical value $2^{203.0}$.

We emphasize that for these latter cases we did not attempt to play with the neighboring bits in the partitioning. We believe, in general, complexities can get very close to the theoretical ones if one tries to do so.

Comparison with the previous results. The first analysis of CubeHash was proposed by Aumasson *et al.* [13] in which the authors showed some non-random properties for several versions of CubeHash. A series of collision examples on `CubeHash`-$1/b$ and `CubeHash`-$2/b$ for large values of b were announced by Aumasson [12] and Dai [95]. Collision cryptanalysis was later investigated deeply by Brier and Peyrin [66]. We cryptanalyzed not only the untouched variants but also improved on all the existing results.

5.5 Generalization

In sections 5.2 and 5.3 we considered modular-addition-based compression functions which use only modular additions and linear transformations. Moreover, we concentrated on XOR approximation of modular additions in order to linearize the compression function. This method is, however, quite general and can be applied to a broad class of hash constructions, covering many of the existing hash functions. Additionally, it lets us consider other linear approximations as well. We view a compression function $H = \mathsf{Compress}(M, V) : \{0,1\}^m \times \{0,1\}^v \to \{0,1\}^h$ as a binary finite state machine (FSM). The FSM has an internal state which is consecutively updated using message M and initial value V. We assume that FSM operates as follows, and we refer to such Compress functions as *binary-FSM-based*. The concept can also cover non-binary fields.

The internal state is initially set to zero. Afterwards, the internal state is sequentially updated in a limited number of steps. The output value H is then derived by truncating the final value of the internal state to the specified output size. At each step, the internal state is updated according to one of these *two* possibilities: either the whole internal state is updated as an affine transformation of the current internal state, M and V, or *only one* bit of the internal state is updated as a *nonlinear* Boolean function of the current internal state, M and V. Without loss of generality, we assume that all of the nonlinear updating Boolean functions (NUBF) have zero constant term (*i.e.*, the output of zero vector is zero) and none of the involved variables appear as a pure linear term (*i.e.*, changing any input variable does not change the output bit with certainty). As we will see, this assumption, coming from the simple observation that we can integrate constants and linear terms in an affine updating transformation (AUT), is essential for our analysis. Linear approximations of the FSM can be

achieved by linearizing AUTs and NUBFs. To this end, an AUT is replaced with a linear transformation by ignoring its constant term (note that the only difference between an affine transformation and a linear one is the possible existence of an additive constant term). Moreover a NUBF is replaced with a linear function. Similar to section 5.2 this gives us a linearized version of the compression function which we denote by $\mathsf{Compress}_{\mathrm{lin}}(M,V)$. As we are dealing with differential cryptanalysis, we take the notation $\mathsf{Compress}_{\mathrm{lin}}(M) = \mathsf{Compress}_{\mathrm{lin}}(M,0)$. The argument given in section 5.2 is still valid: elements of the kernel of the linearized compression function (*i.e.*, Δ's *s.t.* $\mathsf{Compress}_{\mathrm{lin}}(\Delta) = 0$) can be used to construct differential trails.

Let n_{nl} denote the total number of NUBFs in the FSM. We count the NUBFs by starting from zero. We introduce four functions $\Lambda(M,V)$, $\Phi(\Delta)$, $\Lambda^{\Delta}(M,V)$ and $\Gamma(\Delta)$ all of output size n_{nl} bits. To define these functions, consider the two procedures which implement the FSMs of $\mathsf{Compress}(M,V)$ and $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$. Let the Boolean function g^k, $0 \leq k < n_{\mathrm{nl}}$, stand for the kth NUBF and denote its linear approximation as in $\mathsf{Compress}_{\mathrm{lin}}$ by g^k_{lin}. Moreover, denote the input arguments of the Boolean functions g^k and g^k_{lin} in the FSMs which compute $\mathsf{Compress}(M,V)$ and $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$ by the vectors x^k and δ^k, respectively. Note that δ^k is a function of Δ whereas x^k depends on M and V. The kth bit of $\Gamma(\Delta)$, $\Gamma_k(\Delta)$, is set to one iff the argument of the kth linearized NUBF is not the all-zero vector, *i.e.*, $\Gamma_k(\Delta) = 1$ iff $\delta^k \neq 0$. The kth bit of $\Lambda(M,V)$, $\Lambda_k(M,V)$, is the output value of the kth NUBF in the course of evaluation of $\mathsf{Compress}(M,V)$ through the execution of FSM, *i.e.*, $\Lambda_k(M,V) = g^k(x^k)$. Similarly, the kth bit of $\Phi(\Delta)$, $\Phi_k(\Delta)$, is the output value of the kth linearized NUBF in the course of evaluation of $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$ through linearized version of FSM, *i.e.*, $\Phi_k(\Delta) = g^k_{lin}(\delta^k)$. The kth bit of $\Lambda^{\Delta}(M,V)$, $\Lambda^{\Delta}_k(M,V)$, is evaluated in a more complex way: apply the kth nonlinear Boolean function on the XOR-difference of the arguments which are fed into the two Boolean functions which compute $\Lambda_k(M,V)$ and $\Phi_k(\Delta)$, *i.e.*, $\Lambda^{\Delta}_k(M,V) = g^k(x^k \oplus \delta^k)$. Refer to Table 5.1 for a list of these symbols along with the ones used in sections 5.2 and 5.3.1. We can then present the following proposition.

Proposition 5. *Let* $\mathsf{Compress}$ *be a binary-FSM-based compression function. For any message difference* Δ*, let* $\{i_0, \ldots, i_{y-1}\}$*,* $0 \leq i_0 < i_1 < \cdots < i_{y-1} < n_{\mathrm{nl}}$ *be the positions of* 1*'s in the vector* $\Gamma(\Delta)$ *where* $y = \mathrm{wt}\big(\Gamma(\Delta)\big)$*. We define the condition function* $Y = \mathsf{Condition}_{\Delta}(M,V)$ *where the* j*th bit of* Y*,* $0 \leq j \leq y-1$*, is computed as*

$$Y_j = \Lambda_{i_j}(M,V) \oplus \Lambda^{\Delta}_{i_j}(M,V) \oplus \Phi_{i_j}(\Delta). \tag{5.6}$$

Then, if Δ *is in the kernel of* $\mathsf{Compress}_{\mathrm{lin}}$, $\mathsf{Condition}_{\Delta}(M, V) = 0$ *implies that the pair* $(M, M \oplus \Delta)$ *is a collision for* $\mathsf{Compress}$ *with the initial value* V.

Proof. First, we present the following definition, a generalization of Definition 7.

Definition 8. *We say that a message* M *(for a given* V*) conforms to (or follows) the trail of* Δ *till before applying the* i*th NUBF iff*

$$\Lambda_k(M, V) \oplus \Lambda_k^{\Delta}(M, V) = \Phi_k(\Delta), \tag{5.7}$$

for $0 \leq k < i$*. Moreover, we say that the message* M *(for a given* V*) conforms to (or follows) the trail of* Δ *iff the equation (5.7) holds for* $0 \leq k < n_{\mathrm{nl}}$.

To explain the Definition 8, imagine we have three procedures which implement the FSMs to compute $\mathsf{Compress}(M, V)$, $\mathsf{Compress}(M \oplus \Delta, V)$ and $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$. If a message M (for a given V) conforms to the trial of Δ till before applying the ith NUBF, it means that at every step before the ith NUBF the difference of the internal states of the two FSMs which compute $\mathsf{Compress}(M, V)$ and $\mathsf{Compress}(M \oplus \Delta, V)$ equals the internal state value of the FSM which computes $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$. This is guaranteed by equation (5.7) because the only nonlinear operations during this period are the applications of the first $i-1$ NUBFs. Note that if a message M conforms to the trial of Δ for a given V, we then have $\mathsf{Compress}(M, V) \oplus \mathsf{Compress}(M \oplus \Delta) = \mathsf{Compress}_{\mathrm{lin}}(\Delta)$.

For simplicity, let the Boolean function g stand for the ith NUBF and denote its linear approximation as in $\mathsf{Compress}_{\mathrm{lin}}$ by g_{lin}. Moreover, denote the input arguments of the Boolean functions g and g_{lin} in the FSMs which compute $\mathsf{Compress}(M, V)$ and $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$ by the vectors x and δ, respectively. According to our notation $\Gamma_i(\Delta) = 1$ iff $\delta \neq 0$, and $\Lambda_i(M, V) = g(x)$, $\Phi_i(\Delta) = g_{lin}(\delta)$ and $\Lambda_i^{\Delta}(M, V) = g(x \oplus \delta)$. Now the role of $\Gamma_i(\Delta)$ becomes visible.

If $\Gamma_i(\Delta) = 0$, we have $\delta = 0$, and hence $g(x) \oplus g(x \oplus \delta) = g_{\mathrm{lin}}(\delta)$. In this case the equation $\Lambda_i(M, V) \oplus \Lambda_i^{\Delta}(M, V) \oplus \Phi_i(\Delta) = 0$ is satisfied by itself, and therefore we do not need to introduce a condition bit, see equation (5.6). In other words the message M automatically conforms to the trial of Δ for the initial value V till before applying the $(i+1)$-st NUBF, provided that it conforms to the trial of Δ till before applying the ith NUBF.

However, if $\Gamma_i(\Delta) = 1$, we have $x \neq x \oplus \delta$. Therefore, if the message M conforms to the trial of Δ till before applying the ith NUBF, it will also conform to the trial of Δ till before applying the $(i+1)$-st NUBF iff $g(x) \oplus g(x \oplus \delta) = g_{\mathrm{lin}}(\delta)$. That is, we need to impose one condition bit as in equation (5.6). Remember we supposed that g has no linear term, the reason for which is as follows. If g has some linear terms

and if the difference δ is nonzero only in some of those linear inputs, the equation $g(x) \oplus g(x \oplus \delta) = g_{\text{lin}}(\delta)$ is held by itself. In this case, the imposed condition bit is redundant. □

The following proposition summarizes our findings.

Proposition 6. *Suppose that* $\mathsf{Compress}_{\text{lin}}$ *is a linearized version of a given binary-FSM-based compression function* $\mathsf{Compress}(M, V)$*. For a given* Δ *in the kernel of* $\mathsf{Compress}_{\text{lin}}$ *and for an initial value* V*, a message* M *conforms to the trail of* Δ *iff the corresponding condition function satisfies* $\mathsf{Condition}_{\Delta}(M, V) = 0$*. Moreover, for a random initial value* V*, a random message* M *is a conforming one with probability* $2^{-\Gamma(\Delta)}$*.*

5.5.1 Modular addition case

Let's review the compression functions involving only linear transformations and modular addition of n-bit words. We deeply studied this subject in sections 5.2 and 5.3. The modular addition $Z = X + Y$ can be computed by considering one bit memory c for the carry bit. Let $X = (x_0, \ldots, x_{n-1})$, $Y = (y_0, \ldots, y_{n-1})$ and $Z = (z_0, \ldots, z_{n-1})$. We have

$$\begin{array}{rcll} c_{i+1} & = & c_i x_i \oplus c_i y_i \oplus x_i y_i & \text{for} \quad 0 \le i \le n-2 \\ z_i & = & x_i \oplus y_i \oplus c_i & \text{for} \quad 0 \le i \le n-1, \end{array}$$

where $c_0 = 0$. It can be argued that a compression function which uses only linear transformations and n_{add} modular additions can be implemented as a binary-FSM-based compression whose total number of NUBFs is $n_{\text{nl}} = (n-1)n_{\text{add}}$. All the NUBFs are of the form $g(x, y, z) = xy \oplus xz \oplus yz$. The XOR approximation of modular addition in section 5.2 corresponds to approximating all the NUBFs g by the zero function, *i.e.*, $g_{\text{lin}}(x, y, z) = 0$. Having the notation of sections 5.3.1, 5.2 and 5.5 in mind (see also Table 5.1) we deduce that the input argument of the kth NUBF is $(\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k)$; whereas that of the kth linearized NUBF is $(\boldsymbol{\alpha}_k, \boldsymbol{\beta}_k, 0)$. Therefore, we have $\Lambda_k(M, V) = g(\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k)$ and $\Phi_k(\Delta) = g_{\text{lin}}(\boldsymbol{\alpha}_k, \boldsymbol{\beta}_k, 0)$. Moreover, we deduce $\Gamma_k(\Delta) = \boldsymbol{\alpha}_k \vee \boldsymbol{\beta}_k \vee 0$ and $\Lambda_k^{\Delta}(M, V) = g(\mathbf{A}_k \oplus \boldsymbol{\alpha}_k, \mathbf{B}_k \oplus \boldsymbol{\beta}_k, \mathbf{C}_k \oplus 0)$. As a result we get

$$\begin{array}{rcl} Y_j & = & \Lambda_{i_j}(M, V) \oplus \Lambda_{i_j}^{\Delta}(M, V) \oplus \Phi_{i_j}(\Delta) \\ & = & g(\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k) \oplus g(\mathbf{A}_k \oplus \boldsymbol{\alpha}_k, \mathbf{B}_k \oplus \boldsymbol{\beta}_k, \mathbf{C}_k \oplus 0) \oplus 0 \\ & = & (\boldsymbol{\alpha}_{i_j} \oplus \boldsymbol{\beta}_{i_j})\mathbf{C}_{i_j} \oplus \boldsymbol{\alpha}_{i_j} B_{i_j} \oplus \boldsymbol{\beta}_{i_j} A_{i_j} \oplus \boldsymbol{\alpha}_{i_j}\boldsymbol{\beta}_{i_j} \end{array} \tag{5.8}$$

whenever $\boldsymbol{\alpha}_{i_j} \vee \boldsymbol{\beta}_{i_j} = 1$; this agrees with equation (5.5).

