EPFL

Thèse n° 8259

Serial Lightweight Implementation Techniques for Block Ciphers

Présentée le 22 juillet 2021

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

pour l'obtention du grade de Docteur ès Sciences

par

Muhammed Fatih BALLI

Acceptée sur proposition du jury

Prof. P. Ienne, président du jury
Prof. S. Vaudenay, directeur de thèse
Prof. L. Batina, rapporteuse
Prof. T. Peyrin, rapporteur
Prof. D. Atienza Alonso, rapporteur

École polytechnique fédérale de Lausanne

2021

Acknowledgements

Without a number of people's help and support, I would not have been able to reach where I am today, presenting my work and defending this thesis.

I would like to start by expressing my gratitude to my supervisor Prof. Serge Vaudenay. Given my lack of theoretical background, during my first few years, I have had to rely on his guidance, which he graciously provided. I thank him for his time and patience. Without his supervision, I would not have learned how to express ideas formally, how to be rigorous and concise in writing, and how to evaluate things from a more formal and objective perspective. Thanks to his research attitude that allows students to be autonomous, I enjoyed the freedom of working on unrelated topics, collaborate with various people, and learned quite a lot during this repetitive process. I also thank him for encouraging us to take initiatives, such as coming up with a research idea, improving our presentation skills with weekly lab meetings, applying for a research grant, and supervising projects, all of which require a demanding effort, but leads to personal academic growth.

Since I joined LASEC in 2016, it has hosted a number of bright and interesting colleagues, to whom I also owe my gratitude.

With my arrival, it was Sonia, Damian and Handan who welcomed me in the lab. Many thanks to Sonia for sharing her various collection of teas accompanied with boiled water in the morning, her interesting conversations exclusively happening on doorsteps and gathering the folks together even after her graduation. Many thanks to Handan for welcoming me in Lausanne, for having introduced me so many Turkish friends, making me feel like I am home. And thanks to Damian, for his hilarious jokes, well-played pranks, his cheerfulness, after-lunch chocolates, but most importantly nerdy jokes and being a "man of culture" with a solid knowledge of internet meme collection.

Many thanks to Gwangbae, who has been there countless times to save the day, when me and my colleagues are in need of something. He has been there not only as a friend and colleague who started at the same year with me, but also someone to advise me on how to handle various lab chores, be it resolving issue on the lab's web page, or regular T.A. duties. He was our go-to guy, and his shoes were definitely hard to fill after his departure. Also, thanks for all the chocolate, biscuits and Korean malang cow candy that you offered us. Thanks for your witty jokes, your kind rejection to any invitation you receive, and introducing us with your *circle theory*.

My deepest gratitude to Subhadeep, for having introduced me to a research topic at

the intersection of cryptography and engineering, and for providing funding on the energy-efficient cryptography project, which eventually became the foundation of this thesis. Thank you for your guidance in this research direction, and for being always available for discussion and brainstorming, regardless of the time of the day. On a more personal level, thank you for being so easy-going, humble and mature in character. Thanks for storytelling us, sometimes multiple times, your countless events involving bicycle accidents, getting a frostbite while running in the cold, a collection of weirdly transcribed Danish words, your knee injuries, your naughty dog, your awkward memories from past conferences. Thank you for being in the lab in the late hours, and keeping an active working environment. Not to mention being so easily distracted and having the-least-important-thing-comes-first prioritization algorithm among tasks.

The very first time I arrived to lab in 2016, it was Betül in INF 238 who welcomed me to the lab. Over the years, she has been a very good office mate as postdoc, a friend to talk to about literally anything, be it in Turkish or English. Her ideas and mood, changing sometimes swiftly and something to be cautious of, has brought lots of joy and laughter to the lab. I especially thank her for following me down from *le Grammont* on a route which is too steep to be a hiking path. Thanks for being the founding member of the *intern destroying hikes* too.

I cannot simply forget all the times Martine was there to help me out, when I found myself in trouble. She has been, to me and to my colleagues, a gracious Swiss host, a friendly adviser to understand the *Suisse Romande* culture and navigate ourselves with our daily problems. Her kindness I will surely miss after I leave the lab. I also thank her for enduring my basic French with a broken accent.

I would like to thank Khashayar for bringing a whole lot of black humor into our lab, and for his energetic spirit. I thank him for encouraging me to try mountain biking, even though it ended with an injury. I thank Loïs for taking some of our mean jokes on his beloved *Neuchâtel Xamax* so lightly, and helping us understand the particular details and differences on the Neuchâtel culture. Thanks to Simone Colombo, from the canton Ticino, who is almost always there with brutal honesty, humorous critics and mostly rightful complaints. Thanks to Bénédikt for his rigorous attitude in writing. Thanks to Hailun for her kindness, friendship and homemade desert she brought to the lab. Thanks to Andy, for his very disciplined work ethic, for being a reliable and hard-working co-author, and for using spaces over tabs. Thanks to Daniel for his chill and easy-going attitude, and bringing the core leftist ideas into the lab. Thanks to Abdullah Talayhan for his special show of rubick cube solving and juggling at the same time, and for keeping the line of Turkish existence in the lab after I leave.

I am also grateful for having two friends, Recep and Fazil, with whom I was able to share the stress of pursuing PhD around the same time. More than that, they have been my fellows for countless crazy hikes that are full of physical challenges, unsafe routing decisions, bad weather and terrain conditions. Without them, hikes with such memorable events would have remained unattempted.

I would like to also thank many students with whom I had the chance to work with as supervisor. Besides the technical-level discussions we had, and all the exchanges on life, Swiss culture, and world politics. Thanks to Dora Neubrandt, Tijana Klimovic, Tiago Kieliger, Sergio Roldán and Miro Haller for their meticulous work.

And thanks to David Balbás, Clement Humbert, Mladen Dimovski, Victor Mollimard, Ciprian Baetu, Julien Corsin, Iraklis Leontiadis. Thanks to Phillip Gajland for being my last office mate, and getting us familiar with the 996 working hour system.

I would like to thank the jury members Prof. David Atienza Alonso, Prof. Lejla Batina and Prof. Thomas Peyrin for spending their valuable time to read my dissertation and attend my oral exam. I would like to also thank Prof. Paolo Ienne for kindly accepting my invitation to act as the jury president.

Lastly, I would like to also thank my family for their support. To my mother Hatice, from whom I borrowed the very few good personal traits. To my elder sister, Rüveyda, for her encouragement to follow my goals and apply for the PhD position. To my younger siblings, Salih and Rümeysa, for bringing further colors into our family.

Thanks to my beloved wife Sakine, for supporting me in the harshest and the most stressful of times. She has always balanced my pessimism with her optimism. I thank her for believing in me, showing the patience during my ups and downs, and being always there to comfort me.

Abstract

Most of the cryptographic protocols that we use frequently on the internet are designed in a fashion that they are not necessarily suitable to run in constrained environments. Applications that run on limited-battery, with low computational power, or area constraints, therefore requires the new designs as well as improved implementations of cryptographic primitives, hence emerges the field *lightweight cryptography*.

In this thesis, we contribute to this effort in few separate directions, in particular regarding block ciphers and block-cipher-based authentication scheme implementations as application-specific integrated circuits (ASIC).

First, we look at optimizations that can be achieved at higher level. In particular, we show that the complete AES family (with varying key sizes 128, 192 and 256) can be realized as combined lightweight circuit, in a manner that shares the storage elements in order to save up silicon area.

Secondly, we contribute in the evaluation of a new design paradigm of fork cipher. We look at how much lightweight efficiency can be gained with this new AEAD design approach, by implementing ForkAES both in round-based and byte-serial implementations. Our comparison with respect to silicon area and energy consumption provides useful insights into AEAD design process.

Lastly, in the large portion of this thesis, we look at the permutation layer of block ciphers from the perspective of serial-circuits. Based on the permutation theory, we establish a method to divide the permutation layers of AES, SKINNY, GIFT and PRESENT into simpler swap operations. Given that these swap operations are cheap in ASIC, we further provide architectural optimization techniques for the implementation of these block ciphers, and we provide the smallest 1-bit implementations of them.

Keywords: hardware implementation, lightweight cryptography, authenticated encryption with associated data, fork cipher, ForkAES, AES, SKINNY, GIFT, PRESENT, NIST LWC

Résumé

Les protocoles que nous utilisons sur Internet sont pour la plupart conçus pour la puissance de calcul élevée qui est disponible dans nos téléphones ou nos ordinateurs. Cependant, nous avons aujourd'hui de plus en plus de contraintes à devoir réduire cette puissance de calcul. En conséquence, le besoin d'une cryptographie alégée apparaît.

Dans cette thèse, nous nous concentrons sur les techniques d'optimisation pour les chiffrements par blocs et les primitives de chiffrement authentifié basées sur le chiffrement par blocs.

Tout d'abord, nous examinons l'implémentation optimisée de AES pour toutes les tailles de clefs en tant que circuit combiné. Nos résultats montrent qu'un tel circuit peut être conçu avec un petit budget en surface.

Ensuite, nous examinons le concept de *fork cipher* récemment proposé. Ici, nous testons si l'intuition des concepteurs se traduit effectivement par des gains en terme de surface en silicone ou de consommation d'énergie.

Enfin, nous fournissons des techniques pour réaliser des implémentations avec peu de silicone pour du chiffrement par blocs tel que AES, SKINNY, GIFT, PRESENT lorsque l'on considère des circuits en série.

Açıklama

İnternette kullandığımız birçok kriptografik protokol, bilgisayar ve telefonlarımızda var olan güçlü hesaplama kapasitesi göz önüne alınarak dizayn edilmiştir. Fakat günümüzde sınırlı hesaplama ve enerji kapasitesine sahip bir çok uygulama görmekteyiz, ve bu tip uygulamaların sayısı hızla artmakta. Bu kısıtlamalar tekrar değerlendirildiğinde, kriptografik katmandaki protokollerin yerine getirilebilmesi için yeni dizayn ve tasarımlara ihtiyaç vardır. İşte bu yüzden, kriptografik araştırma alanında hafif/kolay hesaplanabilir kriptografi alt dalını görmekteyiz.

Bu doktora tezinde, sıklıkla kullanılan blok şifreleme ve doğrulanmış şifreleme elemanlarına, ASIC (application-specific integrated circuit) teknolojisini göz önünde bulundurarak bakmaktayız. Bu protokollerin daha hafif koşullarda çalıştırılabilmesi için tasarladığımız yeni optimizasyon tekniklerini bu çalışmamızda sunmaktayız. Bunun yanı sıra, var olan dizayn fikirlerini, ASIC teknolojisi üzerinde test ederek onların ne kadar tutarlı ve verimli sonuçlar verdiğini test ediyoruz. Çalışmamız günümüzde sıklıkla kullanılan AES, GIFT, SKINNY gibi blok şifreleme mekanizmaları için, kapladığı alan açısından en küçük ve aynı zamanda optimal hıza sahip devrelerle sonlandırıyoruz.

Contents

1 Introduction

Over the last decades, cryptography has furnished a complete set of tools to secure our daily communications on the internet. Over this medium, a communication between a server and a client is usually secured with the use of TLS, which in return, employs a large number of high-level cryptographic protocols such as key exchange mechanisms, signature schemes, as well as fundamental primitives such as block ciphers and message authenticated codes.

At the time of the writing of this thesis, my browser and the server hosting EPFL's web page[1] agree on the use of AES-GCM to fulfill the three conventional goals of symmetric cryptography, which are *confidentiality*, *authenticity* and *integrity*. In other words, as these blocks are transmitted from my computer to their destination through a number of untrusted intermediate internet service providers (or vice versa), not only the content of the blocks are protected against eavesdropping, but any tampering or injection of bits into the original message are also prevented.

From a more technical point of view, AES-GCM is a construction example of what is known as authenticated encryption with associated data (AEAD), i.e. the primitive that fulfills the three goals mentioned above all at once. As cryptographers tend to follow a modular approach during the designing phase, most AEAD primitives first divide arbitrary-size associated data and message into fixed-size blocks, and process them sequentially (or in parallel depending on the mode). The very algorithm that lives at the core of AEAD whose task is to process blocks and the surrounding mode of operation are the two most crucial determining factors of how lightweight or heavyweight the overall scheme is.

Let us take a look at AES-GCM to observe this approach in practice, which is in fact a hybrid of the two components: the counter mode of operation employing the block cipher AES-128 and an accumulator from the Galois field $\mathsf{GF}(2^{128})$. This simply means that all the exchanged content is first properly chopped into 128-bit blocks, and then XORed

[1] https://www.epfl.ch

with the output blocks from AES-128 to obtain the ciphertext blocks. In parallel, the accumulator value is updated through a multiplication in $\mathsf{GF}(2^{128})$ for each processed 128-bit block. Therefore, one block cipher call and one multiplication in the Galois field is performed per 128-bit. In short, a rather heavyweight operation is used as the provider of confidentiality, authenticity and integrity, which fortunately, is an easy chore for our mobile devices and personal computers. Alas, we cannot expect all applications to have such abundance of computational power.

On the other side of the spectrum lie the applications which lack the computational means to perform such heavyweight encryption operation. Depending on the particular application, the reason behind scarcity of computational power can be any of the following: limits on energy (or power) consumption, expected low latency, tight budget on silicon area etc. We can name a number of examples to these applications: sensor network devices that run on batteries, RFID devices with tight silicon area budget, low-power wearables that track the owner's activity, bio-implants, low-power wide-area network (LoRa/LPWAN) devices etc. This list will grow even larger if we consider the number of emerging IoT applications.

All these applications partly rely on the future promise of the sub-field *lightweight cryptography*, which pledge both the *design* of new cryptographic primitives and *optimization techniques* to realize them efficiently. Therefore, this research field relies not only on the established cryptographic discipline, but also on the good understanding of how cryptography is implemented in practice (be it as ASIC, through FPGA, or on microprocessors). Without cryptographic discipline, the effort is tainted with the risk of producing insecure primitives that omit the decades of accumulated experience on the design of symmetric schemes. And without engineering-focused thinking, the design of cryptographic primitives rely only on high-level intuitions whose outcome is likely to be less-than-optimal in lightweight metrics, which may not create sufficient incentive to move away from heavier AES-GCM.

One should note that the design and optimization efforts are usually intertwined. On one hand, the design of new primitives frequently rely on the intuition gained during optimization attempts. On the other, new findings on the optimization techniques rely on the designers' insight. Advances and new results in one soon sparks results in the other.

The contributions in this thesis falls under the category of finding and improving the optimization techniques on block ciphers and block-cipher-based authenticated encryption schemes. The lightweight metrics are defined with respect to application-specific integrated circuits (ASIC), as most decade-old cryptographic schemes eventually find themselves directly implemented in silicon.

Lightweight cryptography through new designs. One particular and well established collaborative effort in this field is to design a lightweight authenticated encryption

primitive that is efficient with respect to the metrics of power, energy, area and throughput. Among these metrics, optimization of area is particularly well studied, as it aligns well with how simpler the operations are, and also comes with the benefit of being much easily quantifiable.

The block cipher family KATAN (whose precursor was the stream cipher Trivium) and then later Simon were in some sense aimed to achieve a lower limit of lightweight encryption in terms of area occupied in silicon [CDK09, CP08, BSS$^+$]. Both these ciphers have update functions based on shift registers, which is efficient to implement in ASIC. Similarly, PRESENT, RECTANGLE and later GIFT were designed with the disposal of the matrix-based diffusion layer altogether to achieve further reduction in area [BKL$^+$07, ZBL$^+$15, BPP$^+$17]. In order to compensate for the lack of MixColumns-like layer, these three block ciphers rely heavily on the permutation layer that operates on 1-bit level. Another such example is SKINNY family of block ciphers and their low-latency variant MANTIS, which follow the same sequence of layers from AES, but instead they employ a lighter S-box and MixColumns matrix [BJK$^+$16].

Yet another example is Midori which was designed to optimize energy consumption [BBI$^+$15]. With a separate goal, the block cipher Prince was designed for low-latency applications like memory encryption [BCG$^+$12].

These efficient-by-design approach naturally extends all the way into authenticated encryption primitives such as ForkAE, GIFT-COFB, HYENA, LOTUS-AEAD, Romulus, SKINNY-AEAD, SUNDAE-GIFT; which not only pursue this goal by employing more lightweight block ciphers, but also simplify processing of each block of message (and associated data) in the surrounding mode of operation [ALP$^+$19, BCI$^+$19, CDJN19, CDJ$^+$19b, IKMP19, BJK$^+$19, BBP$^+$19].

Lightweight cryptography through circuit optimization techniques. In the second major line of research lies the work dedicated towards, not design, but improvement and optimization of the existing schemes. Before diving into the details of the optimization process, a reliable method for quantifying lightweight metrics is established. Given a particular cryptographic scheme $\mathbb{S}$, how we perform this metric quantification has close relationship with the type of platform $\mathbb{S}$ is implemented on. For instance, with field-programmable gate arrays (FPGA), the number of slices, which inherently consists of look-up tables (LUT) and flip-flops, is interpreted as the area metric for the implementation of $\mathbb{S}$. However, expressing $\mathbb{S}$ in terms of LUTs, as in FPGA, typically does not give the best results for other metrics such as energy and power consumption, hence the method for obtaining these metrics are not reliable in the strictest sense. This is because the power and energy consumption by the FPGA itself is typically large, hence its effects on the implementation of $\mathbb{S}$ stays diminished. On the contrary, ASIC is a more reliable platform for evaluating the efficiency of $\mathbb{S}$ according to multiple metrics; such as power, energy, area or throughput; as $\mathbb{S}$ is implemented at gate-level, with much more

freedom, and the measurements can be extracted directly from the final circuit. As a third option, one can look at the optimization of $\mathbb{S}$ when implemented as assembly and run on particular microprocessor. This thesis concerns only ASIC, yet one should be aware that the type of platform on which $\mathbb{S}$ is eventually implemented acts as the key determiner of how lightweight $\mathbb{S}$ becomes in practice.