5.5.2 Note on the different linear approximations

Various combinations of different linear approximations of the NUBFs provide a diverse range of linear approximations of the compression function. However, one should be careful to avoid approximations which might lead to contradictions due to dependency between different approximations. In fact the probability $2^{-\Gamma(\Delta)}$ would not be a good estimate in this case if there are strong correlations between approximations. In the case of linear approximation of modular addition of n-bit words, we have $(n-1)$ NUBFs for the carry bits, out of which $n-2$ are of the form $xy \oplus xc \oplus yc$ and one of the form xy (corresponding to the carry of the LSB). There are eight linear approximations for the earlier Boolean function (because it effectively depends on three variables) and four linear approximations for the later one (because it effectively depends on two variables). This shows the possibility of $4 \times 8^{(n-2)}$ different linear approximations. For one particular linear approximation, if the difference of the two addends are α and β the output difference γ is uniquely determined. In [162] the notation of "good" differential is introduced to distinguish those differentials which can happen with non-zero probability. A differential $\alpha, \beta \rightarrow \gamma$ is not "good" iff for some $i \in [0, n-1]$, $\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1} \neq \alpha_i \oplus \beta_i \oplus \gamma_i$ [162]. The exact probability of "good" differentials can be computed from Algorithm 2 of [162]. In general, it might not be easy to take redundancies into account. However, cryptanalysts should try their best. We also would like to emphasize that although there exists an exponential number of linear approximations (in terms of n_{nl}) for the compression function, it would be better in practice to concentrate on those for which highly probable linear differential paths are found easily. For example, by approximating the NUBFs with the zero function or sparse linear functions, the $h \times m$ matrix $\mathcal{H}$ which satisfies $\mathsf{Compress}_{\text{lin}}(\Delta) = \mathcal{H}\Delta$ is likely sparser, making it easier to find differential paths with good raw probability.

5.6 Application to MD6

MD6 [206], designed by Rivest *et al.*, was a first round SHA-3 candidate that provides security proofs regarding some differential attacks. The core part of MD6 is the function f which works with 64-bit words and maps 89 input words $(A_0, \ldots, A_{88})$ into 16 output words $(A_{16r+73}, \ldots, A_{16r+88})$ for some integer r representing the number of rounds.

Each round is composed of 16 steps. The function f is computed based on the following recursion

$$A_{i+89} = L_{r_i,l_i}\big(S_i \oplus A_i \oplus (A_{i+71} \wedge A_{i+68}) \oplus (A_{i+58} \wedge A_{i+22}) \oplus A_{i+72}\big), \tag{5.9}$$

where S_i's are some publicly known constants and L_{r_i,l_i}'s are some known simple linear transformations. The 89-word input of f is of the form $Q||U||W||K||B$ where Q is a known 15-word constant value, U is a one-word node ID, W is a one-word control word, K is an 8-word key and B is a 64-word data block. For more details about function f and the mode of operation of MD6, we refer to the submission document [206].[1] We consider the compression function $H = \mathsf{Compress}(M, V) = f(Q||U||W||K||B)$ where $V = U||W||K$, $M = B$ and H is the 16-word compressed value. Our goal is to find a collision $\mathsf{Compress}(M, V) = \mathsf{Compress}(M', V)$ for arbitrary value of V. We later explain how such collisions can be translated into collisions for the MD6 hash function.

According to our model (section 5.5), MD6 can be implemented as an FSM which has $64 \times 16r$ NUBFs of the form $g(x, y, z, w) = x \cdot y \oplus z \cdot w$. Remember that the NUBFs must not include any linear part or constant term. We focus on the case where we approximate all NUBFs with the zero function. This corresponds to ignoring the AND operations in equation (5.9). This essentially says that in order to compute $\mathsf{Compress}_{\mathrm{lin}}(\Delta) = \mathsf{Compress}_{\mathrm{lin}}(\Delta, 0)$ for a 64-word $\Delta = (\Delta_0, \ldots, \Delta_{63})$, we map $(A'_0, \ldots, A'_{24}, A'_{25}, \ldots, A'_{88}) = 0||\Delta = (0, \ldots, 0, \Delta_0, \ldots, \Delta_{63})$ into the 16 output words $(A'_{16r+73}, \ldots, A'_{16r+88})$ according to the linear recursion

$$A'_{i+89} = L_{r_i,l_i}\big(A'_i \oplus A'_{i+72}\big). \tag{5.10}$$

For a given Δ, the function Γ is the concatenation of $16r$ words $A'_{i+71} \vee A'_{i+68} \vee A'_{i+58} \vee A'_{i+22}$, $0 \leq i \leq 16r - 1$. Therefore, the number of bit conditions equals

$$y = \sum_{i=0}^{16r-1} \mathrm{wt}(A'_{i+71} \vee A'_{i+68} \vee A'_{i+58} \vee A'_{i+22}). \tag{5.11}$$

Note that this equation compactly integrates cases 1 and 2 given in section 6.9.3.2 of [206] for counting the number of active AND gates. Algorithm 5 in appendix A.7 shows how the $\mathsf{Condition}$ function is implemented using equations (5.6), (5.9) and(5.10).

[1] In the MD6 document [206], the control word W and the linear function L_{r_i,l_i} are respectively denoted by V and g_{r_i,l_i}.

Using a similar linear algebraic method to the one used in section 5.4.4 for CubeHash, we have found the collision difference of equation (5.12) for $r = 16$ rounds with a raw probability $p_\Delta = 2^{-90}$. In other words, Δ is in the kernel of $\mathsf{Compress}_{\text{lin}}$ and the condition function has $y = 90$ output bits. Note that this does not contradict the proven bound in [206]: one gets at least 26 active AND gates.

$$\Delta_i = \begin{cases} \texttt{F6D164597089C40E} & i = 2 \\ \texttt{2000000000000000} & i = 36 \\ \texttt{0} & 0 \leq i \leq 63, i \neq 2, 36 \end{cases} \tag{5.12}$$

In order to efficiently find a conforming message pair for this differential path, we need to analyze the dependency table of its condition function. Referring to our notation in section 5.3.2, our analysis of the dependency table of function $\mathsf{Condition}_\Delta(M, 0)$ at word level (units of $u = 64$ bits) shows that the partitioning of the condition function is as in Table A.2 for threshold value $th = 0$. For this threshold value clearly $p_i = 0$. The optimal values for q_i's (computed according to the complexity analysis of the same section) are also given in Table A.2, showing a total attack complexity of $2^{30.6}$ (partial) condition function evaluation.[1] By analyzing the dependency table with smaller units the complexity may be subject to reduction.

Having set V to zero (which corresponds to choosing null values for the key, the node ID and the control word in order to simplify things), we found a message M, given in the appendix A.6, which makes the condition function null. In other words, the message pairs M and $M \oplus \Delta$ are colliding pairs for $r = 16$ rounds of f. This 16-round colliding pair provides near collisions for $r = 17$, 18 and 19 rounds, respectively, with 63, 144 and 270 bit differences over the 1024-bit long output of f.

Now let's discuss the potential of providing collisions for reduced-round MD6 hash function from collisions for reduced-round f. The MD6 mode of operation is optionally parametrized by an integer L, $0 \leq L \leq 64$, which allows a smooth transition from the default tree-based hierarchical mode of operation (for $L = 64$) down to an iterative mode of operation (for $L = 0$). When $L = 0$, MD6 works in a manner similar to that of the well-known Merkle-Damgård construction (or the HAIFA method). Since in the iterative Merkle-Damgård the first 16 words of the message block are used as a

[1] By masking M_{38} and M_{55} respectively with `092E9BA68F763BF1` and `DFFBFF7FEFFDFFBF` after random setting, the 35 condition bits of the first three steps are satisfied for free, reducing the complexity to $2^{30.0}$ instead. See Table A.2.

chaining value, and as our difference in equation (5.12) is non-zero in the first 16 words, we do not get a collision but a pseudo-collision. Nevertheless, for 16-round MD6 in the tree-based hierarchical mode of operation (*i.e.*, for $1 \leq L \leq 64$), we get a hash collision. We emphasize that one must choose node ID U and control word W properly in order to fulfill the MD6 restriction on these values as opposed to the null values which we chose. This is the first real collision example for 16-round MD6. The original MD6 submission [206] mentions inversion of the function f up to a dozen rounds using SAT solvers. Some slight nonrandom behavior of the function f up to 33 rounds has also been reported [150].

5.7 Summary

We presented a framework for an in-depth study of linear differential attacks on hash functions. We applied our method to reduced round variants of CubeHash and MD6. As of April 2010, our results on CubeHash are still by far the best known collision cryptanalyses. However, in [132] our results on MD6 were slightly improved by finding better linear differential paths. MD6 is of no interest any more as it was only among the first round SHA-3 candidates and was withdrawn by the designers just before the start of the second round. In contrast, CubeHash is one of the promising candidates in the second round.

6

Conclusion

The field of design and analysis of symmetric cryptographic components is a fascinating realm of research activity. The struggle between primitive designers and cryptanalysts has kept the field very lively. Cryptographers over the years have learned to include a tunable parameter in their design in order to provide flexible security/performance trade-offs. It is scientifically interesting to explore how much the number of mathematical operations can be reduced to construct a cipher that does not cower against cryptanalysts. On the other hand, the cryptographic community and standardization organizations have also learned that the best way to achieve a cipher which can stay alive for a long period of time is to go through public evaluations. Cipher designers choose the security parameter such that their schemes are fast enough while still keeping some reasonable level of confidence on their designs. Cryptanalysts on the other hand try to break reduced-round variants, which are faster, and get as close as possible to the security parameter set by the designers. In this thesis we cryptanalyzed various stream cipher and hash function design proposals, including several candidates of the ECRYPT eSTREAM project and NIST SHA-3 competition. Although we cannot break the schemes in many cases, our results have either remained the best so far or inspired newer improved results.

Appendix A

A.1 The best differential paths found for CubeHash regarding raw probability

Here we give the differential trials for highlighted entries of Table 5.2 and the entries of Table 5.3 which are not starred. A differential path is denoted by $(\Delta^0, \Delta^1, \ldots, \Delta^t)$ where $\Delta = \Delta^0||\Delta^1||\ldots||\Delta^{t-1}$ is in the kernel of $\mathsf{Compress}_{\text{lin}}$ and Δ^t is the corresponding erasing block difference. Recall that each Δ^i consists of b bytes and $\mathsf{Compress}_{\text{lin}}$ linearly maps Δ in t iterations into the last $128 - b$ bytes of the final state. The erasing block difference Δ^t is then the contents of the first b bytes of the state, hence only depending on Δ. Nevertheless, we also provide this value for convenience. See sections 5.4.2 and 5.4.4 for more details. Note that a differential path for a given r and b can be easily transformed to a differential path for the same r and bigger b's by appending enough zeros to each difference block. Therefore, we present the differential path for the smallest valid b.

A.1.1 Differential paths for CubeHash-$1/\star$

`CubeHash-1/4` **with** $y = 32$**:**

$$(\Delta^0, \ldots, \Delta^4) = (\texttt{00000001}, \texttt{00000000}, \texttt{40400010}, \texttt{00000000}, \texttt{00000010})$$

`CubeHash-1/3` **with** $y = 46$**:**

$$(\Delta^0, \ldots, \Delta^4) = (\texttt{010000}, \texttt{000000}, \texttt{104040}, \texttt{000000}, \texttt{100000})$$

`CubeHash-1/2` **with** $y = 221$**:**

$$(\Delta^0, \ldots, \Delta^8) = (\texttt{0080}, \texttt{0000}, \texttt{2200}, \texttt{0000}, \texttt{228A}, \texttt{0000}, \texttt{0280}, \texttt{0000}, \texttt{2000})$$

A.1.2 Differential paths for CubeHash-2/⋆

CubeHash-2/4 **with** $y = 32$**:**

$$(\Delta^0, \Delta^1, \Delta^2) = (\texttt{00000001}, \texttt{40400010}, \texttt{00000010})$$

CubeHash-2/3 **with** $y = 46$**:**

$$(\Delta^0, \Delta^1, \Delta^2) = (\texttt{010000}, \texttt{104040}, \texttt{100000})$$

CubeHash-2/2 **with** $y = 221$**:**

$$(\Delta^0, \ldots, \Delta^4) = (\texttt{0040}, \texttt{1100}, \texttt{1145}, \texttt{0140}, \texttt{1000})$$

A.1.3 Differential paths for CubeHash-3/⋆

CubeHash-3/64 **with** $y = 65$**:**

$$\begin{aligned}
\Delta^0 = {} & \texttt{40000000000000000000000000000000000000000000000000} \\
& \texttt{00000000000000000000080000000000000008000000000} \\
& \texttt{00000000000000000000000000000000} \\
\Delta^1 = {} & \texttt{00000000004100000000000000100000000000000000} \\
& \texttt{80002080000000800800020808000000000000000000000} \\
& \texttt{41000010000000000140000000000000} \\
\Delta^2 = {} & \texttt{00000000000000000000000000000000000000000002000} \\
& \texttt{00000000000020000000000000000000000000000000000} \\
& \texttt{00000000000000000000000000000010}
\end{aligned} \tag{A.1}$$

CubeHash-3/48 **with** $y = 364$**:**

$$\begin{aligned}
\Delta^0 = {} & \texttt{00000000000000000000000000000000000010000000000} \\
& \texttt{00000100000000000000000000000000000000000000000} \\
\Delta^1 = {} & \texttt{00020280000200000002028000020000000000000000000} \\
& \texttt{00000000000000004000010100000000400001010000000} \\
\Delta^2 = {} & \texttt{20202200000000002020220000000000400000050000000} \\
& \texttt{40000005000000004044004040440040404400404044040} \\
\Delta^3 = {} & \texttt{08000000800020808000000080002080000000000000000} \\
& \texttt{00000000000000000040000000000000040000000000000}
\end{aligned}$$

CubeHash-3/12 **with** $y = 400$**:**

$$\begin{aligned}
\Delta^0 = {} & \texttt{00020000000000000020000} \\
\Delta^1 = {} & \texttt{00000000000000000000000} \\
\Delta^2 = {} & \texttt{800A2A8200000000800A2A82} \\
\Delta^3 = {} & \texttt{00000000000000000000000} \\
\Delta^4 = {} & \texttt{00002000000000000002000}
\end{aligned}$$

CubeHash-$3/4$ **with** $y = 478$**:**

$$(\Delta^0, \ldots, \Delta^4) = (\texttt{00000100}, \texttt{00000000}, \texttt{41400515}, \texttt{00000000}, \texttt{00000010})$$

CubeHash-$3/3$ **with** $y = 798$**:**

$$(\Delta^0, \ldots, \Delta^4) = (\texttt{000280}, \texttt{000000}, \texttt{0A2A8A}, \texttt{000000}, \texttt{080020})$$

A.1.4 Differential paths for CubeHash-$4/\star$

CubeHash-$4/95$ **with** $y = 38$**:**

$$\begin{aligned}
\Delta^0 = \;& \texttt{00000010000000000000000000000000000000000000000000} \\
& \texttt{00000200000002000800000000000000000000000000000000} \\
& \texttt{00000000000000000000000000000000004101000000000} \\
& \texttt{00001000000000000000000000000000000280000002} \\
\Delta^1 = \;& \texttt{01000000000000001400001000000000000000000000000} \\
& \texttt{08200000820000080000000000000000000000000000000} \\
& \texttt{00000000000000000000000000000000000000000000000} \\
& \texttt{00000001000000000000000000000000200000082000}
\end{aligned}$$

CubeHash-$4/36$ **with** $y = 130$**:**

$$\begin{aligned}
\Delta^0 = \;& \texttt{00200000000000000200000000000000000000000000000} \\
& \texttt{000000000000000000001000} \\
\Delta^1 = \;& \texttt{00022002000000000022002000000000000000000000000} \\
& \texttt{000000000000000001000110} \\
\Delta^2 = \;& \texttt{00002000000000000002000000000000000000000000000} \\
& \texttt{000000000000000000000010}
\end{aligned}$$

CubeHash-$4/12$ **with** $y = 156$**:**

$$\begin{aligned}
\Delta^0 = \;& \texttt{040000000000000004000000} \\
\Delta^1 = \;& \texttt{004400400000000000440040} \\
\Delta^2 = \;& \texttt{000400000000000000040000}
\end{aligned}$$

CubeHash-$4/4$ **with** $y = 189$**:**

$$(\Delta^0, \Delta^1, \Delta^2) = (\texttt{00001000}, \texttt{01000110}, \texttt{00000010})$$

CubeHash-$4/3$ **with** $y = 195$**:**

$$(\Delta^0, \Delta^1, \Delta^2) = (\texttt{000200}, \texttt{200022}, \texttt{000002})$$

A.1.5 Differential paths for CubeHash-5/⋆

CubeHash-5/96 with $y = 127$:

$$\begin{aligned}
\Delta^0 = \; & 40000000000000004000000000000000000000000000000000 \\
& 00000000000000000200000000000000000000000000000000 \\
& 00000000000000000000000000000000000000000000000040 \\
& 00000000000004000000000000200000000000000000000000 \\
\Delta^1 = \; & 0100011101000111000000000000000008008002A00000000 \\
& 08000022000000000000000000000000000000000000000000 \\
& 00000000000000001104000000000004000010101000111 \\
& 00000000000000000002208000000000800200000000000
\end{aligned} \tag{A.2}$$

CubeHash-5/64 with $y = 205$:

$$\begin{aligned}
\Delta^0 = \; & 04000000000000000000000000000000000000000000000000 \\
& 00000000000000000000008000000000000000800000000 \\
& 00000000000000000000000000000000 \\
\Delta^1 = \; & 00000000004005000000000000104000000000000000000 \\
& 00022202000202802020220008202000000000000000000 \\
& 01400510000000000040040100000000 \\
\Delta^2 = \; & 00000000000000000000000000000000000000000002000 \\
& 00000000000020000000000000000000000000000000000 \\
& 00000000000000000000000000000010
\end{aligned}$$

A.1.6 Differential paths for CubeHash-6/⋆

CubeHash-6/87 with $y = 90$:

$$\begin{aligned}
\Delta^0 = \; & 00000000000000000000001000000000020000000200000 \\
& 00000000000000000000000000000000000008000000000 \\
& 00000000000000000000000000000000000010000000000 \\
& 40000101000000000200008002000 \\
\Delta^1 = \; & 04440400000000001040400000000000800A0000800A000 \\
& 00000000000000000000000000000008000020200000000 \\
& 00000000000000000000000000000004040001000000000 \\
& 00040000000000000020800008000A0
\end{aligned}$$

CubeHash-6/48 with $y = 309$:

$$\begin{aligned}
\Delta^0 = \; & 00000000002000000000000002000000000000000000 \\
& 00000000000000000000000000001000000000000000000 \\
\Delta^1 = \; & 00000000800A2A8200000000800A2A820000000000000000 \\
& 00000000000000000000000041400515000000000000000 \\
\Delta^2 = \; & 00000000002000000000000002000000000000000000 \\
& 00000000000000000000000000001000000000000000000
\end{aligned}$$

CubeHash-6/36 **with** $y = 351$**:**

$$\begin{aligned}\Delta^0 = {} & \texttt{00010000000000000001000000000000000000000000000000} \\ & \texttt{00000000000000000800000} \\ \Delta^1 = {} & \texttt{40051541000000004005154100000000000000000000000} \\ & \texttt{0000000000000020A0828A} \\ \Delta^2 = {} & \texttt{00001000000000000001000000000000000000000000000000} \\ & \texttt{00000000000000000000008}\end{aligned}$$

CubeHash-6/32 **with** $y = 394$**:**

$$\begin{aligned}\Delta^0 = {} & \texttt{00000000000000000000000000000000} \\ & \texttt{00000000000001000000000000000100} \\ \Delta^1 = {} & \texttt{00000000000000000000000000000000} \\ & \texttt{00000000414005150000000041400515} \\ \Delta^2 = {} & \texttt{00000000000000000000000000000000} \\ & \texttt{00000000000001000000000000000010}\end{aligned}$$

CubeHash-6/12 **with** $y = 394$**:**

$$\begin{aligned}\Delta^0 = {} & \texttt{000000400000000000000040} \\ \Delta^1 = {} & \texttt{414510500000000041451050} \\ \Delta^2 = {} & \texttt{000400000000000000040000}\end{aligned}$$

CubeHash-6/4 **with** $y = 478$**:**

$$(\Delta^0, \Delta^1, \Delta^2) = (\texttt{00000100}, \texttt{41400515}, \texttt{00000010})$$

A.1.7 Differential paths for CubeHash-7/⋆

CubeHash-7/96 **with** $y = 251$**:**

$$\begin{aligned}\Delta^0 = {} & \texttt{00000000000000000000000000000000000000040000000} \\ & \texttt{00000000400000000000000000000000000000000000000} \\ & \texttt{00000000000000000000000000200000000000000000000} \\ & \texttt{00020000000000000000040000000000000004000000000} \\ \Delta^1 = {} & \texttt{0000000002800A2000000000000A28820000000000000000} \\ & \texttt{5144544051445440000000010014005000000000000000} \\ & \texttt{00000000000000000000000000000000000000080080200} \\ & \texttt{00000000800A2002000000000000000051445440404400 40}\end{aligned}$$

CubeHash-7/64 **with** $y = 447$**:**

$$\begin{aligned}
\Delta^0 = {} & \texttt{00008000000000000008000000000000000000000000000} \\
& \texttt{00000000000000000000040000000000000000000000000} \\
& \texttt{00000000000000000000000000000000} \\
\Delta^1 = {} & \texttt{A880A2888800808000000000000000005004011400000000} \\
& \texttt{15400400000000000000000004510400000000040011000} \\
& \texttt{00000000000000000880A288A08000888} \\
\Delta^2 = {} & \texttt{00000000000000000000000000000000000000000000000} \\
& \texttt{00000000000001000000000000000000000000000000000} \\
& \texttt{00000000000020000000000000002000}
\end{aligned}$$

A.1.8 Differential paths for CubeHash-8/$\star$

CubeHash-8/88 **with** $y = 151$**:**

$$\begin{aligned}
\Delta^0 = {} & \texttt{00000000000000080000000080002084000000140000001} \\
& \texttt{00000000000000000000000000000000000000004040001} \\
& \texttt{00000000000000000000000000000000000000000020808} \\
& \texttt{00000000080008884000000100001041} \\
\Delta^1 = {} & \texttt{00000000880800080000000080800020040011000400110} \\
& \texttt{00000000000000000000000000000000000000001040400} \\
& \texttt{00000000000000000000000000000000000000008000208} \\
& \texttt{00000000080000000040011040000100}
\end{aligned}$$

A.2 The best differential paths found for CubeHash regarding collision complexity

Here we give the differential trials for some starred entries of Table 5.3. Please see the descriptions in appendix A.1.