For instance, a line of work has already been devoted to the effort of minimizing the area of AES S-box. Satoh was the first to propose a 32-bit serial architecture for AES-128 [SMTM01]. In his attempt to reduce the overall size of the circuit, he also came up with a small AES S-box implementation, by utilizing a more efficient way to compute the inverse in $\mathsf{GF}(2^8)$. Later, a series of papers are devoted to the methods of reducing the area of AES S-box [Can05, BP12, RTA18, ME19]. Among these results, the one from Canright is particularly significant as it held the record of the smallest combined AES S-box circuit record for almost a decade, and therefore has been used widely as the go-to circuit in full-fledged combined AES implementations [Can05]. More recently, the new record on the smallest stand-alone and combined AES S-box results belong to Maximov [ME19]. One could observe that the inherent complexity of mathematical structure within the AES S-box encouraged designers to seek new S-boxes which have simpler circuit representations. The clear examples are 4-bit S-boxes in PRESENT, GIFT and the 8-bit S-box of SKINNY that can be constructed with small number of logic gates [BKL$^+$07, BPP$^+$17, BJK$^+$16].

In the same fashion, AES MixColumns matrix as a stand-alone circuit also received sufficient attention from the research community [JMPS17, KLSW17, EJMY18, BFI19]. This effort also contributes to the search of MDS matrix with efficient circuit characteristics [SS16, CLM16, LW17, LSL$^+$19, DL18].

On the architectural level, a series of papers were dedicated towards minimizing AES-128 circuit, each improving the area metric gradually [FWR05, MPL$^+$11, BBR16a, BBR16b, JMPS17]. In that direction, minimizing the data path width typically yielded further area reduction results, but at the expense of latency. Particularly, the results from Jean et al. is significant as far as this thesis is concerned, as it was the first work to realize AES, SKINNY and PRESENT as 1-bit serialized circuit [JMPS17]. Besides these results in ASIC world, Wegener et al. also looked at minimization of S-box in FPGA, that uses minimum number of slices [WMM20]. Their work reports the smallest implementation of AES-128 with Xilinx Spartan-6 FPGA family. Adomnicai and Peyrin also report the implementation of AES-128 that complete with fewest number of clock cycles (i.e. instruction) for both ARM and RISC-V instruction set architectures [AP21].

On the energy-efficiency direction, Kerckhof et al. first provided initial results on the effects of voltage scaling and round unrolling of six block ciphers including AES, PRESENT and KATAN [KDH$^+$12]. Batina et al. gave a comprehensive comparison between lightweight block ciphers (including AES) and drew attention to the trade-off between area and energy consumption, and the fact that the two metrics barely correlate at all [BDE$^+$13]. The

idea of optimal unrolling is later studied by Banik et al. in a work which tries to find the optimal unrolling degree r which gives the best energy-efficient lightweight block cipher [BBR15]. Later, Caforio et al. extended this work into AEAD schemes and gave comparative analysis of each block-cipher-based candidate from NIST LWC [CBB20a].

NIST Lightweight Cryptography Standardization. NIST describes this effort, in its own words, as "the process to solicit, evaluate, and standardize lightweight cryptographic algorithms that are suitable for use in constrained environments where the performance of current NIST cryptographic standards is not acceptable" [NISa]. The main body of this thesis (in particular the published work [BB19a, BBRV20, BCB21]) was produced while this standardization process was at the second round. Namely, there are 32 authenticated encryption candidates in the second round. The status report published by NIST classifies these candidates into few categories (see Table 2, [NIS19]):

1. 16 of the remaining candidates are designed in permutation-based fashion. Simply put, they employ a sponge (or follow the footsteps of sponge-like construction) at their core, and typically use absorb-then-squeeze paradigm. In most cases, a fixed permutation is used as the update function, therefore becomes the key component from which efficiency is reached. For instance, the primary member of SpoC is based on 192-bit state, with a lightweight update function (i.e. permutation) sLiSCP-light [AGH+19, ARH+18].

2. 9 of the candidates are based on block ciphers, meaning that a block cipher is at the core of the design and determines the overall efficiency of the candidate. For instance, candidates COMET and HYENA respectively employs block ciphers AES and GIFT at their core [GJN19, CDJN19].

3. 6 of the candidates are based on tweaked (or modified) variants of block ciphers. In the mode of operation, the value of the tweak is typically determined by the nonce, the internal block counter (i.e. how many blocks are processed since the start of AEAD operation) and the domain separator. For instance, the primary member of Romulus employs SKINNY-128-384 tweakable block cipher, whose 384-bit tweakey space is sufficiently large to accommodate the key, the nonce, the counter and the domain separator [IKMP19].

4. Not fitting into the previous categories, Grain128-AEAD is the only (remaining) candidate that follows a stream-cipher-based approach [HJM+19]. The design uses a slightly modified version of the stream cipher Grain-128a [ÅHJM11].

Even though there is a great amount of variety among the candidates on fine-grained details, a common similar structure on the high-level view emerges. Namely, it is common to have an internal cipher state, which is updated with each coming block of associated data and message, and eventually this state is used to produce the tag. Ciphertext

blocks also depend on the intermediate value of this internal cipher state. In that sense, most candidates can be interpreted as modes of operation that employ either a keyless permutation, a block cipher or a tweakable block cipher at their core. Therefore, the great deal of effort goes into minimizing the core component, as well as the surrounding auxiliary operations.

The choice of block ciphers studied in this thesis, i.e. AES, SKINNY, GIFT, is determined by their popularity in NIST LWC (as of 2nd round). These three block ciphers split a great share of use among block-cipher-based constructions (for further details, we refer to Table 6.1 in Chapter 6). Therefore, our hope is that the improvements provided in this work can be a part of more efficient implementations of AE schemes, and eventually find themselves in real-world applications. Exception to this popularity is PRESENT, which is also covered in this thesis. Even though no AE candidate was designed with PRESENT in NIST LWC, we include it in this work given that it is predecessor to GIFT.

More recently, NIST has announced the ten finalists in the lightweight cryptography standardization. Among them, Romulus and GIFT-COFB are the ones that are still relevant from the perspective of the results produced in this thesis, as they employ SKINNY and GIFT respectively.

Side-channel attacks. Side-channels constitute a major line of attacks from which the security of embedded devices can suffer from, if the device is physically accessible to the adversary. Nowadays it is quite cheap for a dedicated attacker to obtain a toolchain and perform power side-channel measurements on any given circuit. This means that power side-channel attacks become more and more serious threat, especially for those applications where the adversary has physical access to the victim device. Protecting encryption, realized in the form of ASIC, from leakages remain one of the great challenges.

There are numerous protection mechanisms suggested in the literature against side-channels. In this context, the idea of representing a value x with two shares $x_1 \oplus x_2 = x$ is one of the most commonly used path to achieve this goal, as proposed by Chari et al. and Goubin et al. [CJRR99, GP99]. This particular 2-share masking scheme is secure under the assumption that the adversary is able to mount only single probe of measurement, i.e. first-order security. A security model as well as a scheme that is provably secure against adversary up to t probes is subsequently put forward by Ishai [ISW03]. Ishai's scheme uses $2t + 1$ shares to represent a value, where the designer/implementor can choose t as she wills, at the expense of larger cost in hardware. This probing model is later extended as to include the information leakage caused by glitches by Nikova et al. [NRR06].

These proposed side-channel resistant schemes are typically inspired by, and intended for, hardware circuits. To that effect, a function f is first represented in terms of its circuit realization $\mathcal{C}_f$ at gate-level. Then, a generic masking scheme, or a *transformer* in Ishai's terms [ISW03], replaces each gate with a *gadget*, wherein each wire of the

original gate is represented with k different shares/wires in the gadget. Henceforth, a transformer converts an unmasked circuit $\mathcal{C}_f$ to k-share circuit C_f^k. Then, the $(t+1)$-th order probing model allows the adversary $\mathcal{A}$ to observe up to t different wires of $\mathcal{C}_f^k$ ($k \geq t+1$). Depending on whether the model at hand considers the transient nature of signals on the observed wires, the information that is obtained by the adversary can be analog and transient, i.e. glitch-resistant security models [NRR06, FGP$^+$18], or binary and static during a given time slice [ISW03].

In this thesis, we did not consider producing side-channel resistant implementations as one of our priority goals. We must note that although side-channel attacks can be devastating for some applications, there are still many others where the adversary does not have the physical access or means to perform side-channel attacks. For these applications, one could skip the aformentioned techniques for the sake of efficiency and readily use the implementations provided in this thesis. With that said, it is possible to upgrade most of our implementations following the generic idea of masking or threshold implementation techniques [ISW03, NRR06]. This research venue is further discussed in Chapter 7.

Contributions. The contribution of this thesis goes into few directions under the optimization techniques part, where the implementations are realized as ASIC. We actually focus on block ciphers and block-cipher-based authenticated encryption primitives.

In the litetarure, the previous work had focused on the small-area implementations of AES-128 [JMPS17, Max19, ME19, BBR16a, BBR16b], however there was no reported implementations for AES-192 and AES-256. We filled this gap by our work *Six Shades of AES* [BB19b], presented at Africacrypt 2019, and implemented the complete AES family in the form of a small combined circuit.

With *Exploring Lightweight Efficiency of ForkAES* [BB19a], we turn our attention to the recently proposed forking cipher paradigm. In this work, we have investigated whether the high-level trade-off lies beneath the idea of forking a cipher actually translates into lightweight metrics in hardware. Namely, comparing AES with ForkAES, what is the additional cost incurred by the forking. Such implementation-based evaluations are useful in the sense that it allows us to determine whether a design intuition actually works in practice, even before a great amount of cryptanalysis effort is devoted to the proposed design. Our results show that for area-metric, ForkAES indeed matches the designer's intuition, but the same is not true for the energy consumption. The disruption in the energy-efficient AES circuit leads to significant loss in power consumption, meaning that it is not suitable for applications with tight energy and power constraints. Our work was presented at Indocrypt 2019.

With *Swap and Rotate: Lightweight Linear Layers for SPN-based Blockciphers* [BBRV20], we revisit 1-bit serial implementations of block ciphers, whose permutation layer also operates at 1-bit level, such as GIFT and PRESENT. In a pursuit to achieve the smallest

area implementations in hardware, we observe that the permutation layers, by their 1-bit level nature, are the hardest part of the circuit, which received almost no attention from the community. Building on top of permutation theory, we devise a technique to execute the permutation layers of these block ciphers efficiently, with as few additional gates as possible. As showcase, we show that smaller members of their own family, we give the smallest implementations of GIFT-64 and PRESENT-80. Our work was presented at FSE 2020.

In the last part, with *The Area-Latency Symbiosis: Towards Improved Serial Encryption Circuits* [BCB21], we show that the techniques devised in our previous work [BBRV20] can be applied to wide range of block ciphers, including AES, SKINNY, GIFT and GIFT*[2].These applications lead not only the smallest implementations, but also achieve the theoretical minimum latency a 1-bit serial implementation of a block cipher can have. With these implementations at their core, we give the smallest AEAD implementations from NIST LWC. For instance, with our technique, SUNDAE-GIFT can be realized as cheaply as 1200 GE (w.r.t. STM 90 nm). Our work will be presented at TCHES 2021.

Personal Bibliography. The list of peer-reviewed publications I have co-authored during my PhD studies are given below in chronological order. The content of this thesis is compiled from the articles marked with bold font. Our work *Energy Analysis of Lightweight AEAD Circuits* presented at CANS 2020 received the best paper award.

1. Fatih Balli, F. Betül Durak, and Serge Vaudenay. BioID: A Privacy-Friendly Identity Document. In Sjouke Mauw and Mauro Conti, editors, *Security and Trust Management - 15th International Workshop, STM 2019, Luxembourg City, Luxembourg, September 26-27, 2019, Proceedings*, volume 11738 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2019 [BDV19]

2. **Fatih Balli and Subhadeep Banik. Six Shades of AES. In *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, volume 11627 of *Lecture Notes in Computer Science*, pages 311–329. Springer, 2019 [BB19b]**

3. **Fatih Balli and Subhadeep Banik. Exploring Lightweight Efficiency of ForkAES. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*, volume 11898 of *Lecture Notes in Computer Science*, pages 514–534. Springer, 2019 [BB19a]**

[2]We use GIFT* to denoted the modified version of GIFT that assumes different ordering of input and output bits (see Appendix A, [BPP+17]).

4. **Subhadeep Banik, Fatih Balli, Francesco Regazzoni, and Serge Vaudenay. Swap and Rotate: Lightweight Linear Layers for SPN-based Blockciphers. *IACR Transactions on Symmetric Cryptology*, 2020(1):185–232, 2020 [BBRV20]**

5. **Fatih Balli, Andrea Caforio, and Subhadeep Banik. The Area-Latency Symbiosis: Towards Improved Serial Encryption Circuits. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):239–278, 2021 [BCB21]**

6. Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the Core Primitive for Optimally Secure Ratcheting. In *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 621–650. Springer, 2020 [BRV20]

7. Andrea Caforio, Fatih Balli, and Subhadeep Banik. Energy Analysis of Lightweight AEAD Circuits. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings*, volume 12579 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2020 [CBB20a]

8. Andrea Caforio, Fatih Balli, and Subhadeep Banik. Melting SNOW-V: Improved Lightweight Architectures. *Journal of Cryptographic Engineering*, 2020 [CBB20b]

9. Roldán S. Lombardía, Fatih Balli, and Subhadeep Banik. Six Shades Lighter: a Bit-serial Implementation of the AES Family. *Journal of Cryptographic Engineering*, 2021 [LBB21]

Acknowledgement. The work presented in this thesis is produced during the time the author was affiliated with the *Energy-efficient Cryptography* project, funded by the Swiss National Science Foundation (SNSF) through the Ambizione Grant PZ00P2_179921.

2 Preliminaries

This section first establishes a common mathematical notation that is used throughout the thesis. Then, reminders on the details of block ciphers are given. Lastly, a brief coverage on application-specific integrated circuits (ASIC) is presented.

2.1 Mathematical Notation

We denote the set of integers $\{a, a+1, \ldots, b\}$ with $[a, b]$. Furthermore, we define $(a, b) = [a, b] \setminus \{a, b\}$, $(a, b] = [a, b] \setminus \{a\}$ and $[a, b) = [a, b] \setminus \{b\}$. $[n]$ is shorthand for $[0, n]$. We extend this interval notation to sequences. Namely, let $x_0, x_1, \ldots, x_{\ell-1}$ be a sequence of ℓ elements. Then, $x_{a:b}$ refers to the sub-sequence $x_a, x_{a+1}, \ldots, x_b$ for some $0 \leq a < b \leq \ell - 1$.

The bit string concatenation is denoted with $||$. Given a bit string x, $x \ggg y$ denotes the right rotation of x by y bits. Similarly, $\lll$ denotes the left rotation.

We use the symbol $\mathcal{S}_n$ to denote the permutation group on the set $[n-1]$. Naturally, we have $|\mathcal{S}_n| = n!$ and $\mathcal{S}_n$ is a non-commutative group. For an integer $1 \leq k \leq n$, a k-cycle $\pi \in S_n$ is generally expressed as the k-tuple $(i_1, i_2, \ldots, i_k)$ which implies

1. $\forall j \in [k-1], \quad \pi(i_j) = i_{(j+1) \bmod k}$,

2. $\forall i \in [n-1] \setminus \{i_1, \ldots, i_k\}, \quad \pi(i) = i$.

Moreover, we use the term *swap* to refer to 2-cycles.

Denote by $\mathbb{A}_\pi$ the *activity set* of the permutation, i.e. $\mathbb{A}_\pi = \{i : \pi(i) \neq i\}$. The cycles π_1 and π_2 are called disjoint if they have no active elements in common, i.e. $\mathbb{A}_{\pi_1} \cap \mathbb{A}_{\pi_2} = \emptyset$.

2.2 Hardware-oriented Summary of Primitives

2.2.1 AES

AES is a family of block ciphers based on substitution permutation network (SPN). Let r denote the number of rounds in the encryption algorithm, n denote the number of key expansion rounds, ℓ denote the byte size of the key for a given AES member. Note that, the tuple (r, n, ℓ) receives the values $(10, 10, 16)$, $(12, 8, 24)$ or $(14, 7, 32)$ for AES-128, AES-192, and AES-256 respectively.

In AES, for some integer j, a byte string $B_0||B_1||\dots||B_{4j-1}$ is mapped into $(4 \times j)$-byte matrix in the following fashion:

$$\begin{bmatrix} B_0 & B_4 & \dots & B_{4j-4} \\ B_1 & B_5 & \dots & B_{4j-3} \\ B_2 & B_6 & \dots & B_{4j-2} \\ B_3 & B_7 & \dots & B_{4j-1} \end{bmatrix}.$$

This mapping is used to construct the initial 4×4 cipher state matrix from given plaintext, or construct the $4 \times (\ell/4)$ key state matrix from given ℓ-byte key. At the end of the encryption, the same mapping is used to construct the ciphertext from the final value of the cipher state.