CubeHash-3/48 **with** $y = 368$**:**

$$\begin{aligned}
\Delta^0 = {} & \texttt{00000000000000000000000000004000000000000000} \\
& \texttt{40000000000000000000000000000000000000000000000} \\
\Delta^1 = {} & \texttt{80002080000000808000208000000080000000000000000} \\
& \texttt{00000000000000004040001000000004040001000000000} \\
\Delta^2 = {} & \texttt{88080008000000008808000800000000040011000000000} \\
& \texttt{00400110000000001100101011001010110010101100100} \\
\Delta^3 = {} & \texttt{00000028000020200000002800002020000000000000000} \\
& \texttt{00000000000000001000000000000001000000000000000}
\end{aligned}$$

CubeHash-4/36 **with** $y = 134$**:**

$$\begin{aligned}
\Delta^0 = \ & \texttt{00000008000000000000000800000000000000000000000000} \\
& \texttt{00000000000000004000000} \\
\Delta^1 = \ & \texttt{88008000000000088008000000000000000000000000000} \\
& \texttt{00000000000000000440040} \\
\Delta^2 = \ & \texttt{08000000000000008000000000000000000000000000000} \\
& \texttt{00000000000000000040000}
\end{aligned} \tag{A.3}$$

CubeHash-5/96 **with** $y = 134$**:**

$$\begin{aligned}
\Delta^0 = \ & \texttt{08000208080002080000000000000004000010000000000} \\
& \texttt{00400110000000000000000000000000000000000000000} \\
& \texttt{00000000000000000800A000000000000800088808000208} \\
& \texttt{00000000000000004001100000000000045104000000000} \\
\Delta^1 = \ & \texttt{80000000000000080000000000000000000000000000000} \\
& \texttt{00000000000000000400000000000000000000000000000} \\
& \texttt{00000000000000000000000000000000000000000000080} \\
& \texttt{00000000000008000000000004000000000000000000000}
\end{aligned} \tag{A.4}$$

CubeHash-7/61 **with** $y = 455$**:**

$$\begin{aligned}
\Delta^0 = \ & \texttt{04000000000000000000000000000000000000000000000} \\
& \texttt{00000000000000000000008000000000000008000000000} \\
& \texttt{00000000000000000000000} \\
\Delta^1 = \ & \texttt{00000000450040100000000014000100000000000000000} \\
& \texttt{88A880A288008080880A288A88080008000000000000000} \\
& \texttt{01451040000000005001440000} \\
\Delta^2 = \ & \texttt{00000000000000000000000000000000000000000000002} \\
& \texttt{00000000000000200000000000000000000000000000000} \\
& \texttt{00000000000000000000001}
\end{aligned}$$

CubeHash-7/96 **with** $y = 260$**:**

$$\begin{aligned}
\Delta^0 = \ & \texttt{A008020800000000280002000000000000000000000000} \\
& \texttt{01000111010001110450040100000000000000000000000} \\
& \texttt{00000000000000000000000000000008008202A00000000} \\
& \texttt{080208A000000000000000000000001451151010001111} \\
\Delta^1 = \ & \texttt{00000000000000000000000000000000000040000000000} \\
& \texttt{00004000000000000000000000000000000000000000000} \\
& \texttt{00000000000000000000002000000000000000000000000} \\
& \texttt{00000000000200000000000040000000000000040000}
\end{aligned}$$

A.3 Collisions for CubeHash-3/64

$$
\begin{aligned}
M^{\mathrm{pre}} = \;& \texttt{9B91E97363511AC3AF950F54DBCFD5DF91BC26BDD759104D} \\
& \texttt{F15B37847A4F7015E15A8844ABA3075A3816AE13E583F276} \\
& \texttt{40193317724464649F9BE819EB582ECC} \\
M^{0} = \;& \texttt{B22A98139CC0C8606525818EE6DD7775CF25B34196DC51F4} \\
& \texttt{641E56ACB918296BBD082AD01D7481EECC950B6C176C45B6} \\
& \texttt{23CFE1E2638B16255F61E806F34DE91C} \\
M^{1} = \;& \texttt{4D9E9CD62ED12CBDBA1E0B631856DCFE5BD996571CFF6E94} \\
& \texttt{A52242382E154FA6AEB44AC0A247CB298550C7B82BDCA924} \\
& \texttt{E81D5E51E997CA67FBDD86FF15D04A0D}
\end{aligned}
$$

A.4 Collisions for CubeHash-4/48

$$
\begin{aligned}
M^{\mathrm{pre}} = \;& \texttt{741B87597F94FF1CC01761CA0D80B07CC2E6E760C95DF9A5} \\
& \texttt{08FFCBABDA11474E2CCEA7AC62A7C822BE29EDCBA99D476C} \\
M^{0} = \;& \texttt{1D30F8022F4AE8DBD477FA1F7DE37C1AF2516BC6FA4657F9} \\
& \texttt{E51539C10EC114DA3B8264DD9361FE07C3D56E88E8512201} \\
M^{1} = \;& \texttt{014A11BFE2FF346FC306D1E430EE80268785A9F841562C9A} \\
& \texttt{88A6BF5858E95362F541ACF41C2FDCC1C49470DF1DFAEFDC}
\end{aligned}
$$

A.5 Collisions for CubeHash-5/96

$$
\begin{aligned}
M^{\mathrm{pre}} = \;& \texttt{F06BB068487C5FE1CCCABA700A989262801EDC3A69292196} \\
& \texttt{8848F445B8608777C037795A10D5D799FD16C037A52D0B51} \\
& \texttt{63A74C97FD858EEF7809480F43EB264CD66318632A8CCFE2} \\
& \texttt{EA22B139D99E48888CA844FBECCE3295150CA98EB16B0B92} \\
M^{0} = \;& \texttt{3DB4D4EE02958F578EFF307A5BE9975B4D0A669EE6025663} \\
& \texttt{8DDB6421BAD8F1E4384FE1284EBB7E2A72E165871E44C51B} \\
& \texttt{DA607FD91DDAD41F4180297A1607F9022463D2592B73F829} \\
& \texttt{C79E766D0F672ECC084E841BFC700F053095E8658EEB85D5}
\end{aligned}
$$

A.6 Colliding message for MD6 reduced to 16 rounds

```
5361E9B8579F7CD1, 8B29C52CA2AB51E4, 0BCF2F1E1B116898, 022C254B88191A11,
F0F1CE9D9A7F63B8, 9FB5B2CE87B7D7F5, E7C78F28EEB4F5C7, C5E8C19CEFC07365,
F88B84529ED90209, 8FACF593AE7390CF, 03A93466247C6B54, B12C70C10904143D,
D92EE67244C300D6, 35EEA586ECCC8A77, 9DCF031C64B528F8, C84807607ADCD418,
367E95EE3CB0FC67, 578A2C716FCC5016, B0C30EA5521F61EF, 7F665B24762D5894,
4196BAF0596A7784, ED5F9A8F183B4BCC, 6077463601FCFE46, 495366B1273E119B,
6E11A21AE5B3A48F, 38082264A0F68F93, 4ED510C2DFA9FF98, 35C5ACEC5E9A1756,
1F6731C861879ECD, 8CECD7B4F761CE82, 332A50854FDA8FE6, 588498B1021E9C23,
CB1FFA21CF89C7A5, 63A6871C77848410, 92A550CB4607F31C, 97024803F162E055,
E2D6EA5A57D2DBF3, AEB418A0F1F01CC5, 090A9304040038C1, 5417960E3D9A06A5,
714215C196813F35, BABAD7A4C154F2C3, 71AF3FD02B543940, FA08624B825648DD,
730D61FF48759275, CF85BA5A06D6AED4, 2E12B3150452C65A, 93C7A9FC314220B4,
81B128A4EF361456, BFE652098170C212, 77540989DC246845, 796F353D07721071,
D82776A3CBFEC586, 1132E4391152F408, CE936924CFFB22AA, D338852F80450282,
4F41AB82E790EEF6, F05378CB6BD36203, 5E506F47C6EC4617, FE6FB5A03BDE8E1C,
AB33EA511EEBAEDC, 7D40F8D4F0C62BF4, 1174E2B748B9CC2E, 1EB743671A31547D
```

A.7 Condition function for CubeHash and MD6

Algorithms 5 and 6 respectively show how the Condition function can be constructed for MD6 and CubeHash. It is presumed that the input Δ is in the kernel of their respective condition functions. Note that, in case the Condition function needs to be evaluated several times for a fixed differential path Δ (as is the case for the tree-based search algorithm), we can precompute that part of the function which is independent of the input message M.

A.8 Partitioning example for CubeHash

Table A.1 shows the input and output partitionings of the Condition function of CubeHash-5/96 for the difference Δ in equation (A.4).

A.9 Partitioning example for MD6

Table A.2 shows the input and output partitionings of the Condition function of MD6 with $r = 16$ rounds for the difference Δ in equation (5.12).

i	$\mathcal{M}_i$	$\mathcal{Y}_i$	q_i
0	–	$\emptyset$	0.00
1	$\{2, 6, 66\}$	$\{1, 2\}$	2.00
2	$\{10, 1, 9, 14, 74, 5, 13, 65, 17, 70\}$	$\{5\}$	1.35
3	$\{73, 7, 16, 19, 18, 78, 25, 37, 41\}$	$\{23, 24\}$	2.00
4	$\{69, 77, 24, 33\}$	$\{21, 22\}$	2.00
5	$\{50, 89\}$	$\{12, 13\}$	2.00
6	$\{20, 27, 45, 88\}$	$\{11\}$	1.15
7	$\{57, 4\}$	$\{38\}$	1.00
8	$\{80\}$	$\{7, 8\}$	2.00
9	$\{38, 40, 81, 3, 28, 32\}$	$\{34\}$	1.24
10	$\{49\}$	$\{41\}$	1.00
11	$\{58\}$	$\{19, 20, 42, 43\}$	4.00
12	$\{91\}$	$\{16, 17\}$	2.00
13	$\{23, 34, 44, 83\}$	$\{29, 30\}$	2.07
14	$\{90\}$	$\{14\}$	1.07
15	$\{15, 26\}$	$\{15\}$	1.07
16	$\{36\}$	$\{37, 55\}$	2.31
17	$\{42, 46, 48\}$	$\{25, 26\}$	2.12
18	$\{56\}$	$\{18, 31, 40\}$	3.01
19	$\{59\}$	$\{48, 79\}$	2.00
20	$\{84, 92, 0\}$	$\{35\}$	1.00
21	$\{82\}$	$\{9, 10, 27, 28, 32, 33\}$	6.04
22	$\{31, 51\}$	$\{44, 56, 64\}$	3.03
23	$\{71\}$	$\{6\}$	1.00
24	$\{11, 54, 67\}$	$\{3\}$	1.00
25	$\{75\}$	$\{78\}$	1.00
26	$\{21, 55\}$	$\{46, 59\}$	2.00
27	$\{63\}$	$\{50\}$	1.00
28	$\{79\}$	$\{45, 49, 65, 70\}$	4.00
29	$\{12\}$	$\{71\}$	1.06
30	$\{22\}$	$\{58, 67, 81, 82, 83\}$	5.00
31	$\{29, 62\}$	$\{63\}$	1.03
32	$\{87, 95\}$	$\{53, 54, 74, 76, 85\}$	5.01
33	$\{39, 47\}$	$\{39\}$	1.01
34	$\{53, 8\}$	$\{69, 88, 89\}$	3.30
35	$\{30\}$	$\{77, 86, 94, 98\}$	5.04
36	$\{60, 61\}$	$\{62, 91, 101, 102\}$	4.35
37	$\{35, 52\}$	$\{61, 90, 103\}$	4.22
38	$\{43\}$	$\{36, 57, 60, 104, 111\}$	5.77
39	$\{64\}$	$\{0\}$	1.33
40	$\{68\}$	$\{4\}$	2.03
41	$\{72\}$	$\{97, 100, 121\}$	8.79
42	$\{76\}$	$\{66, 80, 92, 93\}$	13.39
43	$\{85\}$	$\{47, 112\}$	16.92
44	$\{93\}$	$\{51, 52, 68, 72, 75, 87, 95\}$	22.91
45	$\{86, 94\}$	$\{73, 84, 96, 99, 105, \ldots, 110, 113, \ldots, 132, 133\}$	31.87

Table A.1: Partitioning example for CubeHash. This table shows the input and output partitionings of the Condition function of CubeHash-5/96 for the difference Δ in equation (A.4). Numbers in $\mathcal{M}_i$ are message byte indices, whereas numbers in $\mathcal{Y}_i$ are condition bit indices. The total complexity of the attack is dominated by the last step in which 31 condition bits must be satisfied.

Algorithm 5 : Condition function for MD6

Inputs: r, Δ, M, and V.

Outputs: y and $Y = \mathsf{Condition}_\Delta(M, V)$.