State operations. Each round consists of sequential calls of AddRoundKey, SubBytes, ShiftRows and MixColumns. Let $B = B_0||\dots||B_{15}$ denote the cipher state, and K denote the current round key.

1. During AddRoundKey, the state matrix is XORed with the round key, which is extracted from the key state. Therefore, $B \leftarrow B \oplus K$.

2. SubBytes passes each byte of the state from 8-bit input, 8-bit output Rijndael S-box. Therefore, $B_i \leftarrow \mathsf{S\text{-}box}(B_i)$ for all $i \in [15]$.

3. During ShiftRows, the 2nd (resp. 3rd and 4th) row is rotated left by 1 (resp. 2 and 3) positions. Namely, for each $j \in \{1, 2, 3\}$,

$$B_j||B_{j+4}||B_{j+8}||B_{j+12} \leftarrow (B_j||B_{j+4}||B_{j+8}||B_{j+12}) \lll 8j$$

4. During MixColumns, each column $(B_{4j}, B_{4j+1}, B_{4j+2}, B_{4j+3})$ for $j \in [3]$ is multiplied

by the following matrix M in $\mathsf{GF}(2^8)$:

$$\begin{bmatrix} B_{4j} \\ B_{4j+1} \\ B_{4j+2} \\ B_{4j+3} \end{bmatrix} \leftarrow \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} B_{4j} \\ B_{4j+1} \\ B_{4j+2} \\ B_{4j+3} \end{bmatrix}$$

An important property of M is $M^4 = I$, where I denotes the identity matrix. We skip further details of these four layers, and refer the reader to the AES standard [NIS01]. For multiplications in $\mathsf{GF}(2^8)$, the irreducible polynomial $x^8+x^4+x^3+x+1$ is used. Therefore, given a byte values $b = b_0||\ldots||b_7$, $2 \times b$ can be computed as follows:

$$2 \times b = b_1||b_2||b_3||(b_4 \oplus b_0)||(b_5 \oplus b_0)||b_6||(b_7 \oplus b_0)||b_0$$

and $3 \times b$ can be computed with $(2 \times b) \oplus b$.

In the last round, MixColumns is skipped and a final whitening key is XORed into the cipher state to obtain the ciphertext.

Key operations. In order to obtain sufficiently long fresh key material for multiple calls of AddRoundKey operation, AES derives 128 bits of round key for each round by expanding the original encryption/decryption key. This expansion algorithm is denoted with KeyExpand. We recall and emphasize that for AES-192 and AES-256, the encryption/decryption keys are actually larger than 128 bits, hence each invocation of the key expansion algorithm produces 192, 256 bits of round keys respectively. This means that for AES-192, 2 key expansion calls are made for every 3 rounds of execution in the cipher state; and for AES-256, 1 key expansion call is made for every 2 rounds of execution in the cipher state.

We abuse the key notation, and let $K_0, \ldots, K_\ell$ denote the sequence of round key bytes derived by scheduling an encryption key $K_0, \ldots, K_{r-1}$ for a particular choice of AES-128, AES-192 or AES-256 (where the first r bytes conveniently overlap with the key itself). Therefore, $r \in \{176, 208, 240\}$ in given order. Below, we briefly remind the key scheduling algorithm.

The key expansion call is made for 10, 8, 7 times for each version of AES-128, AES-192, AES-256 respectively. These number of calls generate sufficient number of bytes as each state update consumes exactly 16 bytes of round key, regardless of the key length of the AES member.

In particular, suppose that $K_{0:15}$ denotes the encryption key for AES-128, and the full sequence of key bytes $K_{0:175}$ is defined through the iteration of the key expansion algorithm. For each subsequent value of index $i \in \{16, 32, \ldots, 160\}$, we will repeat the following key expansion algorithm to compute the next round key $K_{i:i+15}$ from the previous round key

$K_{i-16:i-1}$. For each i, the sub-sequences $K_{i:i+3}$ are computed by:

$$\begin{bmatrix} K_i \\ K_{i+1} \\ K_{i+2} \\ K_{i+3} \end{bmatrix} \leftarrow \begin{bmatrix} K_{i-16} \\ K_{i-15} \\ K_{i-14} \\ K_{i-13} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S\text{-}box}(K_{i-3}) \oplus \mathsf{RC}_{i/16} \\ \mathsf{S\text{-}box}(K_{i-2}) \\ \mathsf{S\text{-}box}(K_{i-1}) \\ \mathsf{S\text{-}box}(K_{i-4}) \end{bmatrix}$$

and for the remaining sub-sequences $K_{i+4:i+15}$, the formula is simply $K_j \leftarrow K_{j-16} \oplus K_{j-4}$ for each $j \in \{i+4, \ldots, i+15\}$.

In the case of AES-192, the sequence $K_{0:207}$ is derived in a similar fashion from the encryption key $K_{0:23}$. Again, we let i visit the elements of the set $\{24, 48, \ldots, 192\}$ in given order. For the sub-sequences $K_{i:i+3}$, the formula is

$$\begin{bmatrix} K_i \\ K_{i+1} \\ K_{i+2} \\ K_{i+3} \end{bmatrix} \leftarrow \begin{bmatrix} K_{i-24} \\ K_{i-23} \\ K_{i-22} \\ K_{i-21} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S\text{-}box}(K_{i-3}) \oplus \mathsf{RC}_{i/24} \\ \mathsf{S\text{-}box}(K_{i-2}) \\ \mathsf{S\text{-}box}(K_{i-1}) \\ \mathsf{S\text{-}box}(K_{i-4}) \end{bmatrix}$$

and for $K_{i+4:i+23}$, the formula is $K_j \leftarrow K_{j-24} \oplus K_{j-4}$ for $j \in \{i+4, \ldots, i+23\}$.

Finally, in the case of AES-256, the key sequence is $K_{0:239}$ and the encryption key is $K_{0:31}$. Let $i \in \{32, 64, \ldots, 224\}$. The sub-sequences $K_{i:i+3}$ and $K_{i+16:i+19}$ are derived with:

$$\begin{bmatrix} K_i \\ K_{i+1} \\ K_{i+2} \\ K_{i+3} \end{bmatrix} \leftarrow \begin{bmatrix} K_{i-32} \\ K_{i-31} \\ K_{i-30} \\ K_{i-29} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S\text{-}box}(K_{i-3}) \oplus \mathsf{RC}_{i/32} \\ \mathsf{S\text{-}box}(K_{i-2}) \\ \mathsf{S\text{-}box}(K_{i-1}) \\ \mathsf{S\text{-}box}(K_{i-4}) \end{bmatrix}$$

$$\begin{bmatrix} K_{i+16} \\ K_{i+17} \\ K_{i+18} \\ K_{i+19} \end{bmatrix} \leftarrow \begin{bmatrix} K_{i-16} \\ K_{i-15} \\ K_{i-14} \\ K_{i-13} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S\text{-}box}(K_{i+12}) \\ \mathsf{S\text{-}box}(K_{i+13}) \\ \mathsf{S\text{-}box}(K_{i+14}) \\ \mathsf{S\text{-}box}(K_{i+15}) \end{bmatrix}$$

and for the remaining $K_{i+4:i+15}$ and $K_{i+20:i+31}$, the formula is $K_j \leftarrow K_{j-32} \oplus K_{j-4}$ for $j \in \{i+4, \ldots, i+15\} \cup \{i+20, \ldots, i+31\}$.

From a serial circuit perspective, these operations align well with the way we construct pipelines, which will be introduced in Section 2.3. One can simply express key expansion algorithm in terms of four basic operations:

- ke0 (key expand 0) is the operation that computes the first byte of the next round key. Particularly in AES-128 (resp. AES-192, AES-256), this corresponds to computation

of K_{i+16} (resp. K_{i+24}, K_{i+32}). For example in AES-128, this can be done by feeding K_{13} from the last column into the S-box, and XORing three terms as follows: $K_{16} \leftarrow K_0 \oplus \mathsf{S\text{-}box}(13) \oplus \mathsf{RC}_1$.

- ke1 (key expand 1) is the operation that computes the bytes K_{i+17}, K_{i+18} and K_{i+19}, in a similar fashion to ke0. The difference is that we skip the addition of the round constant. For example, in AES-128, this operation would compute $K_{17} \leftarrow K_1 \oplus \mathsf{S\text{-}box}(K_{14})$.
- ke2 (key expand 2) is an operation that is used only in AES-256. It is used to compute the four bytes K_{i+16}, K_{i+17}, K_{i+18}, K_{i+19}. In particular, one such computation is $K_{48} \leftarrow K_{16} \oplus \mathsf{S\text{-}box}(K_{44})$.
- kxor (key XOR) is the operation that computes all remaining bytes of the next round key. It XORs the current key byte with the $(\ell - 4)$-th previous key byte, e.g. $K_{20} \leftarrow K_4 \oplus K_{16}$ in AES-128 or $K_{36} \leftarrow K_4 \oplus K_{32}$ in AES-256. Each key expansion round contains 12, 20, 24 kxor operations in AES-128, AES-192, AES-256 respectively.

In summary, one can express key expansion algorithm with a sequence of these operations, where $\mathsf{ke1}^i$ denotes the execution of ke1 i times:

- In AES-128, the sequence is $\mathsf{ke0}, \mathsf{ke1}^3, \mathsf{kxor}^{12}$.
- In AES-192, the sequence is $\mathsf{ke0}, \mathsf{ke1}^3, \mathsf{kxor}^{20}$.
- In AES-256, the sequence is $\mathsf{ke0}, \mathsf{ke1}^3, \mathsf{kxor}^{12}, \mathsf{ke2}^4, \mathsf{kxor}^{12}$.

And finally, RC_i denotes the sequence of round constant bytes, generated with $\mathtt{0x02}^{i-1}$ in $\mathsf{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.

2.2.2 SKINNY

SKINNY tweakable family of block ciphers was introduced by Beierle et al. [BJK+16]. In this thesis, we are interested only in the three variants that utilize 128-bit blocks. Among these three, SKINNY-128-128 (resp. SKINNY-128-256, SKINNY-128-384) admits 128 (resp. 256, 384) bits of tweakey, and runs for 40 (resp. 48, 56) rounds.

Let $\mathcal{M} = M_0||M_1||\dots||M_{15}$ denote the plaintext. The state matrix is initialized as a 4×4 byte matrix according to the following placement of bytes:

$$\mathsf{St} = \begin{bmatrix} M_0 & M_1 & M_2 & M_3 \\ M_4 & M_5 & M_6 & M_7 \\ M_8 & M_9 & M_{10} & M_{11} \\ M_{12} & M_{13} & M_{14} & M_{15} \end{bmatrix}$$

For $128z$ bits of tweakey, z blocks are constructed, for $z \in \{1, 2, 3\}$. For example, with SKINNY-128-128, $\mathcal{TK}1$ is mapped to a matrix as below:

$$\begin{bmatrix} \mathsf{TK1}_0 & \mathsf{TK1}_1 & \mathsf{TK1}_2 & \mathsf{TK1}_3 \\ \mathsf{TK1}_4 & \mathsf{TK1}_5 & \mathsf{TK1}_6 & \mathsf{TK1}_7 \\ \mathsf{TK1}_8 & \mathsf{TK1}_9 & \mathsf{TK1}_{10} & \mathsf{TK1}_{11} \\ \mathsf{TK1}_{12} & \mathsf{TK1}_{13} & \mathsf{TK1}_{14} & \mathsf{TK1}_{15} \end{bmatrix}$$

Similarly, SKINNY-128-256 contains two tweakey blocks $\mathcal{TK}1$, $\mathcal{TK}2$, and SKINNY-128-384 contains $\mathcal{TK}1$, $\mathcal{TK}2$, $\mathcal{TK}3$.

State operations. SKINNY-128-128 consists of 40 (resp. 48, 56) rounds. Each round is a sequence of SubCells, AddConstants, ShiftRows, AddRoundTweakey and MixColumns.

1. With SubCells, each byte is passed through 8-bit input, 8-bit output SKINNY S-box. SKINNY uses its own dedicated S-box, which follows similar simplistic design approach after PICCOLO [SIH+11a]. The details of S-box is given in Appendix A.1.3.

2. The AddConstants operation simply XORs a round constant matrix (given below) onto the state matrix. First, a 6-bit LFSR is used to compute the round constant for each round. We denote the state of the LFSR with the bit string $\mathsf{rc}_0||\ldots||\mathsf{rc}_5$ with the following update function f:

   $$f(\mathsf{rc}_{0:5}) = \mathsf{rc}_{1:5}||(\mathsf{rc}_0 \oplus \mathsf{rc}_1 \oplus 1)$$

   These six round constant bits are initialized to zero, and updated before being used in the round function. Letting $c_0 = 0^4||\mathsf{rc}_{2:5}$, $c_1 = 0^6||\mathsf{rc}_{0:1}$, $c_2 = 0^6||1||0$, the mapping of them into 4×4 state array is as follows:

   $$\begin{bmatrix} c_0 & 0 & 0 & 0 \\ c_1 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3. During AddRoundTweakey, only the first and the second rows of the tweakey matrices are extracted and XORed onto the state matrix. For example, because SKINNY-128-256 owns two blocks of tweakey $\mathcal{TK}1$ and $\mathcal{TK}2$, the added round key becomes:

   $$\begin{bmatrix} \mathsf{TK1}_0 \oplus \mathsf{TK2}_0 & \mathsf{TK1}_1 \oplus \mathsf{TK2}_1 & \mathsf{TK1}_2 \oplus \mathsf{TK2}_2 & \mathsf{TK1}_3 \oplus \mathsf{TK2}_3 \\ \mathsf{TK1}_4 \oplus \mathsf{TK2}_4 & \mathsf{TK1}_5 \oplus \mathsf{TK2}_5 & \mathsf{TK1}_6 \oplus \mathsf{TK2}_6 & \mathsf{TK1}_7 \oplus \mathsf{TK2}_7 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Table 2.1 – The tweakey permutation P from the key scheduling algorithm of SKINNY.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	8	9	10	11	12	13	14	15	2	0	4	7	6	3	5	1

4. During ShiftRows, the i-th row is shifted in the rightward direction by $i-1$, for $i \in \{2,3,4\}$. This is similar to AES in that each row is shifted, but this time in the rightward direction.

5. And finally, during MixColumns, the state is multiplied by the following matrix in $\mathsf{GF}(2^8)$.

$$M = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Key operations. For each tweakey block, the key scheduling algorithm differs slightly. The two operations are done over each of the tweakey blocks:

1. Each byte of a tweakey block is relocated to another position, through a permutation P given in Table 5.5. The permutation P is defined in a fashion that the byte at position i is moved to the position $P(i)$ after execution of this layer. For all tweakey blocks $\mathcal{TK}1$, $\mathcal{TK}2$ and $\mathcal{TK}3$, the same permutation P is applied.

2. For $\mathcal{TK}2$ (resp. $\mathcal{TK}3$), every byte in the first and second columns are updated with LFSR functions L_2 and L_3, where L_2 and L_3 are defined as below.

$$L_2(x_{0:7}) = x_{1:7}||(x_0 \oplus x_2)$$
$$L_3(x_{0:7}) = (x_1 \oplus x_7)||x_{0:6}$$

Finally, the ciphertext is obtained from converting the state matrix back into a sequence of bytes, following the same ordering among bytes as before.

2.2.3 PRESENT

PRESENT is a small family block cipher based on substitution permutation network. Both members, PRESENT-80 and PRESENT-128, admit 64-bit plaintext block and 80/128-bit key respectively. In this thesis, we are only interested in the 80-bit key variant, therefore we use PRESENT to specifically refer to PRESENT-80.

Table 2.2 – The permutation function P of the linear layer of PRESENT. As our notation of a bit string is inverted (leftmost bit is indexed by 0) in contrast to the original submission [BKL+07], the table is updated accordingly.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

State operations. Let $\mathcal{M} = m_0||m_1||\dots||m_{63}$ be the plaintext. PRESENT encryption contains 31 rounds, where each round consists of, in given order, AddRoundKey, SubBytes and PermLayer.

1. During AddRoundKey, 64-bit is extracted as a round key and XORed with the cipher state.
2. During SubBytes, the state is divided into nibbles, $b_0||\dots||b_{63} = s_0||\dots||s_{15}$, and each nibble s_i is updated through 4-bit input, 4-bit output S-box. Therefore, $s_i \leftarrow \mathsf{S\text{-}box}(s_i)$ for all $i \in [15]$.
3. During PermLayer, each bit at position i is moved to $P(i)$, according to the bit-level permutation given in Table 2.2. Therefore, the updated state b' is computed with $b'_{P(i)} \leftarrow b_i$.

At the end of the round 31, a final AddRoundKey is executed and the ciphertext becomes available.

Key operations. Let $k_0||\dots||k_{79}$ denote the state of the key pipeline. Then the leftmost 64 bits $k_{0:63}$ are extracted as the round key. Afterwards, in order to produce updated key state for the next round, the key sequence is updated as follow:

1. $k_0||k_1||\dots||k_{79} \leftarrow k_{61:79}||k_{0:60}$,
2. $k_{0:3} \leftarrow \mathsf{S\text{-}box}(k_{0:3})$,
3. $k_{60:64} \leftarrow k_{60:64} \oplus \mathsf{rc}_{0:4}$.

Here, the 5-bit round constant value actually denotes the current round of encryption such that $\sum_{j=0}^{4} 2^{4-j}\mathsf{rc}_j$ is the current round value. The counting of rounds starts from 1 in the initial round, and obtains the value 31 in the last round, therefore each non-zero value of $\mathsf{rc}_{0:4}$ is used only once.