$(A_0, \ldots, A_{88}) := Q||V||M$
$(A'_0, \ldots, A'_{24}, A'_{25}, \ldots, A'_{88}) := 0||\Delta$
$j := 0$
for $i = 0, 1, \ldots, 16r - 1$ **do**
  $A_{i+89} := L_i\big(S_i \oplus A_i \oplus (A_{i+71} \wedge A_{i+68}) \oplus (A_{i+58} \wedge A_{i+22}) \oplus A_{i+72}\big)$
  $A'_{i+89} := L_i\big(A'_i \oplus A'_{i+72}\big)$
  $D := A'_{i+71} \vee A'_{i+68} \vee A'_{i+58} \vee A'_{i+22}$
  $T := (A_{i+71} \wedge A_{i+68}) \oplus (A_{i+58} \wedge A_{i+22})$
  $T' := \big((A_{i+71} \oplus A'_{i+71}) \wedge (A_{i+68} \oplus A'_{i+68})\big) \oplus \big((A_{i+58} \oplus A'_{i+58}) \wedge (A_{i+22} \oplus A'_{i+58})\big)$
  for all bit positions $k = 0, 1, \ldots, 63$ such that $D_k = 1$ **do**
    $Y_j := T_k \oplus T'_k$
    $j := j + 1$
  end for
end for
$y := j$

i	$\mathcal{M}_i$	$\mathcal{Y}_i$	q_i
0	–	$\emptyset$	0
1	$\{M_{38}\}$	$\{Y_1, \ldots, Y_{29}\}$	29
2	$\{M_{55}\}$	$\{Y_{43}, \ldots, Y_{48}\}$	6
3	$\{M_0, M_5, M_{46}, M_{52}, M_{54}\}$	$\{Y_0\}$	1
4	$\{M_j \mid j = 3, 4, 6, 9, 21, 36, 39, 40, 42, 45, 49, 50, 53, 56, 57\}$	$\{Y_{31}, \ldots, Y_{36}\}$	6
5	$\{M_{41}, M_{51}, M_{58}, M_{59}, M_{60}\}$	$\{Y_{30}, Y_{51}\}$	2
6	$\{M_j \mid j = 1, 2, 7, 8, 10, 11, 12, 17, 18, 20, 22, 24, 25, 26, 29, 33, 34, 37, 43, 44, 47, 48, 61, 62, 63\}$	$\{Y_{52}, \ldots, Y_{57}\}$	6
7	$\{M_{27}\}$	$\{Y_{37}, \ldots, Y_{42}\}$	6
8	$\{M_{13}, M_{16}, M_{23}\}$	$\{Y_{50}\}$	1
9	$\{M_{35}\}$	$\{Y_{49}\}$	1
10	$\{M_{14}, M_{15}, M_{19}, M_{28}\}$	$\{Y_{58}, Y_{61}\}$	2
11	$\{M_{30}, M_{31}, M_{32}\}$	$\{Y_{59}, Y_{60}, Y_{62} \ldots, Y_{89}\}$	30

Table A.2: Partitioning example for MD6. This table shows the input and output partitionings of the Condition function of MD6 with $r = 16$ rounds for the difference Δ in equation (5.12). Numbers in $\mathcal{M}_i$ are message word indices, whereas numbers in $\mathcal{Y}_i$ are condition bit indices. The total complexity of the attack is dominated by the last step in which 30 condition bits must be satisfied.

Algorithm 6 : Condition function for CubeHash

Inputs: r, b, t, $\Delta = \Delta^0||\ldots||\Delta^{t-1}$, $M = M^0||\ldots||M^{t-1}$, and $V = (V_0, \ldots, V_{31})$.

Outputs: y and $Y = \mathsf{Condition}_\Delta(M, V)$.

1: $(S_0, \ldots, S_{31}) := V$
2: $(S'_0, \ldots, S'_{31}) := (0, \ldots, 0)$
3: $j := 0$
4: **for** t' from 0 to $t-1$ **do**
5: XOR $M^{t'}$ into the first b bytes of the state S {see the footnote on page 89}
6: XOR $\Delta^{t'}$ into the first b bytes of the state S' {see the footnote on page 89}
7: **for** *round* from 1 to r **do**
8: **for** i from 0 to 15 **do**
9: $\alpha := S'_i$, $\beta := S'_{i\oplus 16}$, $A := S_i$, $B := S_{i\oplus 16}$
10: Add S_i into $S_{i\oplus 16}$ but XOR S'_i into $S'_{i\oplus 16}$
11: $D := \alpha \vee \beta$
12: $C := A \oplus B \oplus S_{i\oplus 16}$ {carry word}
13: $T = \big((\alpha \oplus \beta) \wedge C\big) \oplus (\alpha \wedge B) \oplus (\beta \wedge A) \oplus (\alpha \wedge \beta)$ {see equation (5.5) or (5.8)}
14: **for** all bit positions $k = 0, 1, \ldots, 31$ such that $D_k = 1$ **do**
15: $Y_j := T_k$
16: $j := j + 1$
17: **end for**
18: **end for**
19: **for** $0 \le i \le 15$ **do** rotate S_i and S'_i to the left by seven bits **end for**
20: **for** $0 \le i \le 7$ **do** swap S_i and $S_{i\oplus 8}$ as well as S'_i and $S'_{i\oplus 8}$ **end for**
21: **for** $0 \le i \le 15$ **do** XOR $S_{i\oplus 16}$ into S_i and $S'_{i\oplus 16}$ into S'_i **end for**
22: **for** $i \in \{16, 17, 20, 21, 24, 25, 28, 29\}$ **do** swap S_i and $S_{i\oplus 2}$ as well as S'_i and $S'_{i\oplus 2}$ **end for**
23: **for** i from 0 to 15 **do**
24: $\alpha := S'_i$, $\beta := S'_{i\oplus 16}$, $A := S_i$, $B := S_{i\oplus 16}$
25: Add S_i into $S_{i\oplus 16}$ but XOR S'_i into $S'_{i\oplus 16}$
26: $D := \alpha \vee \beta$
27: $C := A \oplus B \oplus S_{i\oplus 16}$ {carry word}
28: $T = \big((\alpha \oplus \beta) \wedge C\big) \oplus (\alpha \wedge B) \oplus (\beta \wedge A) \oplus (\alpha \wedge \beta)$ {see equation (5.5) or (5.8)}
29: **for** all bit positions $k = 0, 1, \ldots, 31$ such that $D_k = 1$ **do**
30: $Y_j := T_k$
31: $j := j + 1$
32: **end for**
33: **end for**
34: **for** $0 \le i \le 15$ **do** rotate S_i and S'_i to the left by eleven bits **end for**
35: **for** $i \in \{0, 1, 2, 3, 8, 9, 10, 11\}$ **do** swap S_i and $S_{i\oplus 4}$ as well as S'_i and $S'_{i\oplus 4}$ **end for**
36: **for** $0 \le i \le 15$ XOR $S_{i\oplus 16}$ **do** into S_i and $S'_{i\oplus 16}$ into S'_i **end for**
37: **for** $i \in \{16, 18, 20, 22, 24, 26, 28, 30\}$ **do** swap S_i and $S_{i\oplus 1}$ as well as S'_i and $S'_{i\oplus 1}$ **end for**
38: **end for**{r rounds}
39: **end for**{t iterations}
40: $y := j$

References

[1] RSA Laboratories, PKCS#1: RSA Cryptography Standard (Version 2.1, June 14, 2002). Available at `http://www.rsa.com/rsalabs/node.asp?id=2125`. 16

[2] Verisign, Inc. Status Responder Certificate. Class 3 Public Primary Certification Authority. Serial number: 70:BA:E4:1D:10:D9:29:34:B6:38:CA:7B:03:CC:BA:BF. Issued 1996/01/29, expires 2028/08/02. `http://www.verisign.com/repository/root.html#c3pca`. 16

[3] RC4 algorithm revealed, 1994. See `http://groups.google.com/group/sci.crypt/msg/10a300c9d21afca0`. 8, 28

[4] Martin Albrecht and Carlos Cid. Algebraic Techniques in Differential Cryptanalysis. In *FSE*, pages 193–208, 2009. 23

[5] Ammar Alkassar, Alexander Geraldy, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. Optimized Self-Synchronizing Mode of Operation. In *FSE*, pages 78–91, 2001. 63

[6] Jee Hea An and Mihir Bellare. Constructing VIL-MACsfrom FIL-MACs: Message Authentication under Weakened Assumptions. In *CRYPTO*, pages 252–269, 1999. 15

[7] Frederik Armknecht. Improving Fast Algebraic Attacks. In *FSE*, pages 65–82, 2004. 23

[8] Frederik Armknecht and Matthias Krause. Algebraic Attacks on Combiners with Memory. In *CRYPTO*, pages 162–175, 2003. 23

[9] Frederik Armknecht, Joseph Lano, and Bart Preneel. Extending the Resynchronization Attack. In *Selected Areas in Cryptography*, pages 19–38, 2004. Extended version available at `http://eprint.iacr.org/2004/232`. 58

[10] François Arnault, Thierry P. Berger, and Cédric Lauradoux. F-FCSR Stream Ciphers. In *The eSTREAM Finalists*, pages 170–178. 2008. `http://www.ecrypt.eu.org/stream/ffcsrpf.html`. 11

[11] François Arnault, Thierry P. Berger, and Abdelkader Necer. A New Class of Stream Ciphers Combining LFSR and FCSR Architectures. In *INDOCRYPT*, pages 22–33, 2002. 8, 62

[12] Jean-Philippe Aumasson. Collision for CubeHash-2/120 − 512. mailing list, 4 Dec 2008, 2008, 2007. `http://ehash.iaik.tugraz.at/uploads/a/a9/Cubehash.txt`. 100

[13] Jean-Philippe Aumasson, Eric Brier, Willi Meier, María Naya-Plasencia, and Thomas Peyrin. Inside the Hypercube. In *ACISP*, pages 202–213, 2009. 99

[14] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube Testers and Key Recovery Attacks on

Reduced-Round MD6 and Trivium. In *FSE*, pages 1–22, 2009. 59, 60

[15] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba. In *FSE*, pages 470–488, 2008. 29, 47, 77, 83

[16] J.P. Aumasson, I. Dinur, L. Henzen, W. Meier, and A. Shamir. Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128. *Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS'09)*, 2009. Available at `http://eprint.iacr.org/2009/218`. 59, 60, 72

[17] S. Babbage. A space/time tradeoff in exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection, volume 408 of IEE Conference Publication*, 1995. 19

[18] S. Babbage. Stream ciphers: What does the industry want. In *State of the Art of Stream Ciphers workshop (SASC04)*, 2004. 62

[19] S. Babbage, C. Cannière, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, and M. Robshaw. The eSTREAM portfolio. *eSTREAM, ECRYPT Stream Cipher Project, April 2008.* `http://www.ecrypt.eu.org/stream/`. 11, 62

[20] S. Babbage, C. Cannière, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, and M. Robshaw. The eSTREAM Portfolio (rev. 1). *eSTREAM, ECRYPT Stream Cipher Project, September 2008.* `http://www.ecrypt.eu.org/stream/`. 11, 29, 45, 47, 62

[21] Steve Babbage and Matthew Dodd. The MICKEY Stream Ciphers. In *The eSTREAM Finalists*, pages 191–209. 2008. `http://www.ecrypt.eu.org/stream/mickeypf.html`. 11, 45

[22] Thomas Baignères, Pascal Junod, and Serge Vaudenay. How Far Can We Go Beyond Linear Cryptanalysis? In *ASIACRYPT*, pages 432–450, 2004. 37

[23] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. *Journal of Cryptology*, 21(3):392–429, 2008. 11

[24] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom Functions Revisited: The Cascade Construction and Its Concrete Security. In *FOCS*, pages 514–523, 1996. 15

[25] Mihir Bellare and Thomas Ristenpart. Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In *ASIACRYPT*, pages 299–314, 2006. 15

[26] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993. 12

[27] Mihir Bellare and Phillip Rogaway. Optimal Asymmetric Encryption. In *EUROCRYPT*, pages 92–111, 1994. 12

[28] Côme Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cédric Lauradoux, Marine Minier, Thomas Pornin, and Hervé Sibert. Sosemanuk, a Fast Software-Oriented Stream Cipher. In *The eSTREAM Finalists*, pages 98–118. 2008. `http://www.ecrypt.eu.org/stream/sosemanukpf.html`. 11

[29] Côme Berbain and Henri Gilbert. On the Security of IV Dependent Stream Ciphers. In *FSE*, pages 254–273, 2007. 58

[30] Daniel J. Bernstein. Salsa20 and ChaCha. eSTREAM discussion forum, May 11, 2007. See also `http://www.ecrypt.eu.org/stream/phorum/read.php?1,1085`. 24

[31] Daniel J. Bernstein. Salsa20. Technical Report 2005/025, eSTREAM, ECRYPT Stream Cipher Project, 2005. See also `http://cr.yp.to/snuffle.html`. 24, 29, 33

[32] Daniel J. Bernstein. What output size resists collisions in a XOR of independent expansions? , howpublished = ECRYPT Workshop on Hash Functions, 2007. See also `http://cr.yp.to/rumba20.html`. 29

[33] Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *SASC 2008 – The State of the Art of Stream Ciphers*. ECRYPT, 2008. See also `http://cr.yp.to/chacha.html`. 24, 29, 33

[34] Daniel J. Bernstein. CubeHash specification (2.b.1). Submission to NIST SHA-3 competition., 2008. `http://cubehash.cr.yp.to/`. 24, 77, 89, 90

[35] Daniel J. Bernstein. The Salsa20 Family of Stream Ciphers. In *The eSTREAM Finalists*, pages 84–97. 2008. `http://www.ecrypt.eu.org/stream/salsa20pf.html`. 11, 24

[36] Daniel J. Bernstein. CubeHash parameter tweak: 16 times faster. Submission to NIST SHA-3 competition., 2009. `http://cubehash.cr.yp.to/submission/`. 89

[37] Daniel J. Bernstein. Why haven't cube attacks broken anything? Page of author's website, 2009. `http://cr.yp.to/cubeattacks.html`. 59

[38] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Radiogatun, a belt-and-mill hash function. Presented at Second Cryptographic Hash Workshop, Santa Barbara, August 2006. `http://radiogatun.noekeon.org/`. 22, 76, 77, 83, 85

[39] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge Functions. Presented at Second Cryptographic Hash Workshop, Santa Barbara, August 2006. `http://sponge.noekeon.org/`. 15, 78

[40] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In *EUROCRYPT*, pages 181–197, 2008. 15

[41] Eli Biham. On the applicability of differential cryptanalysis to hash functions. E.I.S.S. Workshop on Cryptographic Hash Functions, Oberwolfach (D), March 25-27, 1992. 22

[42] Eli Biham. New Types of Cryptoanalytic Attacks Using related Keys (Extended Abstract). In *EUROCRYPT*, pages 398–409, 1993. 22

[43] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In *EUROCRYPT*, pages 12–23, 1999. 21

[44] Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In *CRYPTO*, pages 290–305, 2004. 16, 22, 26, 30, 37, 76, 77, 83, 85, 94

[45] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In *EUROCRYPT*, pages 36–57, 2005. 16, 22, 26, 76

[46] Eli Biham, Orr Dunkelman, and Nathan Keller. New Results on Boomerang and Rectangle Attacks. In *FSE*, pages 1–16, 2002. 21

[47] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In *CRYPTO*, pages 2–21, 1990. 20

[48] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *J. Cryptology*, 4(1):3–72, 1991. 20, 58

[49] Eli Biham and Adi Shamir. Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer. In *CRYPTO*, pages 156–171, 1991. 22

[50] Eli Biham and Adi Shamir. Differential Cryptoanalysis of Feal and N-Hash. In *EUROCRYPT*, pages 1–16, 1991. 22

[51] Eli Biham and Adi Shamir. Differential Cryptanalysis of the Full 16-Round DES. In *CRYPTO*, pages 487–496, 1992. 20

[52] Eli Biham and Adi Shamir. *Differential cryptanalysis of the data encryption standard*. Springer-Verlag London, UK, 1993. 20

[53] Alex Biryukov and Christophe De Cannière. Block Ciphers and Systems of Quadratic Equations. In *FSE*, pages 274–289, 2003. 23

[54] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved Time-Memory Trade-Offs with Multiple Data. In *Selected Areas in Cryptography*, pages 110–127, 2005. 6, 20

[55] Alex Biryukov and Adi Shamir. Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers. In *ASIACRYPT*, pages 1–13, 2000. 20

[56] Alex Biryukov and Adi Shamir. Structural Cryptanalysis of SASAS. In *EUROCRYPT*, pages 394–405, 2001. 22, 58

[57] Alex Biryukov, Adi Shamir, and David Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *FSE*, pages 1–18, 2000. 11

[58] Alex Biryukov and David Wagner. Advanced Slide Attacks. In *EUROCRYPT*, pages 589–606, 2000. 60

[59] John Black, Martin Cochran, and Thomas Shrimpton. On the Impossibility of Highly-Efficient Blockcipher-Based Hash Functions. In *EUROCRYPT*, pages 526–541, 2005. 15, 17

[60] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In *CRYPTO*, pages 320–335, 2002. 14

[61] Bluetooth™. Bluetooth Specification, version 1.2, pp. 903–948, November, 2003. Available at `http://www.bluetooth.org`. 8

[62] Martin Boesgaard, Mette Vesterager, and Erik Zenner. The Rabbit Stream Cipher. In *The eSTREAM Finalists*, pages 69–83. 2008. `http://www.ecrypt.eu.org/stream/rabbitpf.html`. 11

[63] Yuri L. Borissov, Svetla Nikova, Bart Preneel, and Joos Vandewalle. 58

[64] Marc Briceno, Ian Goldberg, and David Wagner. A pedagogical implementation of the GSM A5/1 and A5/2 voice privacy encryption algorithms, 1999. See `http://cryptome.org/gsm-a512.htm`. 8

[65] Eric Brier, Shahram Khazaei, Willi Meier, and Thomas Peyrin. Linearization Framework for Collision Attacks: Application to CubeHash and MD6. In *ASIACRYPT*, pages 560–577, 2009. 75

[66] Eric Brier and Thomas Peyrin. Cryptanalysis of CubeHash. In *ACNS*, pages 354–368, 2009. 100

[67] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004. 12

[68] Christophe De Cannière. Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles. In *ISC*, pages 171–186, 2006. 24, 45, 47, 55

[69] Christophe De Cannière, Özgül Küçük, and Bart Preneel. Analysis of Grain's Initialization Algorithm. In *AFRICACRYPT*, pages 276–289, 2008. 60

[70] Christophe De Cannière, Florian Mendel, and Christian Rechberger. Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In *Selected Areas in Cryptography*, pages 56–73, 2007. 22, 76

[71] Christophe De Cannière and Bart Preneel. Trivium specifications. Technical Report 2005/001, eSTREAM, ECRYPT Stream Cipher Project, 2005. 24

[72] Christophe De Cannière and Bart Preneel. Trivium. In *The eSTREAM Finalists*, pages 244–266. 2008. `http://www.ecrypt.eu.org/stream/triviumpf.html`. 11, 24

[73] Christophe De Cannière and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In *ASIACRYPT*, pages 1–20, 2006. 22, 76, 77, 83, 85

[74] Anne Canteaut and Florent Chabaud. A New Algorithm for Finding Minimum-Weight Words in a Linear Code: Application to McEliece's Cryptosystem and to Narrow-Sense BCH Codes of Length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, 1998. 83, 92, 94

[75] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In

CRYPTO, pages 56–71, 1998. 16, 22, 75, 76, 82

[76] Donghoon Chang, Sangjin Lee, Mridul Nandi, and Moti Yung. Indifferentiable Security Analysis of Popular Hash Functions with Prefix-Free Padding. In *ASIACRYPT*, pages 283–298, 2006. 15

[77] J. Chen, B. Wang, and Y. Hu. A new method for resynchronization attack. *Journal of Electronics (China)*, 23(3):423–427, 2006. 58

[78] C. Cid, S. Murphy, and M. Robshaw. *Algebraic aspects of the advanced encryption standard*. Springer New York, 2006. 23

[79] D. Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM Journal of Research and Development*, 38(3):243–250, 1994. 20

[80] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In *CRYPTO*, pages 430–448, 2005. 15

[81] Nicolas Courtois. Fast Algebraic Attacks on Stream Ciphers with Linear Feedback. In *CRYPTO*, pages 176–194, 2003. 23

[82] Nicolas Courtois. Algebraic Attacks on Combiners with Memory and Several Outputs. In *ICISC*, pages 3–20, 2004. 23

[83] Nicolas Courtois and Willi Meier. Algebraic Attacks on Stream Ciphers with Linear Feedback. In *EUROCRYPT*, pages 345–359, 2003. 23

[84] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In *ASIACRYPT*, pages 267–287, 2002. 23

[85] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, USA, 1991. 36, 37, 53

[86] Y. Crama and P.L. Hammer. *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*. Cambridge University Press, 2010. 51

[87] Paul Crowley. Truncated Differential Cryptanalysis of Five Rounds of Salsa20. In *SASC 2006 – Stream Ciphers Revisited*, 2006. 30

[88] Joan Daemen. Cipher and Hash Function Design. Strategies based on Linear and Differential Cryptanalysis. *PhD thesis, Katholieke Universiteit Leuven*. 62

[89] Joan Daemen, René Govaerts, and Joos Vandewalle. A Practical Approach to the Design of High Speed Self-Synchronizing Stream Ciphers. *Singapore ICCS/ISITA'92*, pages 279–293, 1992. Available at: `https://www.cosic.esat.kuleuven.be/publications/article-134.pdf`. 8, 62

[90] Joan Daemen, René Govaerts, and Joos Vandewalle. Resynchronization Weaknesses in Synchronous Stream Ciphers. In *EUROCRYPT*, pages 159–167, 1993. 58

[91] Joan Daemen and Paris Kitsos. The Self-synchronizing Stream Cipher Mosquito. Technical Report

2005/018, eSTREAM, ECRYPT Stream Cipher Project, 2005. See also `http://www.ecrypt.eu.org/stream/ciphers/mosquito/mosquito.pdf`. 8, 62

[92] Joan Daemen and Paris Kitsos. The Self-synchronizing Stream Cipher Moustique. In *The eSTREAM Finalists*, pages 210–223. 2008. 8, 62

[93] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The Block Cipher Square. In *FSE*, pages 149–165, 1997. 22, 58

[94] Joan Daemen, Joseph Lano, and Bart Preneel. Chosen ciphertext attack on SSS. *SASC 2006 – The State of the Art of Stream Ciphers.*, pages 45–51, 2006. Available at: `https://www.ecrypt.eu.org/stream/papersdir/044.pdf`. 8, 62

[95] Watanabe Dai. Collisions for CubeHash-1/45 and CubeHash-2/89. Available online, 2008. `http://www.cryptopp.com/sha3/cubehash.pdf`. 100

[96] Ivan Damgård. Collision Free Hash Functions and Public Key Signature Schemes. In *EUROCRYPT*, pages 203–216, 1987. 13

[97] Ivan Damgård. A Design Principle for Hash Functions. In *CRYPTO*, pages 416–427, 1989. 13, 14, 15

[98] Bert den Boer and Antoon Bosselaers. An Attack on the Last Two Rounds of MD4. In *CRYPTO*, pages 194–203, 1991. 16, 22

[99] Bert den Boer and Antoon Bosselaers. Collisions for the Compressin Function of MD5. In *EUROCRYPT*, pages 293–304, 1993. 16, 22

[100] Itai Dinur and Adi Shamir. Cube Attacks on Tweakable Black Box Polynomials. In *EUROCRYPT*, pages 278–299, 2009. 22, 58, 59, 60, 64, 72

[101] Hans Dobbertin. Cryptanalysis of MD4. In *FSE*, pages 53–69, 1996. 16, 22

[102] Yevgeniy Dodis, Rosario Gennaro, Johan Håstad, Hugo Krawczyk, and Tal Rabin. Randomness Extraction and Key Derivation Using the CBC, Cascade and HMAC Modes. In *CRYPTO*, pages 494–510, 2004. 15

[103] Yevgeniy Dodis, Krzysztof Pietrzak, and Prashant Puniya. A New Mode of Operation for Block Ciphers and Length-Preserving MACs. In *EUROCRYPT*, pages 198–219, 2008. 15

[104] Yevgeniy Dodis, Leonid Reyzin, Ronald L. Rivest, and Emily Shen. Indifferentiability of Permutation-Based Compression Functions and Tree-Based Modes of Operation, with Applications to MD6. In *FSE*, pages 104–121, 2009. 15

[105] Orr Dunkelman and Nathan Keller. Treatment of the Initial Value in Time-Memory-Data Tradeoff Attacks on Stream Ciphers. *Information Processing Letters*, 107(5):133–137, 2008. 6, 20

[106] ECRYPT. eSTREAM, the ECRYPT Stream Cipher Project. See `http://www.ecrypt.eu.org/stream`. 11, 24, 29, 62

[107] Carlos Cid (Ed.), Martin Albrecht, Daniel Augot, Anne Canteaut, and Ralf-Philipp Weinmann. D.STVL.7 – Algebraic cryptanalysis of symmetric primitives. ECRYPT Report, July 2008. Available at `https://www.ecrypt.eu.org/ecrypt1/documents/D.STVL.7.pdf`. 23

[108] Patrik Ekdahl and Thomas Johansson. Another attack on A5/1. *IEEE Transactions on Information Theory*, 49(1):284–289, 2003. 58

[109] Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A Framework for Chosen IV Statistical Analysis of Stream Ciphers. In *INDOCRYPT*, pages 268–281, 2007. 46, 49, 50, 55, 57, 58, 64

[110] H. Feistel. Cryptography and Computer Privacy. *Scientific american*, 228(5):15–23, 1973. 30, 37

[111] Eric Filiol. A New Statistical Testing for Symmetric Ciphers and Hash Functions. In *ICICS*, pages 342–353, 2002. 46, 49, 50, 58