2.2.4 GIFT

GIFT family of block ciphers were inspired by PRESENT, and includes two members GIFT-64 and GIFT-128 [BPP+17].

GIFT-64

GIFT-64 admits 128-bit key, 64-bit blocks, and consists of 28 rounds. We describe the block cipher below.

State operations. Let $\mathcal{M} = m_0||m_1||\dots||m_{63}$ be the plaintext. A round consists of, in given order, SubBytes, PermLayer, and AddRoundKey:

1. During SubBytes, the state is divided into nibbles, and each nibble is updated through 4-bit input, 4-bit output S-box. Namely, let $b_0||\dots||b_{63} = s_0||\dots||s_{15}$, then for each $i \in [15]$, $s_i \leftarrow \mathsf{S\text{-}box}(s_i)$ is computed. This S-box is described in Appendix A.1.2.

2. During PermLayer, each bit at position i is moved to $G_{64}(i)$, according to the permutation given in Table 2.3. Therefore, the updated state $b'_0||\dots||b'_{63}$ is computed from the current state $b_{0:63}$ such that $b'_{G_{64}(i)} \leftarrow b_i$.

3. During AddRoundKey, two 16-bit words U and V are extracted from the key state. Let these two words be decoded as $U||V = u_0||\dots||u_{15}||v_0||\dots||v_{15}$. Then, the half of the state bits are XORed with the key bits such that

$$b_{4i+2} \leftarrow b_{4i+2} \oplus u_i, \quad b_{4i+3} \leftarrow b_{4i+3} \oplus v_i \quad \forall i \in [15]$$

   which is immediately followed by the addition of the round constant bits:

$$b_0 \leftarrow b_0 \oplus 1, \quad b_{40} \leftarrow b_{40} \oplus c_0, \quad b_{44} \leftarrow b_{44} \oplus c_1$$
$$b_{48} \leftarrow b_{48} \oplus c_2, \quad b_{52} \leftarrow b_{52} \oplus c_3, \quad b_{56} \leftarrow b_{56} \oplus c_4, \quad b_{60} \leftarrow b_{60} \oplus c_5$$

At the end of the round 28, the ciphertext is obtained from this internal cipher state.

Key operations. Let $w_0||\dots||w_7$ denote the key state, where each w_i denotes a 16-bit word. At each round, $(U, V) \leftarrow (w_6, w_7)$ is extracted as the round key from the key state

Table 2.3 – The permutation function G_{64} of the linear layer of GIFT-64. As our notation of a bit string is inverted (leftmost bit is indexed by 0) in contrast to the original submission [BPP+17], the table is updated accordingly.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$G_{64}(i)$	48	1	18	35	32	49	2	19	16	33	50	3	0	17	34	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$G_{64}(i)$	52	5	22	39	36	53	6	23	20	37	54	7	4	21	38	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$G_{64}(i)$	56	9	26	43	40	57	10	27	24	41	58	11	8	25	42	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$G_{64}(i)$	60	13	30	47	44	61	14	31	28	45	62	15	12	29	46	63

before the update. Then the key update is performed as follows, where $\ggg$ denotes the right rotation by given number of steps within the 16-bit word boundaries.

$$w_{0:7} \leftarrow (w_6 \ggg 2)||(w_7 \ggg 12)||w_{0:5}$$

The 6-bit round constant value is updated with an LFSR. The value of the LFSR is reset to 0^6 at the beginning of the encryption, and updated before the first use. The update is performed exactly as in SKINNY:

$$\mathsf{rc}_{0:5} \leftarrow \mathsf{rc}_{1:5}||(\mathsf{rc}_0 \oplus \mathsf{rc}_1 \oplus 1)$$

GIFT-128

GIFT-128 admits 128-bit key, 128-bit plaintext block, and consists of 40 rounds. The state and key operations remain mostly same, with some minor adjustments for larger state size.

State operations. Let $\mathcal{M} = m_0||m_1||\dots||m_{127}$ be the plaintext. A round consists of, in given order, SubBytes, PermLayer, and AddRoundKey:

1. During SubBytes, the state is divided into 32 nibbles, and each nibble is updated through 4-bit input, 4-bit output S-box. Namely, let $b_0||\dots||b_{127} = s_0||\dots||s_{31}$, then for each $i \in [31]$, $s_i \leftarrow \mathsf{S\text{-}box}(s_i)$ is computed. This S-box is the same that is used in the smaller member GIFT-64 (see Appendix A.1.2).

2. During PermLayer, each bit at position i is moved to $G_{128}(i)$, according to the permutation given in Table 2.4. Therefore, the updated state $b'_0||\dots||b'_{127}$ is computed

Table 2.4 – The permutation function G_{128} of the linear layer of GIFT-128. As our notation of a bit string is inverted (leftmost bit is indexed by 0) in contrast to the original submission [BPP+17], the table is updated accordingly.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$G_{128}(i)$	96	1	34	67	64	97	2	35	32	65	98	3	0	33	66	99
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$G_{128}(i)$	100	5	38	71	68	101	6	39	36	69	102	7	4	37	70	103
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$G_{128}(i)$	104	9	42	75	72	105	10	43	40	73	106	11	8	41	74	107
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$G_{128}(i)$	108	13	46	79	76	109	14	47	44	77	110	15	12	45	78	111
i	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
$G_{128}(i)$	112	17	50	83	80	113	18	51	48	81	114	19	16	49	82	115
i	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
$G_{128}(i)$	116	21	54	87	84	117	22	55	52	85	118	23	20	53	86	119
i	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
$G_{128}(i)$	120	25	58	91	88	121	26	59	56	89	122	27	24	57	90	123
i	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
$G_{128}(i)$	124	29	62	95	92	125	30	63	60	93	126	31	28	61	94	127

from the current state $b_{0:127}$ such that $b'_{G_{128}(i)} \leftarrow b_i$.

3. During AddRoundKey, two 32-bit bit strings U and V are extracted from the key state. Let these two words be decoded as $U||V = u_0||\dots||u_{31}||v_0||\dots||v_{31}$. Then, the half of the state bits are XORed with the key bits such that

$$b_{4i+1} \leftarrow b_{4i+1} \oplus u_i, \quad b_{4i+2} \leftarrow b_{4i+2} \oplus v_i \quad \forall i \in [31],$$

which is immediately followed by the addition of the round constant bits:

$$b_0 \leftarrow b_0 \oplus 1, \quad b_{104} \leftarrow b_{104} \oplus c_0, \quad b_{108} \leftarrow b_{108} \oplus c_1$$
$$b_{112} \leftarrow b_{112} \oplus c_2, \quad b_{116} \leftarrow b_{116} \oplus c_3, \quad b_{120} \leftarrow b_{120} \oplus c_4, \quad b_{124} \leftarrow b_{124} \oplus c_5.$$

At the end of the round 40, the ciphertext is obtained from this internal cipher state.

Key operations. Let $w_0||\dots||w_7$ denote the key state, where each w_i denotes a 16-bit word. At each round, $(U, V) \leftarrow (w_3||w_4, w_6||w_7)$ is extracted as the round key from the key state before the update. Then the key update and round constant computations are performed exactly as in GIFT-64.

2.2.5 GIFT*

The sibling variant GIFT* differs from GIFT in the way the bits are arranged into the state matrix, and they are used in the NIST LWC candidates GIFT-COFB and SUNDAE-GIFT [BCI+19, BBP+19]. In particular, we follow the notation used by SUNDAE-GIFT [BBP+19].

We are only interested in the 128-bit member the modified family of GIFT*, hence we skip the details for the smaller member GIFT*-64. Therefore, GIFT* solely refers to the 128-bit key variant. Below, we describe the slight differences in the operations applied on top of the cipher state.

State operations. Let $\mathcal{M} = m_0||m_1||\dots||m_{127}$ be the plaintext. A round consists of, in given order, SubBytes, PermLayer, and AddRoundKey. However, the internal cipher state s is denoted as a matrix of 4×32. Namely, the internal cipher state is initialized as:

$$\begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} s_{0,31} & s_{0,30} & \dots & s_{0,0} \\ s_{1,31} & s_{1,30} & \dots & s_{1,0} \\ s_{2,31} & s_{2,30} & \dots & s_{2,0} \\ s_{3,31} & s_{3,30} & \dots & s_{3,0} \end{bmatrix} \leftarrow \begin{bmatrix} m_3 & m_7 & \dots & m_{127} \\ m_2 & m_6 & \dots & m_{126} \\ m_1 & m_5 & \dots & m_{125} \\ m_0 & m_4 & \dots & m_{124} \end{bmatrix}$$

1. During SubBytes, each vertical column is updated through 4-bit input, 4-bit output S-box (same as in GIFT, Appendix A.1.2). Namely, for each $i \in [31]$:

$$s_{3,i}||s_{2,i}||s_{1,i}||s_{0,i} \leftarrow \mathsf{S\text{-}box}(s_{3,i}||s_{2,i}||s_{1,i}||s_{0,i})$$

2. During PermLayer, each bit $s_{i,j}$ is moved to the new position $(i, G_i(j))$, according to the four permutations G_i given in Table 2.5. In other words, bits are only relocated through row-local permutations. Therefore, the updated state s' is:

$$s'_{i,G_i(j)} \leftarrow s_{i,j}$$

   for each row $i \in [3]$ and column $j \in [31]$.

3. The key scheduling and orientation of the bits remain exactly same as in GIFT. Therefore, during AddRoundKey, two 32-bit bit strings U and V are extracted from the key state. Let these two words be decoded as $U||V = u_0||\dots||u_{31}||v_0||\dots||v_{31}$. Here, U and V are respectively XORed into the second and third columns:

$$s_{2,31-j} \leftarrow s_{2,31-j} \oplus u_j, \quad s_{1,31-j} \leftarrow s_{1,31-j} \oplus v_j$$

   for $j \in [31]$ and a final round constant addition is performed:

$$S_3 \leftarrow S_3 \oplus \texttt{0x800000XY}$$

Table 2.5 – Bit-sliced GIFT* permutation where index 0 is the rightmost bit of a row segment. Following the notation by SUNDAE-GIFT proposal, the bit identified by j moves to the its new position denoted in G_i after application of the permutation [BBP+19].

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
G_0	29	25	21	17	13	9	5	1	30	26	22	18	14	10	6	2	31	27	23	19	15	11	7	3	28	24	20	16	12	8	4	0
G_1	30	26	22	18	14	10	6	2	31	27	23	19	15	11	7	3	28	24	20	16	12	8	4	0	29	25	21	17	13	9	5	1
G_2	31	27	23	19	15	11	7	3	28	24	20	16	12	8	4	0	29	25	21	17	13	9	5	1	30	26	22	18	14	10	6	2
G_3	28	24	20	16	12	8	4	0	29	25	21	17	13	9	5	1	30	26	22	18	14	10	6	2	31	27	23	19	15	11	7	3

where the byte XY corresponds to $00||\mathsf{rc}_{0:5}$.

2.2.6 ForkAES and ForkAE

Let $\mathcal{B}$, $\mathcal{K}$, and $\mathcal{T}$ be non-empty sets, corresponding to plaintext blocks, keys and tweaks. A tweakable forkcipher is a tuple of three deterministic algorithms $(\mathsf{E}, \mathsf{D}, \mathsf{R})$. Each of the three algorithms takes the key K and the tweak T as input, and differ in the following manner:

1. An encryption algorithm $\mathsf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{B} \to \mathcal{B}^2$ produces two blocks of ciphertext (C_1, C_2), given the single-block plaintext P. Notation-wise, we let $\mathsf{E}_K^T(P)[0] = C_0$ and $\mathsf{E}_K^T(P)[1] = C_1$.
2. A decryption algorithm $\mathsf{D} : \mathcal{K} \times \mathcal{T} \times \mathcal{B} \times \{0,1\} \to \mathcal{B}$ admits either of C_0 or C_1 to recover the plaintext P. An additional 1-bit input clarifies whether the input is C_0 or C_1. Then, it holds that $\mathsf{D}_K^{T,b}(\mathsf{E}_K^T(P)[b]) = P$, for all $(K, T, P, b) \in \mathcal{K} \times \mathcal{T} \times \mathcal{B} \times \{0,1\}$.
3. A tag-reconstruction algorithm $\mathsf{R} : \mathcal{K} \times \mathcal{T} \times \mathcal{B} \times \{0,1\} \to \mathcal{B}$ admits C_0 (resp. C_1) and recovers C_1 (resp. C_0). An additional 1-bit input clarifies the direction of the reconstruction, i.e. either $C_0 \mapsto C_1$ or $C_1 \mapsto C_0$. Then, it holds that $\mathsf{R}_K^{T,b}(C_b) = C_{b \oplus 1}$, for all $(K, T, P, b) \in \mathcal{K} \times \mathcal{T} \times \mathcal{B} \times \{0,1\}$ with $(C_0, C_1) = \mathsf{E}_K^T(P)$.

In this thesis, we are especially interested in ForkAES, a forked block cipher based on AES-128. Therefore, $\mathcal{B} = \{0,1\}^{128}$, $\mathcal{K} = \{0,1\}^{128}$ and $\mathcal{T} = \{0,1\}^{64}$. In most of the context we use these algorithms, K and T are fixed, therefore we drop them in notation and simply use $(\mathsf{E}, \mathsf{D}^b, \mathsf{R}^b)$ for this tuple of algorithms.

State operations. The fork cipher ForkAES utilizes the round function of AES. Encryption in total consists of 15 rounds, where C_0 and C_1 are forked from the common internal cipher state after 5 rounds, known as *the forking state*. The forking state is used to compute both C_0 and C_1 with additional 5 rounds, but with different round keys. This

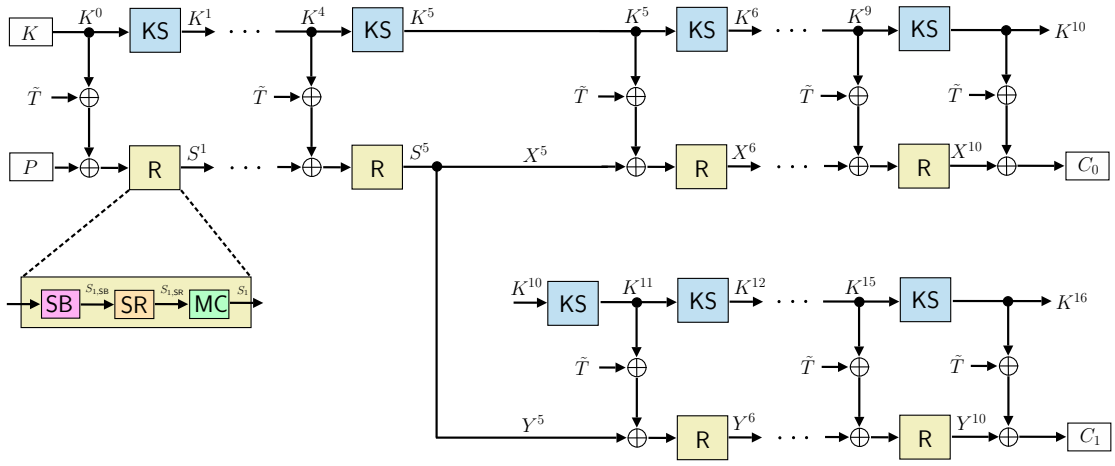

Figure 2.1 – ForkAES Tweakable Block Cipher. SB, SR,MC are SubBytes, ShiftRows and MixColumns operations of AES respectively; KS is a one-round key schedule operation. Formal descriptions of algorithms are given in Figure 2.2.

is visualized in Figure 2.2, where R denotes the combination of SubBytes, ShiftRows and MixColumns.

Key operations. During encryption, AddRoundKey is executed 17 times in total. Therefore, in contrast to AES, the key scheduling algorithm must be invoked 16 times to generate sufficiently many blocks of round keys. For that, ForkAES directly borrows the key scheduling algorithm, denoted by KS, from AES-128.

Besides AddRoundKey, an additional tweakey is also XORed into the state, simultaneously with the round key. However, in order to stretch 64-bit tweak T into 128-bit string, $\tilde{T} = \mathsf{Transpose}(T||0^{64})$ is used, where Tranpose corresponds to transposition in the 4×4 matrix representation.

Exact descriptions of the three algorithms $\mathsf{E}, \mathsf{D}^0, \mathsf{R}^0$ are given in Figure 2.2, and they are used in the construction of two authenticated encryption modes SAEF and PAEF by Andreeva et al. [ARVV18].

2.3 ASIC Details

Understanding the underlying technology on which circuits are implemented is essential in the scope of this thesis, as the very definition of *lightweight* depends immensely on how these circuits are realized. Simply put, an application-specific integrated circuit eventually reduces to complex mesh of N and P-doped layers printed on a silicon wafer.

Encryption $\mathbf{E}_K^T(P)$:
1: $K^0, \ldots, K^{16} \leftarrow \mathsf{KS}^{16}(K)$
2: $\tilde{T} \leftarrow \mathsf{Transpose}(T||0^{64})$
3: $S^0 \leftarrow P$
4: **for** $i = 1$ to 5 **do**
5: $\quad z^i \leftarrow S^{i-1} \oplus K^{i-1} \oplus \tilde{T}$
6: $\quad S^i \leftarrow \mathsf{R}(z^i)$
7: $X^5 \leftarrow S^5$; $Y^5 \leftarrow S^5$
8: **for** $i = 6$ to 10 **do**
9: $\quad u^i \leftarrow X^{i-1} \oplus K^{i-1} \oplus \tilde{T}$
10: $\quad X^i \leftarrow \mathsf{R}(u^i)$
11: $C_0 \leftarrow X^{10} \oplus K^{10} \oplus \tilde{T}$
12: **for** $i = 6$ to 10 **do**
13: $\quad v^i \leftarrow Y^{i-1} \oplus K^{i+5} \oplus \tilde{T}$
14: $\quad Y^i \leftarrow \mathsf{R}(v^i)$
15: $C_1 \leftarrow Y^{10} \oplus K^{16} \oplus \tilde{T}$
16: **return** (C_0, C_1)

Decryption $\mathbf{D}_K^{T,0}(C_0)$:
1: $K^0, \ldots, K^{16} \leftarrow \mathsf{KS}^{16}(K)$
2: $\tilde{T} \leftarrow \mathsf{Transpose}(T||0^{64})$
3: $X^{10} \leftarrow C_0 \oplus K^{10} \oplus \tilde{T}$
4: **for** $i = 10$ to 6 **do**
5: $\quad u^i \leftarrow \mathsf{R}^{-1}(X^i)$
6: $\quad X^{i-1} \leftarrow u^i \oplus K^{i-1} \oplus \tilde{T}$
7: $S^5 \leftarrow X^5$
8: **for** $i = 5$ to 1 **do**
9: $\quad z^i \leftarrow \mathsf{R}^{-1}(S^i)$
10: $\quad S^{i-1} \leftarrow z^i \oplus K^{i-1} \oplus \tilde{T}$
11: $P \leftarrow S_0$
12: **return** P

Reconstruction $\mathbf{R}_K^{T,0}(C_0)$:
1: $K^0, \ldots, K^{16} \leftarrow \mathsf{KS}^{16}(K)$
2: $\tilde{T} \leftarrow \mathsf{Transpose}(T||0^{64})$
3: $X^{10} \leftarrow C_0 \oplus K^{10} \oplus \tilde{T}$
4: **for** $i = 10$ to 6 **do**
5: $\quad u^i \leftarrow \mathsf{R}^{-1}(X^i)$
6: $\quad X^{i-1} \leftarrow u^i \oplus K^{i-1} \oplus \tilde{T}$
7: $S^5 \leftarrow X^5$; $Y^5 \leftarrow S^5$
8: **for** $i = 6$ to 10 **do**
9: $\quad v^i \leftarrow Y^{i-1} \oplus K^{i+5} \oplus \tilde{T}$
10: $\quad Y^i \leftarrow \mathsf{R}(v^i)$
11: $C_1 \leftarrow Y^{10} \oplus K^{16} \oplus \tilde{T}$
12: **return** C_1

Figure 2.2 – The encryption $\mathbf{E}$, decryption $\mathbf{D}^0$, and reconstruction $\mathbf{R}^0$ algorithms of ForkAES.

2.3.1 Technology Libraries and Cells

There is a clear line of abstraction that separates the nitty-gritty details of semiconductor technology and the logical behavior of a given circuit. Manually designing a custom transistor-level semiconductor layout for each unique circuit evidently would not scale very well, hence a *technology library* provides this abstraction layer. Common logical elements, among them are logic gates such as NAND, NOR, XOR as well as sequential elements such as flip-flops, are provided as *cells*. Each cell of the library comes with a functional description (what it does), as well as physical characteristics, such as timing delays, power consumption etc. On the higher level, an engineer can provide a circuit with the help of an hardware description language, which is later realized, thanks to those cells provided by the library.

We use five different technology libraries in the thesis: STM 90 nm, UMC 90 nm, TSMC 90 nm, NanGate 45 nm and NanGate 15nm. The first three among those are commercial, hence they are not publicly available. On the other hand, NanGate libraries are publicly available, but they contain a minimal set of cells, whose power consumption and area metrics are much less optimized than their commercial counterparts. Further details on the libraries are given in Table 2.6.

On the combinatorial part, technology libraries always provide the basic set of logical operations directly as cells, such as XOR, NAND, NOT gates. More often than not, multi-input variants of these gates are also provided, e.g. 3-input XOR/OR/AND. Furthermore, some common simple expressions can also be provided as cells from the library. For

Table 2.6 – The exact variants of technology libraries used in this thesis.

Library	TSMC 90 nm	STM 90 nm	UMC 90 nm	NanGate 45 nm	NanGate 15 nm
Variant	tcbn90lphp low-power	CMOS090LP low-power	fday_sp_rvt_1v0_2011 standard	fast fast	fast fast

instance, the cell `AOI21` from the NanGate 15 nm library implements the function $\neg((A \wedge B) \vee C)$, where A, B and C are input wires to this cell. A 2-input multiplexer (MUX) is also among such common cells with the expression $(A \wedge (\neg S)) \vee (B \wedge S)$, which selects between signals A or B, based on the signal S. NanGate 15 nm provides this as the cell `MUX2`. Further examples for the other libraries can be found in Table 2.7.

On the sequential part, the basic 1-bit storage element is the D flip-flop, whose ports are the input data wire D, the input clock wire C and the output data wire Q. Q propagates the stored value of the flip-flop, and exhibits a very brief glitch while the stored value is being updated. The update always happens at the rising edge of C and the value of D is captured and stored by the flip-flop. Sometimes, a few variants of flip-flops might also be provided by technology libraries:

- Enabled flip-flop takes an additional control wire E, which can enable or disable whether the flip-flop will trigger the update operation on the rising edge of the clock signal. Therefore, if the wire E is low, then the flip-flop is frozen and maintains its stored value by disregarding D. If such functionality is not implemented as cell, it can as well be realized with a simple D flip-flop preceded by a MUX gate.
- A flip-flop with reset signal takes an additional input wire R, which can asynchronously reset the stored value to low (or 0), without having to wait for the next rising edge of the clock signal. Similarly, a flip-flop can have an input set wire, which can asynchronously set the stored content to high (or 1).
- A scan flip-flop takes two data input ports, $D1$ and $D2$, and a selection signal S. On the rising edge of the clock signal, the value to be stored is chosen either from $D1$ or $D2$, based on the value of S. In terms of functionality, this is equivalent to a simple D flip-flop preceded by a MUX gate at its input.

These mentioned functionalities are not necessarily exclusive. For example, UMC 90 nm possess a flip-flop with both reset and set functionalities, i.e. the cell `DFFRSBX1`. On the other hand, NanGate 45 nm possesses a cell with reset and scan functionalities combined, i.e. the cell `SDFFR_X1`.

It is also usual for libraries to have a group of cells that implement the same function but show different delay, area and power characteristics. Names of these cells conveniently differ from their siblings merely by a reserved postfix. If we take NAND gate in Table 2.7

Table 2.7 – The precise name of the cells corresponding to NAND, MUX and D flip-flop from the five technology libraries with their respective area given in μm^2.

Gate	TSMC 90 nm	STM 90 nm	UMC 90 nm	NanGate 45 nm	NanGate 15 nm
NAND	`CKND2D0` 2.822	`ND2HVTX1` 4.390	`ND2CKX1` 3.136	`NAND2_X1` 0.798	`NAND2_X1` 0.197
MUX	`MUX2D0` 6.350	`MUX21HVTX1` 8.781	`MUX2CKX1` 7.056	`MUX2_X1` 1.862	`MUX2_X1` 0.639
D flip-flop	`DFQD1` 12.701	`FD1QSVTX1` 14.269	`QDFFX1` 13.328	`DFF_X1` 4.522	`DFFSNQ_X1` 1.278

as an example, given cell names contain suffixes such as `-X1` or `-D0`, which indicates that these cells are likely the smallest NAND realizations found in those libraries. As a further example, UMC 90 nm library actually contains five different cells for NAND gate with names `ND2CKX1`, `ND2CKX2`, `ND2CKX3`, `ND2CKX4`, and `ND2CKX6`.

2.3.2 Test Bench and Synthesis Options

For all results reported in the thesis, we maintained the following design flow. The design was first implemented at register-transfer level (RTL). A functional verification of the VHDL code was then done using *Mentor Graphics ModelSim*.

Thereafter, *Synopsys Design Compiler* was used to synthesize the RTL design against a technology library. With few exceptions, we used the `compile_ultra` directive of the tool, which instructs the compiler to perform all-in-one optimization with regards to circuit footprint, delay and power consumption. After synthesis, we obtain the netlist, which is the a realization of our RTL-described circuit in terms of the cells provided by the technology library. At this point, owing to the fact that the technology library contains the physical characteristics of each cell, it is possible to compute area and critical path delays of the circuit.

We then use the netlist for two purposes. First, we verify once more the correctness of the circuit with post-synthesis simulation, via *Synopsys VCS MX Compiled Simulator*. Secondly and at the same time, we collect the switching activity of each gate, so that *Synopsys Power Compiler* can use it to derive the average power consumption of the circuit, using the back annotated switching activity.

2.3.3 Higher-level Building Blocks

Pipelines. In a circuit, a bit-serial pipeline is a series of flip-flops that are arranged in a way that allows the stored value to be shifted in a fixed direction, one bit position at each

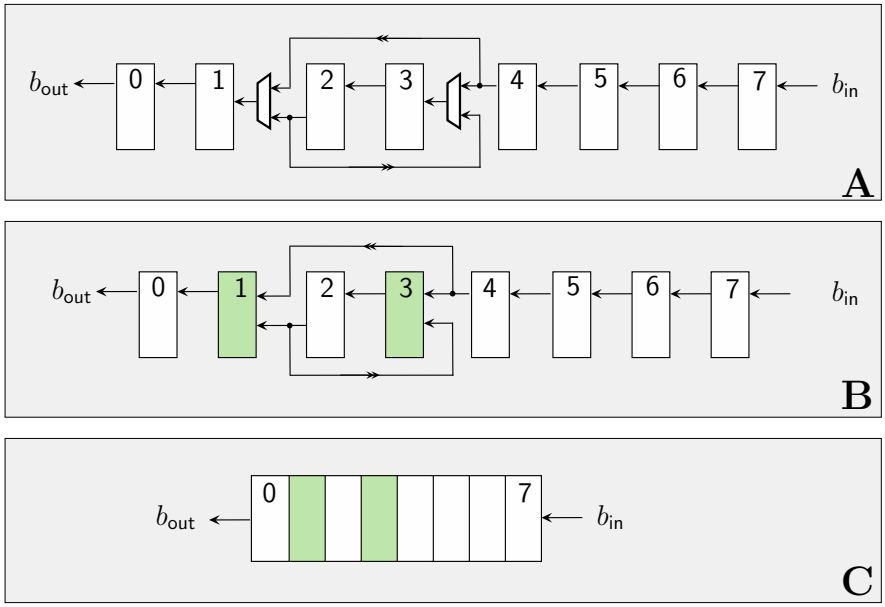

Figure 2.3 – The three equivalent representations of an 8-bit pipeline equipped with the swap operation $(2, 4)$, where each box denotes a flip-flop. **A)** The swap is implemented by adding MUXes at the output of intended swap positions 2 and 4. **B)** If the technology library supports, a scan flip-flop (marked with green) can be used instead, for the flip-flops that immediately come after the intended swap positions. **C)** We can simply color boxes 1 and 3 to denote the swap $(2, 4)$.

clock cycle. In mathematical terms, an n-bit pipeline can be denoted with a series of 1-bit variables $\mathsf{FF}_0, \mathsf{FF}_1, \ldots, \mathsf{FF}_{n-1}$ that supports shifting of the stored bits. The shifting operation then corresponds to

$$b_{\mathsf{out}} \leftarrow \mathsf{FF}_0, \quad \mathsf{FF}_{0:n-1} \leftarrow \mathsf{FF}_{1:n-1} || b_{\mathsf{in}}$$

where b_{in} is the new input value of the pipeline, and b_{out} is the exit value from the pipeline. Typically, an encryption circuit consists of two pipelines, one for the state update operations and the other for the key. Therefore, we use FF^{S} to denote the flip-flops of the state pipeline, and FF^{K} to denote the key pipeline.

On top of the pipeline, we can further introduce additional operations. One particular operation of interest is the *swap* operation (u, v) for some $u, v \in [n-1]$ such that $u \neq v$. The swap operations are denoted with tuples, and (u, v) swaps the contents of FF_u and FF_v without touching the other values. Therefore, it corresponds to:

$$\forall i \in [n-1] \setminus \{u, v\},\ \mathsf{FF}_i \leftarrow \mathsf{FF}_i, \quad \text{and} \quad (\mathsf{FF}_u, \mathsf{FF}_v) \leftarrow (\mathsf{FF}_v, \mathsf{FF}_u).$$

Swap operations are treated specially, because they can be efficiently realized in hardware, by merely introducing two MUXs into the pipeline. This is illustrated in Figure 2.3 with an 8-bit pipeline with the swap $(2, 4)$. In the given circuit, a swap operation (u, v) can be interleaved between two consecutive shift invocation on the pipeline, by controlling the select signals of the MUXes. If the technology library further supports scan flip-flops, this can be implemented even more efficiently with the use of scan flip-flops.

Color Coding. Throughout the thesis, we make extensive use of colors to mark the swap positions in pipelines, as shown in Figure 2.4. This figure should be interpreted as a legend for the rest of the figures. Let us briefly explain the meaning of the color coding with respect to both functionality and cost. Let ℓ_{FF} and ℓ_{MUX} denote the areas of a simple flip-flop (without reset functionality) and a 2-input multiplexer respectively. As shown in Figure 2.4, a white box denotes a simple flip-flop and occupies area equal to ℓ_{FF}. Any box that contains n different colors represents a flip-flop that can accept $n+1$ input bits, one of which is latched on the flip-flop in the next rising clock edge, depending on some select signal. All such instances cost $\ell_{\mathsf{FF}} + n \times \ell_{\mathsf{MUX}}$ units of silicon area (unless the technology library has a dedicated cell implementation for this primitive).

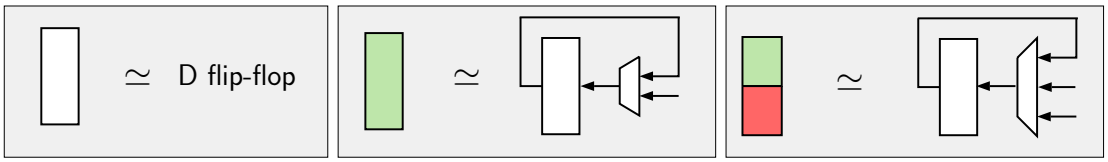

Figure 2.4 – The color legend for interpreting circuit figures. White boxes denote a regular D flip-flops, a single-colored box is used for 2-input scan flip-flop and an n-color flip-flop denotes an $(n + 1)$-input flip-flop. Two boxes that share a color mean that they are interconnected and therefore they can swap their input bits.

3 All-in-one AES Circuit

This chapter focuses on the design of a lightweight AES circuit that can perform both encryption and decryption for each of AES-128, AES-192 and AES-256. Despite large variations in the key scheduling algorithm for each of these six functionalities, we provide a way to express them in terms of common simpler operations, which in return allows us to construct small-size circuit. The results presented in this chapter includes the extension of the work done in collaboration with Subhadeep Banik, which was presented in Africacrypt 2019 [BB19b].

We briefly discuss the few proposed architectures in the literature that also target small-size implementation for AES-128 in Section 3.1, and lay out the details of our contributions in Section 3.3.

In Section 3.4, we give the high-level view of a 8-bit serialized AES circuit, with input and output port details, as well as the arrangement of the bytes for loading to and retrieving data from the circuit. In Section 3.5, we briefly cover the low-level circuit components that are used in the design. Then, in Section 3.6, we give high-level description of the key and state pipelines. In Section 3.7, we give the simpler operations that can be performed by our state and key pipelines, and in Section 3.8 we explain how to express AES encryption and decryption in terms of these operations. In Section 3.9, we explain how to run key scheduling algorithm both in forwards and backwards, for each of AES-128, AES-192 and AES-256.

In Section 3.9, we report the measurements of our implementation and conclude the chapter in Section 3.11.

The VHDL source code of the implementations can be found as a public archive [6AE].

3.1 Related Work

There have been several lightweight implementations of AES suggested in the literature. Satoh et al. [SMTM01] suggested a 32-bit serial architecture. Their main contribution includes minimizing the S-box circuit via tower field implementation, as well as combinatorial optimization of the MixColumns circuitry. The size of this implementation is around 5400 GE. Later, Grain of Sand implementation of Feldhofer et al. was constructed as an 8-bit serialized architecture occupying 3400 GE of area in silicon, yet it comes with a latency over 1000 clock cycles for each of encryption and decryption operations [FWR05]. The implementation by Moradi et al. with size equal to 2400 GE and encryption latency of 226 cycles is one of the smallest known architectures for AES [MPL$^+$11]. Similarly, Mathew et al. reported an 8-bit serial implementation that takes 1947/2090 GE for the encryption/decryption circuits respectively. Their contribution is on a separate direction, as they use intermediate register files for storing the key and the cipher state. These register files can be synthesized in the ASIC flow using memory compilers.

More recently, two further serial architectures have been proposed for AES-128. The first, Atomic AES and its modified version Atomic AES v2.0 by Banik et al. [BBR16a, BBR16b], builds on the ideas of Moradi et al. [MPL$^+$11], with the main contribution being fusion of encryption and decryption into a single pipeline. Namely, they show that the decryption functionality can also be realized along with encryption only with a small area cost. Their combined circuit occupies around 2060 GE. In order to achieve even smaller area results, Jean et al. [JMPS17] takes the design one step further, by proposing the first 1-bit serial architecture of AES, which costs less than 1600 GE. This architecture naturally requires many more clock cycles for encryption and decryption operations, roughly multiplying the latency by 8 (compared to 8-bit serial architectures).