[112] PUB FIPS. 81, DES Modes of Operation. *Issued December*, 2, 1980. 63

[113] Simon Fischer, Shahram Khazaei, and Willi Meier. Chosen IV Statistical Analysis for Key Recovery Attacks on Stream Ciphers. In *AFRICACRYPT*, pages 236–245, 2008. 45, 68

[114] Simon Fischer, Willi Meier, Côme Berbain, Jean-François Biasse, and Matthew J. B. Robshaw. Non-randomness in eSTREAM Candidates Salsa20 and TSC-4. In *INDOCRYPT*, pages 2–16, 2006. 30

[115] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. In *Selected Areas in Cryptography*, pages 1–24, 2001. 58

[116] Réjane Forré. The Strict Avalanche Criterion: Spectral Properties of Boolean Functions and an Extended Definition. In *CRYPTO*, pages 450–468, 1988. 37

[117] Pierre-Alain Fouque, Gwenaëlle Martinet, and Guillaume Poupard. Practical Symmetric On-Line Encryption. In *FSE*, pages 362–375, 2003. 63

[118] Thomas Fuhr and Thomas Peyrin. Cryptanalysis of RadioGatún. In *FSE*, pages 122–138, 2009. 77, 83, 85

[119] Taher El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO*, pages 10–18, 1984. 4

[120] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman & Co Ltd, January 1979. 4

[121] Jovan Dj. Golic. Cryptanalysis of Alleged A5 Stream Cipher. In *EUROCRYPT*, pages 239–255, 1997. 19

[122] Jovan Dj. Golic, Vittorio Bagini, and Guglielmo Morgari. Linear Cryptanalysis of Bluetooth Stream Cipher. In *EUROCRYPT*, pages 238–255, 2002. 58

[123] Jovan Dj. Golic and Guglielmo Morgari. On the Resynchronization Attack. In *FSE*, pages 100–110, 2003. 58

[124] C. G. Günther. Alternating Step Generators Controlled by De Bruijn Sequences. In *EUROCRYPT*, pages 5–14, 1987. 28

[125] Martin Hell and Thomas Johansson. Breaking the F-FCSR-H Stream Cipher in Real Time. In *ASIACRYPT*, pages 557–569, 2008. 11

[126] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. A Stream Cipher Proposal: Grain-128. In *Information Theory, 2006 IEEE International Symposium on*, pages 1614–1618, July 2006. 24, 45, 47, 57

[127] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain Family of Stream Ciphers. In *The eSTREAM Finalists*, pages 179–190. 2008. `http://www.ecrypt.eu.org/stream/grainpf.html`. 11, 24, 45

[128] Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *IJWMC*, 2(1):86–93, 2007. 11, 24, 45

[129] M. Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980. 19

[130] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, 2003. 12

[131] Howard M. Heys. An Analysis of the Statistical Self-Synchronization of Stream Ciphers. In *INFOCOM*, pages 897–904, 2001. 63

[132] Thomas Hodanek. Analysis of Reduced MD6. In *Western European Workshop on Research in Cryptology, WEWoRC*, July 2009. 107

[133] Jin Hong and Palash Sarkar. New Applications of Time Memory Data Tradeoffs. In *ASIACRYPT*, pages 353–372, 2005. 6, 20

[134] Sebastiaan Indesteege and Bart Preneel. Practical Collisions for EnRUPT. In *FSE*, pages 246–259, 2009. 76

[135] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In *CRYPTO*, pages 306–316, 2004. 15, 17

[136] Antoine Joux and Frédéric Muller. A Chosen IV Attack Against Turing. In *Selected Areas in Cryptography*, pages 194–207, 2003. 58

[137] Antoine Joux and Frédéric Muller. Loosening the KNOT. In *FSE*, pages 87–99, 2003. 8, 62

[138] Antoine Joux and Frédéric Muller. Two Attacks Against the HBB Stream Cipher. In *FSE*, pages 330–341, 2005. 8, 62

[139] Antoine Joux and Frédéric Muller. Chosen-Ciphertext Attacks Against MOSQUITO. In *FSE*, pages 390–404, 2006. 8, 62

[140] Antoine Joux and Thomas Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In *CRYPTO*, pages 244–263, 2007. 83

[141] Oliver Jung and Christoph Ruland. Encryption with Statistical Self-Synchronization in Synchronous

Broadband Networks. In *CHES*, pages 340–352, 1999. 63

[142] B. Kaliski. The MD2 message-digest algorithm. *Internet Request for Comments (RFC) 1319*, April 1992. 15

[143] J.B. Kam and G.I. Davida. Structured design of substitution-permutation encryption networks. *IEEE Transactions on Computers*, 100(28):747–753, 1979. 30, 37

[144] Emilia Käsper, Vincent Rijmen, Tor E. Bjørstad, Christian Rechberger, Matthew J. B. Robshaw, and Gautham Sekar. Correlated Keystreams in Moustique. In *AFRICACRYPT*, pages 246–257, 2008. 8, 62

[145] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In *EUROCRYPT*, pages 183–200, 2006. 15, 17

[146] John Kelsey, Tadayoshi Kohno, and Bruce Schneier. Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent. In *FSE*, pages 75–93, 2000. 21

[147] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In *EUROCRYPT*, pages 474–490, 2005. 15, 17

[148] Shahram Khazaei, Simon Knellwolf, Willi Meier, and Deian Stefan. Improved Linear Differential Attacks on CubeHash. In *AFRICACRYPT*, pages ??–??, 2010. 75, 92, 94

[149] Shahram Khazaei and Willi Meier. New Directions in Cryptanalysis of Self-Synchronizing Stream Ciphers. In *INDOCRYPT*, pages 15–26, 2008. 8, 59, 61

[150] Dmitry Khovratovich. Nonrandomness of the 33-round MD6. Presented at the rump session of FSE'09, 2009. Slides are available online at `http://fse2009rump.cr.yp.to/`. 107

[151] Vlastimil Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105, 2006. `http://eprint.iacr.org/2006/105`. 22, 76, 83, 85

[152] Alexander Klimov and Adi Shamir. New Applications of T-Functions in Block Ciphers and Hash Functions. In *FSE*, pages 18–31, 2005. 8, 24, 61, 62, 63, 65, 66, 67

[153] Lars R. Knudsen. DEAL — a 128-bit block cipher. Technical Report 151, Department of Informatics, University of Bergen, Norway, 1998. 21

[154] Lars R. Knudsen. Cryptanalysis of LOKI91. In *AUSCRYPT*, pages 196–208, 1992. 22

[155] Lars R. Knudsen. Truncated and Higher Order Differentials. In *FSE*, pages 196–211, 1994. 21, 23, 58

[156] Lars R. Knudsen and John Erik Mathiassen. Preimage and Collision Attacks on MD2. In *FSE*, pages 255–267, 2005. 17

[157] Lars R. Knudsen, John Erik Mathiassen, Frédéric Muller, and Søren S. Thomsen. Cryptanalysis of MD2. *J. Cryptology*, 23(1):72–90, 2010. 17

[158] Lars R. Knudsen and David Wagner. Integral Cryptanalysis. In *FSE*, pages 112–127, 2002. 22, 58

[159] Xuejia Lai. Higher order derivatives and differential cryptanalysis. In U. Maurer R.E. Blahut, D.J. Costello Jr and T. Mittelholzer, editors, *Communications and cryptography – two sides of one tapestry*, pages 227–233. Kluwer Academic Publishers, 1994. Scanned copy online at `http://cr.yp.to/cubeattacks.html`. 59

[160] Xuejia Lai and James L. Massey. Markov Ciphers and Differentail Cryptanalysis. In *EUROCRYPT*, pages 17–38, 1991. 21

[161] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part III – Algorithms, Modes, and Identifiers. *Internet Request for Comments (RFC) 1115*, August 1989. 15

[162] Helger Lipmaa and Shiho Moriai. Efficient Algorithms for Computing Differential Properties of Addition. In *FSE*, pages 336–350, 2001. 79, 104

[163] Yi Lu, Willi Meier, and Serge Vaudenay. The Conditional Correlation Attack: A Practical Attack on Bluetooth Encryption. In *CRYPTO*, pages 97–117, 2005. 11, 58

[164] Yi Lu and Serge Vaudenay. Cryptanalysis of Bluetooth Keystream Generator Two-Level E0. In *ASIACRYPT*, pages 483–499, 2004. 58

[165] Stefan Lucks. Attacking Seven Rounds of Rijndael under 192-bit and 256-bit Keys. In *AES Candidate Conference*, pages 215–229, 2000. 22, 58

[166] Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In *ASIACRYPT*, pages 474–494, 2005. 15, 17, 78

[167] Hideki Imai Makoto Sugita, Mitsuru Kawazoe. Gröbner Basis Based Cryptanalysis of SHA-1. Cryptology ePrint Archive, Report 2006/098, 2006. `http://eprint.iacr.org/`. 22, 26, 76, 83, 85

[168] Stéphane Manuel and Thomas Peyrin. Collisions on SHA–0 in One Hour. In *FSE*, pages 16–35, 2008. 22, 76, 83

[169] S. M. Matyas, C. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985. 14

[170] Ueli M. Maurer. New Approaches to the Design of Self-Synchronizing Stream Ciphers. In *EUROCRYPT*, pages 458–471, 1991. 62

[171] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In *TCC*, pages 21–39, 2004. 15

[172] Ueli M. Maurer and Johan Sjödin. Single-Key AIL-MACs from Any FIL-MAC. In *ICALP*, pages 472–484, 2005. 15

[173] Alexander Maximov and Dmitry Khovratovich. New State Recovery Attack on RC4. In *CRYPTO*, pages 297–316, 2008. 11

[174] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone.

Handbook of Applied Cryptography. CRC Press, 2001. 1, 14, 15, 78

[175] Ralph C. Merkle. One Way Hash Functions and DES. In *CRYPTO*, pages 428–446, 1989. 14, 15

[176] Joydip Mitra. A Near-Practical Attack Against B Mode of HBB. In *ASIACRYPT*, pages 412–424, 2005. 8, 62

[177] Shoji Miyaguchi, Masahiko Iwata, and Kazuo Ohta. New 128-bit hash function. In *Proc. 4th International Joint Workshop on Computer Communications, Tokyo, Japan*, pages 279–288, 1989. 14

[178] Frédéric Muller. The MD2 Hash Function Is Not One-Way. In *ASIACRYPT*, pages 214–229, 2004. 17

[179] Yusuke Naito, Yu Sasaki, Takeshi Shimoyama, Jun Yajima, Noboru Kunihiro, and Kazuo Ohta. Improved Collision Search for SHA–0. In *ASIACRYPT*, pages 21–36, 2006. 22, 76, 83, 94

[180] Moni Naor and Moti Yung. Universal One-Way Hash Functions and their Cryptographic Applications. In *STOC*, pages 33–43, 1989. 13

[181] National Institute of Science and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Federal Register, 72(112), November 2007. See `http://csrc.nist.gov/groups/ST/hash/sha-3/`. 11, 17, 24, 77, 89

[182] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the Advanced Encryption Standard (AES). *Journal of Research of National Institute of Science and Technology, volume 106, pages 511–576, 2001. Available at:* `http://csrc.nist.gov/archive/aes/`. 11, 17

[183] NESSIE. New European Schemes for Signatures, Integrity and Encryption. See `http://www.cosic.esat.kuleuven.be/nessie/`. 11

[184] Karsten Nohl and Chris Paget. GSM: SRSLY? The 26th Chaos Communication Congress (26C3), 27-30 December 2009, Berlin, Germany. See `http://events.ccc.de/congress/2009/Fahrplan/events/3654.en.html`. 11

[185] National Bureau of Standards. Data Encryption Standard (DES). *Federal Information Processing Standards Publication (FIPS PUB) 46. , January 1977.* 5

[186] National Institute of Standards and Technology/U.S. Department of Commerce. Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication (FIPS PUB) 197, November 2001.* 5

[187] National Institute of Standards and Technology. *FIPS PUB 180: Secure Hash Standard.* May 1993. 16

[188] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard.* April 1995. 14, 16

[189] National Institute of Standards and Technology. *FIPS PUB 180-2: Secure Hash Standard.* August 2002. 14, 16, 134

[190] National Institute of Standards and Technology. *Change Notice for FIPS PUB 180-2 [189]*. February 2004. 14, 16

[191] Sean O'Neil. Algebraic Structure Defectoscopy. Cryptology ePrint Archive, Report 2007/378, 2007. `http://eprint.iacr.org/2007/378`. See also `http://www.defectoscopy.com`. 46, 49, 50, 57, 58

[192] G. Paul and S. Maitra. On biases of permutation and keystream bytes of RC4 towards the secret key. *Cryptography and Communications*, 1(2):225–268, 2009. 11