3.2 Motivation

One of the main motivations, besides post quantum trend, to build the smallest all-in-one AES in hardware is that some devices are expected to support large number of standards at the same time. For instance many smart cards are designed to support a large variety of symmetric and asymmetric cryptographic primitives altogether, including all six functionalities of AES[1]. However, the number of protocols that these units can support are limited due to the tight area budget. Our design proposes an alternative combined solution with little extra area requirement, which would allow these cryptographic units to be able to benefit from the use of the full AES with small cost. Besides, a combined implementation provides an upper bound on individual implementations of AES-192 and AES-256, that have not received sufficient attention in the literature.

Another major motivation to develop the combined circuit is the fact that many newer

[1] See Infineon jTOP ID SLJ 52GCA150CL Java Card 3.0.4 150K as an example.

Table 3.1 – Comparison with the state of the art.

block cipher	area	library	impl.	functionality	latency	reference
AES-128	5400 GE	–	32-bit	enc+dec	5	[SMTM01]
AES-128	3400 GE	P 350	8-bit	enc+dec	>100	[FWR05]
AES-128	2400 GE	UMC 180	8-bit	enc	21	[MPL+11]
AES-128	2060 GE	STM 90	8-bit	enc+dec	21	[BBR16b]
AES-128/192/256	3674 GE	STM 90	8-bit	enc+dec	24/32	[BB19b]
AES-128	1785 GE	STM 90	8-bit	enc	16	[BCB21]
AES-128	1596 GE	UMC 90	1-bit	enc+dec	168/248	[JMPS17]
AES-128/192/256	2268 GE	STM 90	1-bit	enc+dec	128	[LBB21]
AES-128	1267 GE	STM 90	1-bit	enc	128	[BCB21]

this work state-of-the-art reference follow-up work

NIST post-quantum designs use AES-256 as a sub-primitive in randomness generation [EBB15, NISb]. Therefore it is necessary to have constrained implementations of AES-256 in hardware without drastically increasing the area budget.

3.3 Contributions

In this chapter, we present an 8-bit serial architecture that performs all encryption and decryption operations of three instances AES-128, AES-192 and AES-256 in a combined circuit. In other words, the circuit supports six functionalities of AES at the same time, with the area-minimization goal in mind. We further eliminate the burden, that the bytes should be ordered according to non-standard row-first fashion, and construct our circuit in a fashion that it admits inputs arranged in the standard column-first fashion. For our novel 6AES circuit, the original work reports around 3674 GE when synthesized with the standard cell library of the STM 90nm CMOS logic process. In this thesis, we extend these results to five libraries mentioned in Section 2.3.1. The comparison with the state of the art is given in Table 3.1.

3.4 Input and Output Formats

Our AES architecture is a sequential one with 8-bit data path. The architecture consists of the following ports:

- 8-bit input KeyIn port, from which the key is loaded in one byte per clock cycle fashion.
- 8-bit input DataIn port, from which the plaintext (resp. ciphertext) during encryption (resp. decryption) is loaded in one byte per clock cycle fashion.

- 3-bit input Ins port, which allows to choose the functionality of the circuit from either one of the members AES-128, AES-192, AES-256 plus encryption/decryption direction.
- An asynchronous active-low signal Rst as input.
- A clock signal Clk as input.
- 8-bit output DataOut port, from which the ciphertext (resp. plaintext) during encryption (resp. decryption) is propagated in one-byte-per-clock-cycle fashion.
- 1-bit Rdy signal as output, which indicates whether the operation is completed.

Loading the input values takes up to 16 (resp. 24, 32) clock cycles for AES-128 (resp. AES-192, AES-256), and receiving the output similarly requires 16 clock cycles. In between, the encryption and decryption operations take few hundreds clock cycles.

We denote the data (i.e. the input plaintext/ciphertext) as a byte sequence $B_0, \ldots, B_{15}$. We denote the original key with $K_0, \ldots, K_{\ell-1}$ where $\ell \in \{16, 24, 32\}$ for AES-128, AES-192 and AES-256 respectively. And lastly, we use K' sequence to denote the last ℓ bytes of the round keys used in AddRoundKey, i.e. the sequence is $K'_0, \ldots, K'_{\ell-1}$. More precisely, this sequence corresponds to $K_{160}, \ldots, K_{175}$ in AES-128; $K_{184}, \ldots, K_{207}$ in AES-192; and $K_{208}, \ldots, K_{239}$ in AES-256.

Loading Cycles. In AES-128, the key and the data has the same size, therefore loading both of them can be synchronized, i.e. K_i (resp. K'_i) and B_i are loaded at the same clock cycle for encryption (resp. decryption). However, in AES-192/256, the key is larger than the data, therefore we should clarify which bytes of the key and the data are loaded at which cycles.

For encryption, the data and the first 16 bytes of key are loaded during the first 16 cycles. If there are remaining bytes of the key, then $K_{16}, \ldots, K_{\ell-1}$ are loaded in the following 8 (resp. 16) cycles in AES-192 (resp. AES-256).

For decryption, the first $\ell - 16$ bytes of the last used round key (i.e. the sequence $K'_0, \ldots, K'_{\ell-17}$) are loaded. In particular, first 8 (resp. 16) cycles are used to load $K'_0, \ldots, K'_7$ (resp. $K'_0, \ldots, K'_{15}$) in AES-192 (resp. AES-256). Then, the following 16 cycles are used to load $K'_{\ell-16}, \ldots K'_{\ell-1}$ and $B_0, \ldots, B_{15}$ simultaneously.

Input Format. For encryption, the key $K_0, \ldots, K_{\ell-1}$ and the data $B_0, \ldots, B_{15}$ are loaded. For decryption, the key byte sequence $K'_0, \ldots, K'_{\ell-1}$ is loaded instead of the original key $K_0, \ldots, K_{\ell-1}$.

Result Cycles. The result data sequence $C_0, \ldots, C_{15}$ (ciphertext for encryption or

plaintext for decryption) is observed at DataOut in given order. The signal Rdy is raised during the 16 cycles during which the result is available.

3.5 Components

On the higher-level, the circuit can be decomposed into following primitives, few of which are already described in Section 2.3.1:

- An enabled byte flip-flop (henceforth referred to as EFF) is a byte storage unit that preserves its output during many cycles when enable signal is unset. When enable signal is set, its value (and thereby output) is updated following the rising edge of the clock signal.
- An enabled byte scan flip-flop (henceforth referred as SEFF) is an EFF combined with a multiplexer (or can be constructed directly with enabled scan flip-flops, if the technology library has them). Two separate bytes are wired as input, and its next value is assigned to either one of them based on an additional selection signal. Its value is updated on the rising edge of the clock signal, if the enable signal is set. If enable signal is unset, its value is preserved. They are used mostly in the state pipeline.
- Control logic, which consists of a finite-state machine activated with the release of the asynchronous reset signal Rst, and a large combinatorial circuit that computes all control signals. It controls all flip-flop enable signals, scan flip-flop selectors, MUX selectors, mask AND selectors, S-box direction signal and Rdy signal.
- The combinatorial MixColumns circuitry that takes 32-bit columns as input and computes the 32-bit by multiplication over $\mathsf{GF}(2^8)$, as described in Section 2.2.1.
- We use the combined S-box architecture by Canright that performs both forward and inverse operations with a low hardware footprint [Can05], i.e. both S-box and its inverse $\mathsf{S\text{-}box}^{-1}$. The direction of the operation is determined with an additional selection signal.
- Round constant lookup table contains ten round constant bytes used in all three instances. An internal 4-bit counter is used to choose the correct entry from the table.

In order to minimize number of gates, we limit our design to a single two-directional S-box (shared between SubBytes and KeyExpand), a single MixColumns circuit that can work in both direction. At the core of the design, we construct two pipelines from a series of EFF and SEFF: one for the state and another for the key. As the key length in AES-256 is 32 bytes, the key pipeline contains 32 byte flip-flops to accommodate it.

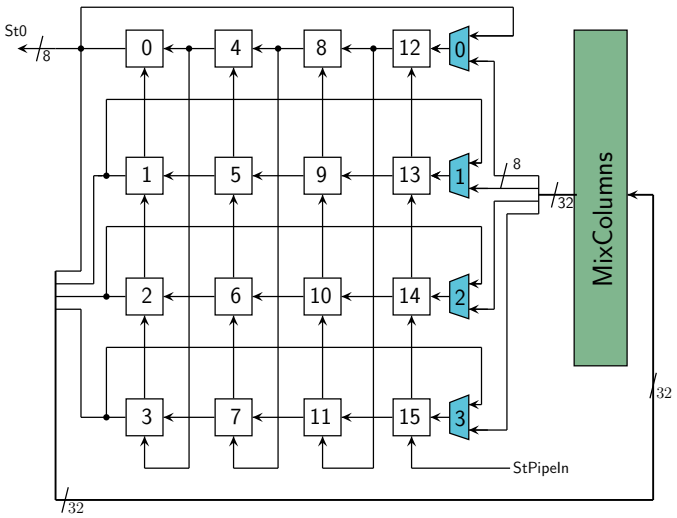

Figure 3.1 – The state pipeline of our combined AES architecture.

3.6 High Level Description of the Design

Our design mainly consists of two parts: the state pipeline and the key pipeline. Below, we describe each of them separately.

State Pipeline. 16 enabled scan byte flip-flops are arranged in an upward-moving serial fashion, where a byte value enters into the pipeline from $\mathsf{FF}^{\mathsf{S}}_{15}$, visits each flip-flops in their descending index order, and eventually exits the pipeline from $\mathsf{FF}^{\mathsf{S}}_{0}$. This is done via the vertical connections among flip-flops which further permit loading the cipher state into the bus with one-byte-per-clock fashion so that AddRoundKey and SubBytes layers can be executed simultaneously. Moreover, alternative lateral connections, e.g. $\mathsf{FF}^{\mathsf{S}}_{12} \rightarrow \mathsf{FF}^{\mathsf{S}}_{8} \rightarrow \mathsf{FF}^{\mathsf{S}}_{4} \rightarrow \mathsf{FF}^{\mathsf{S}}_{0}$, allow each column to be loaded into MixColumns circuit in 4 consecutive cycles. The same lateral connections in the leftward direction also allow us to perform ShiftRows, by carefully enabling and disabling rows in harmony in 3 consecutive clock cycles. With the help of MUXes connected to $\mathsf{FF}^{\mathsf{S}}_{12}$, $\mathsf{FF}^{\mathsf{S}}_{13}$, $\mathsf{FF}^{\mathsf{S}}_{14}$, $\mathsf{FF}^{\mathsf{S}}_{15}$, we can choose between ShiftRows and MixColumns operations. The task of correctly controlling the selection signals is assigned to the control circuit. Therefore, the controller can decide to freeze the whole pipeline (through enable functionality of flip-flops), rotate rows in the leftward direction (through scan functionality of flip-flops), choose to load a column from the output of the MixColumns circuit (through four additional MUXs). The high-level view of the state pipeline circuit is given in Figure 3.1.

Key Pipeline. It consists of 32 enabled flip-flops (without scan functionality) to accommodate the 32 byte key in AES-256. The connections of the pipeline are tweaked through two MUXes (denoted with 5 and 6 in Figure 3.2) in such a way that:

- During AES-128 operations, the flip-flops $\mathsf{FF}^{\mathsf{K}}_{8:23}$ are bypassed and the output of $\mathsf{FF}^{\mathsf{K}}_{24}$ is wired to the input of $\mathsf{FF}^{\mathsf{K}}_{7}$ through MUX 5. Therefore the key pipeline effectively shrinks to 16 byte flip-flops.

- During AES-192 operations, the flip-flops $\mathsf{FF}^{\mathsf{K}}_{16:23}$ are bypassed and the output of $\mathsf{FF}^{\mathsf{K}}_{24}$ is wired to the input of $\mathsf{FF}^{\mathsf{K}}_{15}$ through MUX 6. Furthermore, the output of $\mathsf{FF}^{\mathsf{K}}_{8}$ is wired to the input of $\mathsf{FF}^{\mathsf{K}}_{7}$ through MUX 5. The key pipeline effectively shrinks to 24 flip-flops.

- During AES-256 operations, all flip-flops are active. The output of $\mathsf{FF}^{\mathsf{K}}_{16}$ is wired to the input of $\mathsf{FF}^{\mathsf{K}}_{15}$ through MUX 6. The output of $\mathsf{FF}^{\mathsf{K}}_{8}$ is wired to the input of $\mathsf{FF}^{\mathsf{K}}_{7}$ through MUX 5.

In order to work in harmony with the state pipeline, the task of the key pipeline is to provide the particular byte of key to the bus, so that AddRoundKey can be performed correctly with the byte coming from the state. This key byte from the pipeline can be fetched either from $\mathsf{FF}^{\mathsf{K}}_{0}$, $\mathsf{FF}^{\mathsf{K}}_{20}$ or $\mathsf{FF}^{\mathsf{K}}_{40}$ based on the selection signal of MUX 10. The pipeline supports rotation through connections $\mathsf{FF}^{\mathsf{K}}_{0} \rightarrow \mathsf{FF}^{\mathsf{K}}_{23}$ through MUX 12. As before, enable signals are configured by the control logic and can freeze the pipeline when another operation is stalling the state pipeline.

The most challenging part of our design is by far the computation of proper round key for AddRoundKey operation for 6 different instances on the same circuit. For this reason, a combination of XOR/AND gates is connected to the key pipeline to execute the key expansion algorithm on-the fly (while the pipeline is simultaneously providing key bytes for AddRoundKey). These gates, highlighted with lightgray background in Figure 3.2, are connected to $\mathsf{FF}^{\mathsf{K}}_{0}$ and $\mathsf{FF}^{\mathsf{K}}_{4}$ (positioned above the key pipeline) and they are utilized for key expansion during both encryption and decryption. Moreover, the gates connected to $\mathsf{FF}^{\mathsf{K}}_{7}$, $\mathsf{FF}^{\mathsf{K}}_{11}$, $\mathsf{FF}^{\mathsf{K}}_{15}$, $\mathsf{FF}^{\mathsf{K}}_{27}$, $\mathsf{FF}^{\mathsf{K}}_{31}$ (positioned below the key pipeline) also help key expansion in the reverse direction during decryption.

Main Bus. The main bus is used both by the state pipeline and the key pipeline, and practically allows the sharing of the S-box between both. For the state pipeline, during encryption, the bus can execute AddRoundKey and SubBytes operations simultaneously in given order. During decryption, the bus can execute the inverse of the S-box and then perform inverse key addition, with given reverted order. Therefore, this circuitry is constructed with a single combined S-box that can perform both forward and inverse computation of the S-box, XORs for AddRoundKey, and a couple of MUXs to choose the correct signal accordingly.

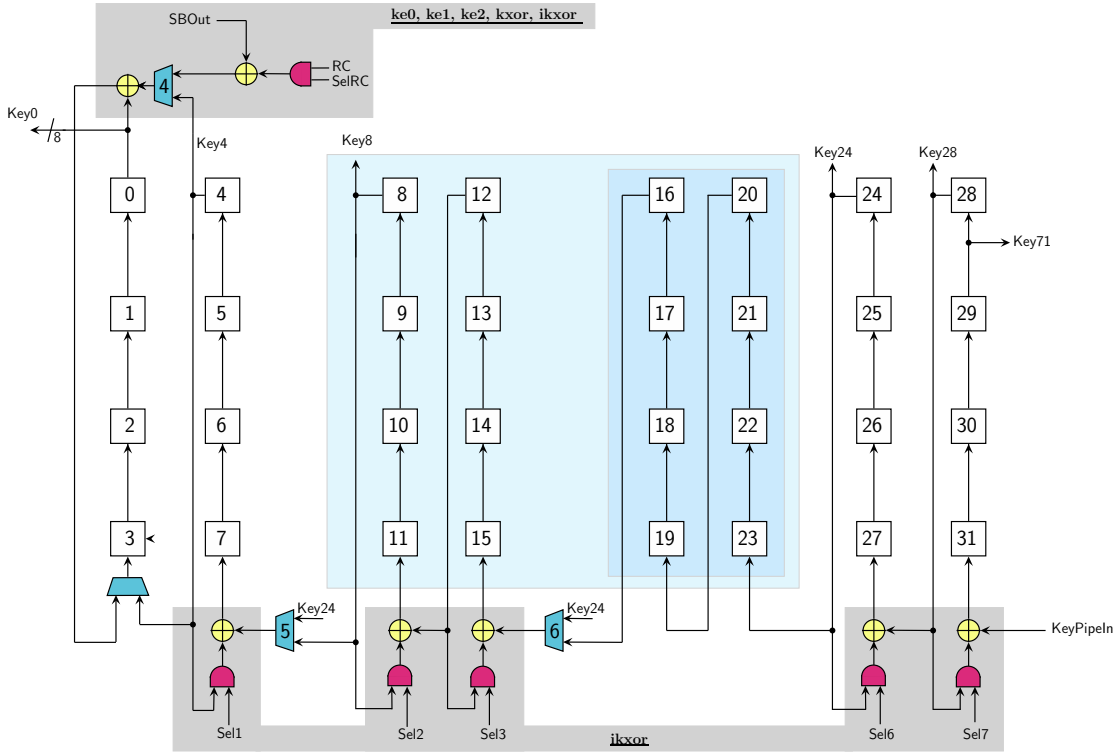

Figure 3.2 – The state pipeline of our combined AES architecture.

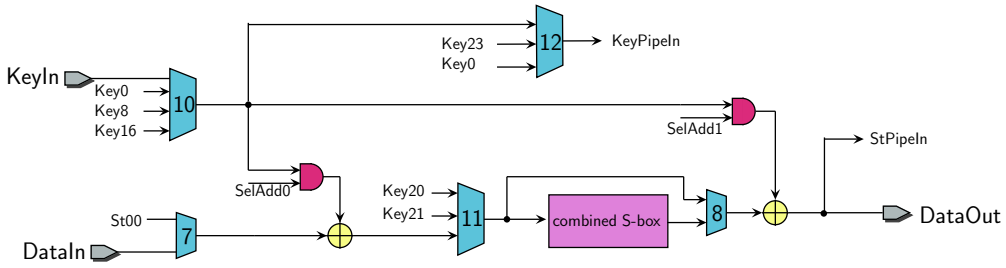

Figure 3.3 – The main bus of our combined AES architecture.

3.7 Elementary Operations of Layers

In order to simplify the explanation of how our circuit operates, we conceptually divide the control of the circuit into various operations. We also explain their connections to four different layers plus KeyExpand in the key scheduling part. Some of the operations described below are computed on completely independent parts of the circuit, hence they can be performed simultaneously by our hardware. The goal is to squeeze them into same cycles to the extent possible. Each of the following instructions sets particular control bits in order to perform its corresponding operation during a clock cycle. If an operation does not explicitly mandate how a certain flip-flop should behave, then we assume that it is disabled (through enable signal). In order to draw a clear distinction, the key pipeline flip-flops are denoted with $\mathsf{FF}_i^{\mathsf{K}}$ and the state pipeline flip-flops are denoted with $\mathsf{FF}_i^{\mathsf{S}}$.

add Both the key and the state pipelines are fully active, and two bytes from each are loaded into the bus. The state byte is fetched from $\mathsf{FF}_0^{\mathsf{S}}$ of the state pipeline. On the other hand, the key byte is fetched from $\mathsf{FF}_0^{\mathsf{K}}$ of the key pipeline (exceptionally in AES-192, $\mathsf{FF}_8^{\mathsf{K}}$ and $\mathsf{FF}_{24}^{\mathsf{K}}$ are also used for fetching the key byte). If the circuit is at initialization phase, then the key and the byte values are actually read from the input ports DataIn and KeyIn of the circuit. If the chosen functionality of the circuit indicated by Ins signal is encryption, the two bytes on the bus are first XORed, and then passed through S-box (therefore AddRoundKey and SubBytes are done concurrently). Otherwise (if Ins indicates decryption), the state byte is passed through inverse S-box, and then the key addition is done (therefore InvSubBytes and AddRoundKey are done concurrently). In either case, the computed byte is stored to $\mathsf{FF}_{15}^{\mathsf{S}}$ of the state pipeline, at the rising edge of the clock cycle. Simultaneously, the byte key is stored back to $\mathsf{FF}_{31}^{\mathsf{K}}$.

sbox MUXes 8, 11 and S-box selection signals are configured accordingly so that S-box can be computed.

isbox MUXes 8, 11 and S-box selection signals are configured accordingly so that inverse S-box can be computed. Both sbox and isbox are performed simultaneously with add during encryption/decryption operations respectively.

srow0 Rotates each of the last three rows of the state pipeline to left by one position. The control logic uses the selection signal of the scan flip-flops to change the direction in the pipeline, and freezes the unused state flip-flops.

srow1 Rotates each of the last two rows of the state pipeline to left by one position.

srow2 Rotates only the last row of the state pipeline to left by one position.

isrow0 Rotates only the second row of the state pipeline to left by one position, i.e. $\mathsf{FF}_1^{\mathsf{S}} \rightarrow \mathsf{FF}_5^{\mathsf{S}} \rightarrow \mathsf{FF}_9^{\mathsf{S}} \rightarrow \mathsf{FF}_{13}^{\mathsf{S}}$.

isrow1 Rotates the second and the third rows of the state pipeline to left by one position.

mixcol MUXes 0, 1, 2, 3 are configured to load the input from MixColumns circuit. Again, the selection signal of all state flip-flops are configured by the control logic so that the pipeline moves in the leftward direction.

ke0 Performs the key expansion operation ke0, as described in Section 2.2.1. In the rising edge of the clock cycle, $\mathsf{FF}^{\mathsf{K}}_3$ is loaded with $\mathsf{FF}^{\mathsf{K}}_0 \oplus \mathsf{S\text{-}box}(\mathsf{FF}^{\mathsf{K}}_{29}) \oplus \mathsf{RC}$. During ke0, all flip-flops of the key pipeline except $\mathsf{FF}^{\mathsf{K}}_{0:3}$ and $\mathsf{FF}^{\mathsf{K}}_{28:31}$ are frozen. Columns $\mathsf{FF}^{\mathsf{K}}_{0:3}$ and $\mathsf{FF}^{\mathsf{K}}_{28:31}$ rotate in the upwards direction. The state pipeline is also frozen.

ke1 The only difference from ke0 is that RC is not XORed into the computation, through SelRC signal. In the rising edge of the clock cycle, $\mathsf{FF}^{\mathsf{K}}_3$ is loaded with $\mathsf{FF}^{\mathsf{K}}_0 \oplus \mathsf{S\text{-}box}(\mathsf{FF}^{\mathsf{K}}_{29})$. Columns $\mathsf{FF}^{\mathsf{K}}_{0:3}$ and $\mathsf{FF}^{\mathsf{K}}_{28:31}$ rotate in the upwards direction. The state pipeline is also frozen.

ke2 Similar to ke1, but the input byte of S-box is not rotated. In the rising edge of the clock cycle, $\mathsf{FF}^{\mathsf{K}}_3$ is loaded with $\mathsf{FF}^{\mathsf{K}}_0 \oplus \mathsf{S\text{-}box}(\mathsf{FF}^{\mathsf{K}}_{28})$. Columns $\mathsf{FF}^{\mathsf{K}}_{0:3}$ and $\mathsf{FF}^{\mathsf{K}}_{28:31}$ rotate in the upwards direction. The state pipeline is also frozen.

kxor For key XOR operation of the key expansion algorithm, the input select bits of $\mathsf{FF}^{\mathsf{K}}_3$ and the MUX 4 are configured to store $\mathsf{FF}^{\mathsf{K}}_4 \oplus \mathsf{FF}^{\mathsf{K}}_0$ in the rising edge of the clock cycle.

ikxor This performs the inverse XOR operation and used during decryption. The corresponding circuitry is shown in a gray background in Figure 3.2. Sel1 Sel2, Sel3, Sel6, Sel7 are the corresponding signals that are configured such that key XOR is done, e.g. $\mathsf{FF}^{\mathsf{K}}_7 \leftarrow \mathsf{FF}^{\mathsf{K}}_4 \oplus \mathsf{FF}^{\mathsf{K}}_8$, only at selected clock cycles during decryption. Similar to kxor, the key pipeline must be fully active, and state pipeline is frozen.

load MUX 10 is configured in such a way that the key is loaded from the input port directly into the pipeline. This is necessary for AES-192, AES-256, for which the key size is larger than block size. The key pipeline is fully active, and the state pipeline is frozen.

rot The key is rotated in the pipeline, where the exit key byte $\mathsf{FF}^{\mathsf{K}}_0$ is fed back into the pipeline through $\mathsf{FF}^{\mathsf{K}}_{31}$. The key pipeline is fully active.

rxor Pseudonym for combination of rot and kxor. Therefore the key is updated on the pipeline with key XOR operation as it rotates.

In the following subsections, we will first look at encryption and decryption round functions performed on the state pipeline, with the assumption that the round key bytes are provided correctly for AddRoundKey. Encryption and decryption round function operations are rather easy to implement with the design given in Figure 3.1 and remains quite similar across six instances. However, the same does not hold for the key expansion, as the different key sizes become a major challenge to deal with.

3.8 Generic Encryption/Decryption Overview

First, for the sake of argument, suppose that the key pipeline always contains the necessary round key K_i at round i, with which AddRoundKey is being computed. Then we can readily convert the encryption algorithm into a sequence of operations. AddRoundKey and SubBytes can be done simultaneously through add and sbox operations in 16 cycles. Then for ShiftRows, it suffices to run srow0, srow1, srow2 subsequently in 3 cycles. Then, in 4 cycles of mixcol, we complete MixColumns. This sequence corresponds to one round of operation in the encryption algorithm, and can be repeated as many times as necessary, as long as the key pipeline handles the key expansion and provides the correctly computed key bytes during AddRoundKey. The same line of reasoning also applies to decryption, where inverse SubBytes and AddRoundKey can be done with isbox and add simultaneously in 16 cycles, inverse ShiftRows can be done with isrow0, isrow1, srow0 in 3 cycles; and inverse MixColumns can be done in 12 cycles of mixcol (as inverse MixColumns is equivalent to 3 repetitions of MixColumns).

Therefore, what remains is to continuously *refresh* the key in the pipeline, by removing *dirty* (used) key bytes, and replacing with *fresh* (unused) key bytes. If the current operation is encryption, then refreshing means computing the next round key. Otherwise (in decryption), we use it to refer to computing the round key in the reverse direction. In the following section, we describe how key bytes are managed in the key pipeline, and how its operations are interleaved with the four layers of encryption and decryption.

3.9 Key Expansion Details

In this section, we describe mainly how the key expansion algorithm is implemented with the key pipeline for six instances of AES, while taking the running mechanism of the state pipeline into account.

AES-128 Encryption. The detailed chronology of operations is given in Figure 3.4. During the first 16 cycles, MUXes 7 and 10 are configured such that the key and the data are loaded to the bus through input ports DataIn, KeyIn (see Figure 3.3). At the same time, AddRoundKey and SubBytes operations are done simultaneously, where the computed state is loaded into the state pipeline, and the key is loaded into $\mathsf{FF}^{\mathsf{K}}_{0:7}$ and $\mathsf{FF}^{\mathsf{K}}_{24:31}$.

A round takes 23 clock cycles to complete. At the beginning of the round, all the keys in the pipeline are dirty (i.e. already used for key addition, hence require key expansion). Therefore, we use the first 4 clock cycles to refresh the first four bytes of the key with executions of ke0, $\mathsf{ke1}^3$. ShiftRows and MixColumns are also performed in the meanwhile. At the end of 7 clock cycles (after ke0, $\mathsf{ke1}^3$ and 3 clock cycles of stalling), the key pipeline still contains 12 dirty key bytes. These bytes are refreshed in the following 12 clock

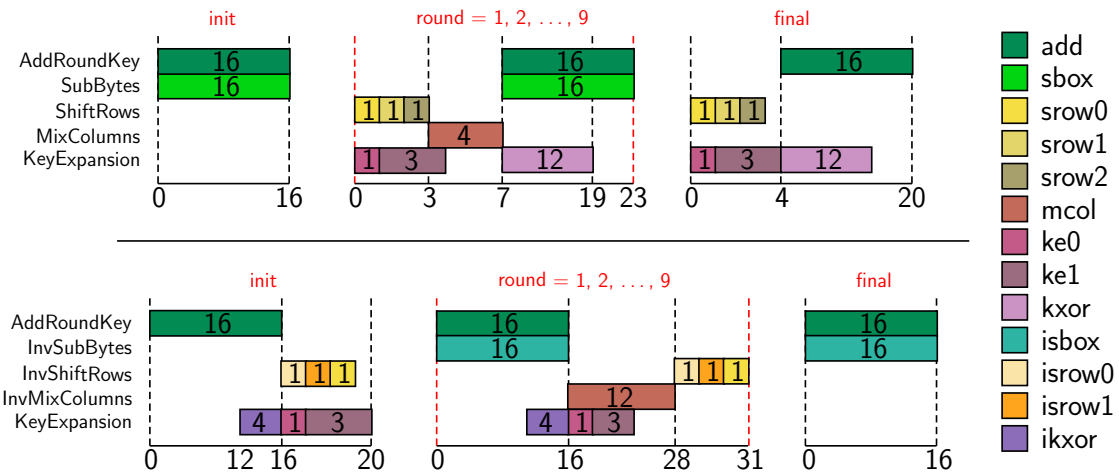

Figure 3.4 – The chronology of operations in AES-128 encryption (on top) and AES-128 decryption (below). The numbers in the boxes indicate the number of cycles over which the operation is executed.

cycles with kxor as the pipeline moves, as they are loaded into $\mathsf{FF}_3^{\mathsf{K}}$ (in Figure 3.2). This computation also overlaps with add and sbox. At the end of a round, all bytes in the key pipeline again become dirty.

In the final round, MixColumns is skipped, and the ciphertext is available during the very last 16 clock cycles.

AES-128 Decryption. We remind that for decryption, KeyIn loads the very last 16 bytes of key used with the last AddRoundKey instead of the original key used for encryption. The rounds can be seen as the symmetrically opposite versions of encryption.

In decryption, a round takes 31 clock cycles to complete, 8 clock cycles more compared to encryption. At the beginning of a round, all bytes in the key pipeline are fresh. At the end of 12 clock cycles, the key pipeline contains only 4 fresh bytes. Then, ikxor is enabled through $\mathsf{FF}_7^{\mathsf{K}}$, $\mathsf{FF}_{27}^{\mathsf{K}}$, $\mathsf{FF}_{31}^{\mathsf{K}}$ (through control signals Sel1, Sel6, Sel7) for 4 clock cycles. Therefore at cycle 16, the key pipeline contains exactly 12 bytes of fresh key, which are stored in $\mathsf{FF}_{4:7}^{\mathsf{K}}$ and $\mathsf{FF}_{24:31}^{\mathsf{K}}$. The remaining dirty key column is refreshed by execution of the sequence ke0, $\mathsf{ke1}^3$ in the following 4 clock cycles. Therefore, at the end of the round, all bytes in the key pipeline become fresh again.

As before, the output of decryption, i.e. the plaintext, becomes available during the last 16 clock cycles.

AES-192 Encryption. The detailed chronology of operations is given in Figure 3.5. Performing the key expansion in AES-192 becomes quite challenging given the fact that each key expansion round generates 24 bytes of new round key, whereas only 16 of them

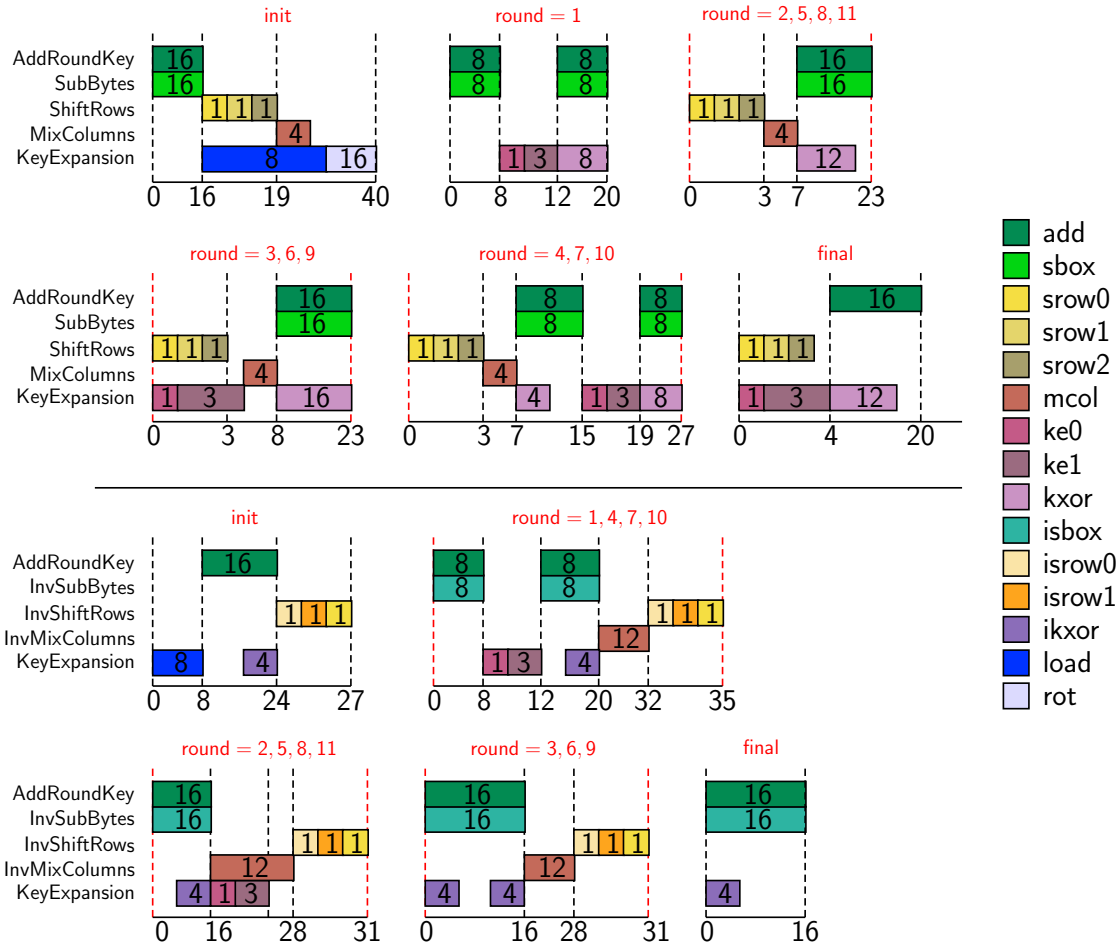

Figure 3.5 – The chronology of operations in AES-192 encryption (on top) and AES-192 decryption (below). The numbers in the boxes indicate the number of cycles over which the operation is executed.

are used for each encryption round. This leads to misalignment and desynchronization between the state pipeline and the key pipeline.