[193] Sylvain Pelissier. Cryptanalysis of Reduced Word Variants of Salsa. In *Western European Workshop on Research in Cryptology, WEWoRC*, July 2009. 44

[194] Thomas Peyrin. Cryptanalysis of Grindahl. In *ASIACRYPT*, pages 551–567, 2007. 77, 83, 85

[195] Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Exploiting Coding Theory for Collision Attacks on SHA–1. In *IMA Int. Conf.*, pages 78–95, 2005. 76, 82, 92, 94

[196] Bart Preneel. Analysis and design of cryptographic hash functions. *PhD thesis, Katholieke Universiteit Leuven*. 17

[197] Bart Preneel, Rene Govaerts, and Joos Vandewalle. Differential cryptanalysis of hash functions based on block ciphers. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 183–188, New York, NY, USA, 1993. ACM. 22

[198] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In *CRYPTO*, pages 368–378, 1993. 14

[199] Bart Preneel, Marnix Nuttin, Vincent Rijmen, and Johan Buelens. Cryptanalysis of the CFB Mode of the DES with a Reduced Number of Rounds. In *CRYPTO*, pages 212–223, 1993. 63

[200] Norman Proctor. A Self-Synchronizing Cascaded Cipher System With Dynamic Control of Error-Propagation. In *CRYPTO*, pages 174–190, 1984. 62

[201] Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search? Application to DES (Extended Summary). In *EUROCRYPT*, pages 429–434, 1989. 19

[202] A. Regenscheid, R. Perlner, S. Chang, J. Kelsey, M. Nandi, and S. Paul. Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition. National Institute of Science and Technology, NISTIR 7620, September, 2009. See `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/`. 17

[203] Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In *CT-RSA*, pages 58–71, 2005. 76, 82

[204] Ronald L. Rivest. The MD4 Message Digest Algorithm. In *CRYPTO*, pages 303–311, 1990. 16

[205] Ronald L. Rivest. The MD5 message-digest algorithm. *Internet Request for Comments (RFC) 1321*, April 1992. 14, 16

[206] Ronald L. Rivest, Benjamin Agre, Daniel V. Bailey, Christopher Crutchfield, Yevgeniy Dodis, Kermin Elliott Fleming Asif Khan, Jayant Krishnamurthy, Yuncheng Lin, Leo Reyzin, Emily Shen, Jim Sukha, Drew Sutherland, Eran Tromer, and Yiqun Lisa Yin. The MD6 hash function — a proposal to NIST for SHA–3. Submission to NIST SHA-3 competition., 2008. `http://groups.csail.mit.edu/cis/md6/`. 24, 77, 104, 105, 106, 107

[207] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. 4

[208] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In *FSE*, pages 371–388, 2004. Extended version available at: `http://web.cecs.pdx.edu/~teshrim/`. 13

[209] Phillip Rogaway and John P. Steinberger. Constructing Cryptographic Hash Functions from Fixed-Key Blockciphers. In *CRYPTO*, pages 433–450, 2008. 17

[210] Phillip Rogaway and John P. Steinberger. Security/Efficiency Tradeoffs for Permutation-Based Hashing. In *EUROCRYPT*, pages 220–236, 2008. 17

[211] N. Rogier and Pascal Chauvaud. MD2 Is not Secure without the Checksum Byte. *Des. Codes Cryptography*, 12(3):245–251, 1997. 17

[212] Bart Van Rompay, Alex Biryukov, Bart Preneel, and Joos Vandewalle. Cryptanalysis of 3-Pass HAVAL. In *ASIACRYPT*, pages 228–245, 2003. 22

[213] Gregory Rose, Philip Hawkes, Michael Paddon, and Miriam Wiggers de Vries. Primitive Specifications for SSS. Technical Report 2005/028, eSTREAM, ECRYPT Stream Cipher Project, 2005. See also `http://www.ecrypt.eu.org/stream/ciphers/sss/sss.pdf`. 8, 62

[214] Markku-Juhani O. Saarinen. Chosen-IV Statistical Attacks on eStream Ciphers. In *SECRYPT*, pages 260–266, 2006. 46, 49, 50, 58

[215] Palash Sarkar. Hiji-bij-bij: A New Stream Cipher with a Self-synchronizing Mode of Operation. In *INDOCRYPT*, pages 36–51, 2003. 8, 62

[216] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. Wiley-India, 2007. 8, 28

[217] Bruce Schneier. *Secrets & lies: digital security in a networked world*. John Wiley & Sons, Inc. New York, NY, USA, 2000. 2

[218] Claude Shannon. Communication Theory of Secrecy Systems. *The Bell System Technical Journal*, 28(4):656–715, October 1949. 3, 23

[219] Victor Shoup. OAEP Reconsidered. *J. Cryptology*, 15(4):223–249, 2002. 12

[220] Thomas Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications.

IEEE Transactions on Information Theory, 30(5):776–, 1984. 37

[221] Thomas Siegenthaler. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Trans. Computers*, 34(1):81–85, 1985. 30

[222] Thomas Siegenthaler. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *IEEE Transactions on Computers*, 34(1):81–85, 1985. 36, 37

[223] Martijn Stam. Beyond Uniformity: Better Security/Efficiency Tradeoffs for Compression Functions. In *CRYPTO*, pages 397–412, 2008. 17

[224] Martijn Stam. Blockcipher-Based Hashing Revisited. In *FSE*, pages 67–83, 2009. Full version available at: `http://eprint.iacr.org/2008/071`. 14

[225] Douglas R. Stinson. *Cryptography: theory and practice*. CRC press, 2006. 3

[226] Makoto Sugita, Mitsuru Kawazoe, Ludovic Perret, and Hideki Imai. Algebraic Cryptanalysis of 58-Round SHA-1. In *FSE*, pages 349–365, 2007. 22, 26, 76, 83, 85

[227] Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki, and Hiroki Nakashima. Differential Cryptanalysis of Salsa20/8. In *SASC 2007 – The State of the Art of Stream Ciphers*, 2007. 30, 39, 42, 43

[228] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1):1–28, 1999. 19

[229] Henk C.A. van Tilborg. *Encyclopedia of cryptography and security*. Springer Verlag, 2005. 18

[230] Gilbert S. Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of the IEEE*, 55:109–115, 1926. 4

[231] Michael Vielhaber. Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack. Cryptology ePrint Archive, Report 2007/413, 2007. `http://eprint.iacr.org/2007/413`. 22, 47, 49, 50, 55, 58, 64

[232] David Wagner. The Boomerang Attack. In *FSE*, pages 156–170, 1999. 21

[233] Xiaoyun Wang. The Collision Attack on SHA-0. In Chinese, 1997. 22, 76

[234] Xiaoyun Wang. The Improved Collision Attack on SHA-0. In Chinese, 1998. 22, 76

[235] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *Rump session of Crypto 2004*, 2004. 76

[236] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In *EUROCRYPT*, pages 1–18, 2005. 16, 22, 26, 76

[237] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *CRYPTO*, pages 17–36, 2005. 17, 22, 76

[238] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Func-

tions. In *EUROCRYPT*, pages 19–35, 2005. 16, 22, 26, 76, 78, 83, 94

[239] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision Search Attacks on SHA-0. In *CRYPTO*, pages 1–16, 2005. 16, 22, 76

[240] A. F. Webster and Stafford E. Tavares. On the Design of S-Boxes. In *CRYPTO*, pages 523–534, 1985. 37

[241] Hongjun Wu. The Stream Cipher HC-128. In *The eSTREAM Finalists*, pages 39–47. 2008. `http://www.ecrypt.eu.org/stream/hcpf.html`. 11

[242] Hongjun Wu and Bart Preneel. Resynchronization Attacks on WG and LEX. In *FSE*, pages 422–432, 2006. 58

[243] Gideon Yuval. How to swindle Rabin. *Cryptologia*, 3(3):187–191, 1979. 19

[244] Bin Zhang, Hongjun Wu, Dengguo Feng, and Feng Bao. Chosen Ciphertext Attack on a New Class of Self-Synchronizing Stream Ciphers. In *INDOCRYPT*, pages 73–83, 2004. 8, 62

Curriculum Vitae

Education

Ph.D. in Computer Science
Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
July 2006–June 2010

- Thesis: "Neutrality-Based Symmetric Cryptanalysis"
- Supervisors: Prof. Arjen K. Lenstra and Dr. Willi Meier

M.Sc. in Communication Engineering
Sharif University of Technology, Tehran, Iran
Sept. 2002–Nov. 2004

- Thesis: "Cryptanalysis of Stream Ciphers"
- Supervisors: Prof. Mahmoud Salmasizadeh and Mr. Javad Mohajeri

B.Sc. in Electrical Engineering
Sharif University of Technology, Tehran, Iran
Sept. 1998– Aug. 2002

Refereed Papers

1. S. Khazaei, S. Knellwolf, W. Meier and D. Stefan, "Improved Linear Differential Attacks on CubeHash". In the proceedings of AFRICACRYPT 2010, LNCS 6055, pp. 407–418 (2010).
2. E. Brier, S. Khazaei, W. Meier and T. Peyrin, "Linearization Framework for Collision Attacks: Application to CubeHash and MD6". ASIACRYPT 2009, LNCS 5912, pp. 560–577 (2009).

3. S. Khazaei and W. Meier, "On Reconstruction of RC4 Keys from Internal States". Mathematical Methods in Computer Science MMICS 2008, LNCS 5393, pp. 179–189 (2008).

4. S. Khazaei and W. Meier, "New Directions in Cryptanalysis of Self-Synchronizing Stream Ciphers". INDOCRYPT 2008, LNCS 5365, pp. 15–26 (2008).

5. S. Fischer, S. Khazaei and W. Meier, "Chosen IV Statistical Analysis for Key Recovery Attacks on Stream Ciphers". In the proceedings of AFRICACRYPT 2008, LNCS 5023, pp. 236–245 (2008).

6. J.-Ph. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger, "New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba". In the proceedings of Fast Software Encryption FSE 2008, LNCS 5086, pp. 470–488 (2008).

7. S. Khazaei, S. Fischer and W. Meier, "Reduced Complexity Attacks on the Alternating Step Generator". In the proceedings of Selected Areas in Cryptography SAC 2007, LNCS 4876, pp. 1–16 (2007).

8. T. Helleseth, C. J.A. Jansen, S. Khazaei and A. Kholosha, "Security of Jump Controlled Sequence Generators for Stream Ciphers". In the proceedings of Sequences and Their Applications (SETA'06), LNCS 4086, pp. 141–152 (2006).

9. M. Hasanzadeh, S. Khazaei and A. Kholosha, "On IV Setup of Pomaranch". In State of Art of Stream Ciphers (SASC'06) pp. 7–12 (2006).

10. Y. Tsunoo, T. Saito, M. Shigeri, T. Suzaki, H. Ahmadi, T. Eghlidos, and S. Khazaei, "Evaluation of SOSEMANUK with regard to guess-and-determine attacks". In State of Art of Stream Ciphers (SASC'06) pp. 25–34 (2006).

11. M. Hassanzadeh, E. Shakour, S. Khazaei, "Improved Cryptanalysis of Polar Bear". In State of Art of Stream Ciphers (SASC'06) pp. 154–160 (2006).

12. M. Kiaei, S. Ghaemmaghami, S. Khazaei, "Efficient Fully Format Compliant Selective Scrambling Methods for Compressed Video Streams". Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW'06), Guadeloupe, French Caribbean (2006).

Reports

1. S. Khazaei, M. Hasanzadeh and M. Kiaei, "Linear Sequential Circuit Approximation of Grain and Trivium Stream Ciphers". Cryptology ePrint Archive, Report 2006/141, (2006).
2. H. Ahmadi, T. Eghlidos and S. Khazaei, "Improved guess and determine attack on SOSEMANUK", eSTREAM, ECRYPT Stream Cipher Project, Report 2005/085 (2005).

Invited Talks

1. "Linearization Framework for Collision Attacks". Early Symmetric Crypto (ESC) seminar, 11-15 January 2010. Remich, Luxembourg.
2. "Key recovery Attack on Interactable Keyed Functions". Dagstuhl Symmetric Cryptography Seminar, Dagstuhl, Germany, 11-16 January 2009. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany.
3. "eSTREAM, the revival of stream ciphers". Invited speaker for the Iranian Society of Cryptology, 11 May 2005. Sharif University of Technology, Tehran, Iran.

Paper Reviews

1. Conferences: Asiacrypt'06, SAC'07, Eurocrypt'08, FSE'08, SAC'08, Asiacrypt'08, Indocrypt'08, FSE'09, Asiacrypt'09, Eurocrypt'10, FSE'10.
2. Journals: International Journal of Applied Cryptography (IJACT).