We overcome this by interrupting AddRoundKey and SubBytes operations whenever necessary. Namely, if there is no fresh key byte left on the key pipeline during the execution of these two operations, we stall the state pipeline and run the key expansion algorithm. This leads to three different types of rounds:

1. The first type of round has no fresh key byte in the pipeline at the beginning and has to run a key expansion algorithm before key addition.

2. The second type of round has 4 leftover fresh bytes in $\mathsf{FF}^{\mathsf{K}}_{0:3}$ and 4 dirty bytes in $\mathsf{FF}^{\mathsf{K}}_{4:7}$ that can be refreshed with kxor as the pipeline moves. This means that AddRoundKey and SubBytes can run for 8 clock cycles, but then we stall and

handle the key expansion. Afterwards, the execution of key addition can resume for 8 more clock cycles.

3. The third type of round has 4 fresh bytes in $\mathsf{FF}^{\mathsf{K}}_{0:3}$, and 12 dirty bytes in $\mathsf{FF}^{\mathsf{K}}_{4:15}$ that can be refreshed with kxor as the pipeline moves. Therefore, the key expansion can be run in parallel to key addition, without stalling the state pipeline.

During the first 16 cycles, AddRoundKey and SubBytes are simultaneously performed as before. The next 8 cycles are used to load the rest of the key into the key pipeline. Then, in order to align the key properly, the key pipeline is rotated for 16 cycles with rot. Thereby, at the end, 8 fresh bytes are located at $\mathsf{FF}^{\mathsf{K}}_{0:7}$, and the dirty bytes are at $\mathsf{FF}^{\mathsf{K}}_{8:15}$ and $\mathsf{FF}^{\mathsf{K}}_{24:31}$.

During round 1, we have to interrupt AddRoundKey and SubBytes after 8 cycles, at which point all the bytes in the key pipeline are dirty. The 4 bytes of key that require to be updated by key expand 0 and key expand 1 operations are located at $\mathsf{FF}^{\mathsf{K}}_{0:3}$, therefore we run ke0, $\mathsf{ke1}^3$ in the following four clock cycles. The remaining 20 dirty key bytes are refreshed as they are loaded into $\mathsf{FF}^{\mathsf{K}}_{3}$, by running kxor alongside add and sbox operations, and it overflows into the next round. 8 dirty key bytes are refreshed in the current encryption round and the remaining 12 are deferred to the next round. Note that since 8 AddRoundKey operations are done simultaneously, at the end of this round the number of fresh bytes in the key pipeline is $4 + 8 - 8 = 4$.

At the beginning of rounds 2, 5, 8, 11 (which are type (3) rounds) the pipeline contains only 4 bytes of fresh key, but the following 12 dirty bytes can be refreshed with kxor. Therefore, in order to align correctly, we run kxor during the first 12 cycles of AddRoundKey and SubBytes. At the end of this round, all fresh bytes are therefore used up.

At the beginning of rounds 3, 6, 9 (which are type (1) rounds); the key in the pipeline is completely dirty and the first column requires key expansion 0 and key expansion 1 operations. Therefore the sequence ke0, $\mathsf{ke1}^3$ is run in the first 4 clock cycles. The following 20 bytes of key can be easily refreshed with kxor alongside add and sbox. Among these kxor operations, 16 of them are executed in the current round and the remaining 4 are deferred to the next round.

At the beginning of rounds 4, 7, 10 (which are type (2) rounds) there are 4 bytes of fresh keys followed by 4 bytes of dirty keys that can be refreshed with kxor in the key pipeline. However, the following column of key requires the key expand 0 and key expand 1 operations, so add and sbox is interrupted for key expansion. The remaining 8 bytes of addition continues after 4 cycles of ke0, $\mathsf{ke1}^3$. The ciphertext is available in DataOut during the last 16 cycles.

AES-192 Decryption. Besides the misalignment issues as in encryption, a second obstacle that arises during the decryption is that fresh bytes in the key pipeline do not

necessarily always start from $\mathsf{FF}^{\mathsf{K}}_0$. Recall that for decryption, the last 24 bytes of used round keys are loaded initially, therefore we have to run the key expansion algorithm in the reverse order. Therefore, we have to start refreshing key columns starting with the highest index, i.e. whichever column of key was used last in the encryption should be removed first. At the same time, due to flow direction of the pipeline, the lowest indexed key column occupies $\mathsf{FF}^{\mathsf{K}}_{0:3}$, whereas during various stages of operation, the key columns to be used in key addition are located at $\mathsf{FF}^{\mathsf{K}}_{8:11}$ or $\mathsf{FF}^{\mathsf{K}}_{24:27}$. Our solution is to use MUX 10, so that we can choose the exit byte of the pipeline either from $\mathsf{FF}^{\mathsf{K}}_0$, $\mathsf{FF}^{\mathsf{K}}_8$ or $\mathsf{FF}^{\mathsf{K}}_{24}$. By doing so, we can fetch the correct fresh key byte, even when the key is misaligned in the key pipeline. Meanwhile, we can continue AddRoundKey, InvSubBytes operations without requiring additional clock cycles for rotation. This irregular exit of key bytes from the pipeline is only necessary for AES-192 decryption.

As the last 24 bytes of round key are loaded into the circuit, 8 clock cycles are used for loading the first 8 bytes of this key. Then the following 16 cycles are used for add. During the last four cycles of add, ikxor is also performed through $\mathsf{FF}^{\mathsf{K}}_{15}$, $\mathsf{FF}^{\mathsf{K}}_{27}$, $\mathsf{FF}^{\mathsf{K}}_{31}$ (specifically excluding $\mathsf{FF}^{\mathsf{K}}_3$, $\mathsf{FF}^{\mathsf{K}}_7$, $\mathsf{FF}^{\mathsf{K}}_{11}$). Therefore, at cycle 24, the key pipeline contains 20 fresh bytes (8 unused from the initial load and 12 from ikxor), where the 4 dirty bytes are stored in $\mathsf{FF}^{\mathsf{K}}_{8:11}$ and they can only be refreshed with ke0, $\mathsf{ke1}^3$. Therefore, we will wait until this key moves into $\mathsf{FF}^{\mathsf{K}}_{0:3}$.

At the beginning of rounds 1, 4, 7, 10; the key pipeline contains 20 fresh bytes. However the next 8 fresh bytes to be used for add are located at $\mathsf{FF}^{\mathsf{K}}_{24:31}$, whereas the remaining 8 bytes required for add are located at $\mathsf{FF}^{\mathsf{K}}_{0:7}$. Therefore we fetch the next byte key into the bus from $\mathsf{FF}^{\mathsf{K}}_{24}$, and at the same time rotate the pipeline by connecting $\mathsf{FF}^{\mathsf{K}}_0 \rightarrow \mathsf{FF}^{\mathsf{K}}_{31}$. After 8 cycles, we interrupt isbox and add because the dirty column of key that requires the key expand 0/1 operations to update reaches $\mathsf{FF}^{\mathsf{K}}_{0:3}$, so we can perform ke0, $\mathsf{ke1}^3$. After refreshing this column of keys in 4 cycles, we resume fetching key bytes from $\mathsf{FF}^{\mathsf{K}}_{24}$ for AddRoundKey and InvSubBytes. Concurrently, at the last 4 cycles, we do ikxor with $\mathsf{FF}^{\mathsf{K}}_3$, $\mathsf{FF}^{\mathsf{K}}_7$, $\mathsf{FF}^{\mathsf{K}}_{11}$, $\mathsf{FF}^{\mathsf{K}}_{15}$ to obtain 16 fresh bytes for the next round. We reach to a point where all 24 bytes in the pipeline are fresh.

At the beginning of rounds 2, 5, 8, 11; the key pipeline is completely fresh. However the next 16 bytes of key to be used with add are located at $\mathsf{FF}^{\mathsf{K}}_{8:15}$ and $\mathsf{FF}^{\mathsf{K}}_{24:31}$. Therefore, key bytes are fetched from $\mathsf{FF}^{\mathsf{K}}_8$ into the pipeline, and the pipeline is rotated as before. During the last 4 cycles of add, ikxor is performed over $\mathsf{FF}^{\mathsf{K}}_{11}$. After the 16 add cycles, the bytes of key that require update by key expand 0/1 arrive at $\mathsf{FF}^{\mathsf{K}}_{0:3}$, and therefore ke0, $\mathsf{ke1}^3$ is executed to generate 4 fresh bytes. At the end, the key pipeline contains 16 fresh key bytes in $\mathsf{FF}^{\mathsf{K}}_{0:15}$.

At the beginning of rounds 3, 6, 9; the key pipeline contains 16 fresh bytes starting from $\mathsf{FF}^{\mathsf{K}}_0$, and they are aligned with the state pipeline for add. In order to arrange upcoming key bytes, we still perform ikxor on $\mathsf{FF}^{\mathsf{K}}_{15}$ and $\mathsf{FF}^{\mathsf{K}}_{27}$ for the first 4 clockcycles, and $\mathsf{FF}^{\mathsf{K}}_{15}$

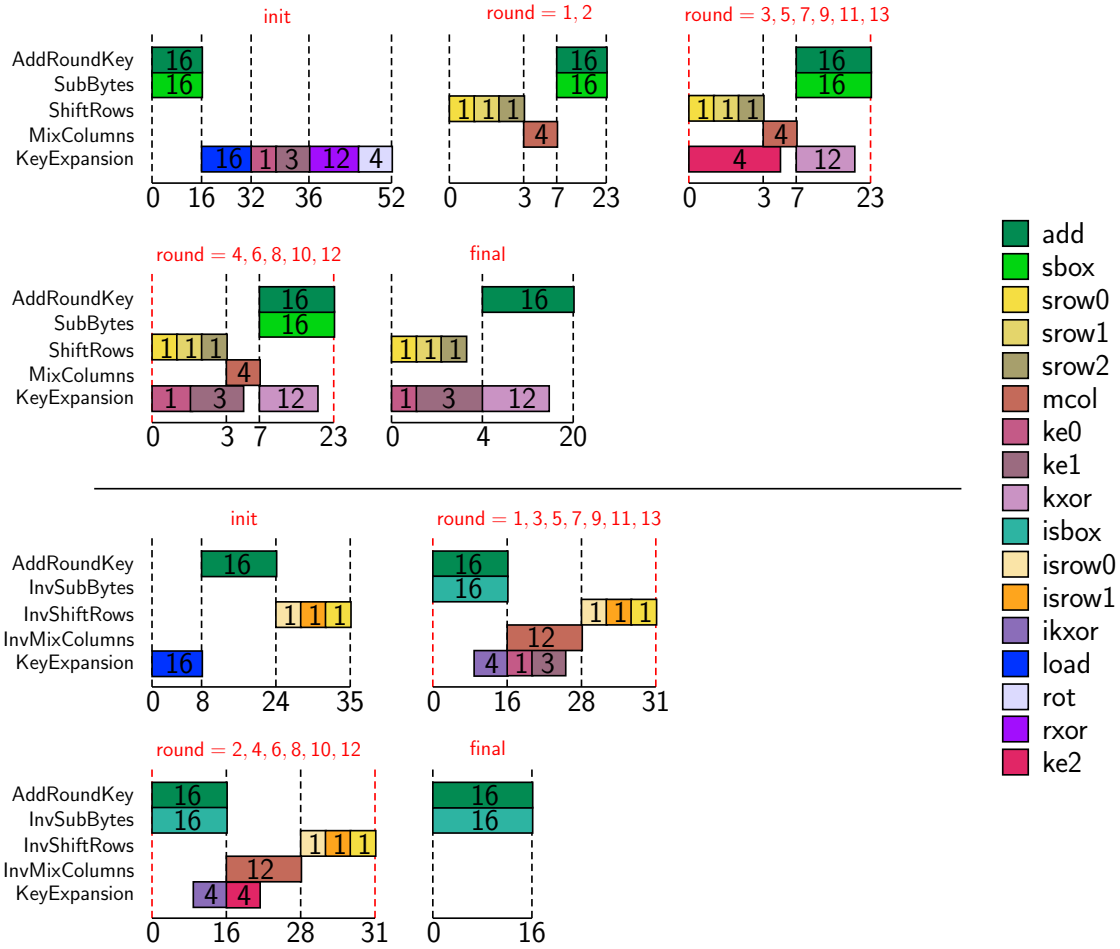

Figure 3.6 – The cycle arrangement of AES-256 encryption (on top) and AES-128 decryption (below). The numbers in the boxes indicate the number of cycles over which the operation is executed.

and $\mathsf{FF}^{\mathsf{K}}_{27}$, $\mathsf{FF}^{\mathsf{K}}_{31}$ for the last 4 clock cycles. At the end, the key pipeline contains 4 dirty bytes located at $\mathsf{FF}^{\mathsf{K}}_{8:11}$.

AES-256 Encryption. The detailed chronology of operations is given in Figure 3.6. AES-256 remains simpler to achieve than AES-192, because each key expansion round produces enough keys for two AddRoundKey operations. During the first 16 cycles, add, sbox is performed. We spend other 16 cycles to load the rest of the key. Then the key expansion is performed with the sequence ke0, $\mathsf{ke1}^3$, and the key is rotated for 16 cycles to move the fresh key bytes to $\mathsf{FF}^{\mathsf{K}}_{0:15}$. During the first 12 cycles of this period, we also enable kxor (named rxor for convenience) so that dirty keys are refreshed as they rotate through the pipeline.

At the beginning of round 1, the key pipeline is completely fresh, therefore there are sufficient bytes of keys for round 2 as well. Therefore, no key expansion operation is done

during the first two rounds.

At the beginning of rounds 3, 5, 7, 9, 11, 13; the key pipeline is completely dirty, and the bytes at $\mathsf{FF}^{\mathsf{K}}_{0:3}$ require 4 clock cycles, during which we need to run $\mathsf{ke2}^4$. Then, during the first 12 clock cycles of add, kxor is also enabled so that the following 12 dirty bytes can be refreshed in parallel.

The rounds 4, 6, 8, 10,12 work exactly same, except that the special key column requires the sequence ke0, $\mathsf{ke1}^3$ for update, instead of $\mathsf{ke2}^4$. The ciphertext is available in the last 16 rounds of the final round.

AES-256 Decryption. As the last 32 used bytes of key are loaded into the circuit, we use first 16 cycles to load the first half of this key. During the following 16 clock cycles, both the data (i.e. ciphertext) and the second half of the key are loaded at the same time, therefore we execute add simultaneously.

At the beginning of rounds 1, 3, 5, 7, 9, 11, 13; the first 16 bytes of the key pipeline are fresh and the rest are dirty. At the last 4 clock cycles of add, ikxor is performed through $\mathsf{FF}^{\mathsf{K}}_7$, $\mathsf{FF}^{\mathsf{K}}_{11}$, $\mathsf{FF}^{\mathsf{K}}_{15}$, so that 12 key bytes are refreshed. The following 4 bytes are also refreshed with the sequence ke0, $\mathsf{ke1}^3$.

The rounds 2, 4, 6, 8, 10, 12 work exactly same except the key column requiring update by key expand 2 is refreshed with the sequence $\mathsf{ke2}^4$ instead of ke0, $\mathsf{ke1}^3$. The plaintext is available in the last 16 clock cycles of the final round.

3.10 Hardware Evaluation

In order to perform a fair performance evaluation of our design, the circuit was synthesized following the method described in Section 2.3.2. Given that the circuit is rather comparatively large, we used the compiler directive `compile -exact_map -area_effort high`, which tells the synthesizer to prioritize area. We did not use `compile_ultra`, as all-in-one optimization approach takes very long time and therefore proves to be inefficient for our circuit. We outline some of the essential lightweight metrics of our architecture in Table 3.2.

In Figure 3.7, we present a component-wise breakdown of the circuit size when synthesized with the STM 90 nm logic process. A significant area is required for generating the control signals, as accommodating 6 different functionalities in a single circuit requires more fine-grained control over specific circuit components. This is because both the structure (w.r.t. the sequence of operations) and duration (w.r.t. the number of clock cycles) of a single round shows a wide range of variations, as the size of the key changes in AES.

Table 3.2 – Performance comparison of our AES architecture for 5 different technology libraries (where E denotes encryption, and D denotes decryption). Average power consumption is reported at a clock frequency of 10 MHz.

Instance	Area (μm^2)	Area (GE)	Power (μW) @ 10 MHz	Latency (cycles)	Energy (nJ/128-bit)	Throughput (Mbit/s)
STM 90 nm						
AES-128 (E)	16129	3674	194.7	243	4.73	72.07
AES-192 (E)				322	6.27	54.39
AES-256 (E)				371	7.22	47.21
AES-128 (D)				315	6.13	55.60
AES-192 (D)				400	7.79	43.79
AES-256 (D)				454	8.84	38.58
UMC 90 nm						
AES-128 (E)	15587	4971	160.7	243	3.91	61.94
AES-192 (E)				322	5.17	46.75
AES-256 (E)				371	5.96	40.57
AES-128 (D)				315	5.06	47.78
AES-192 (D)				400	6.43	37.63
AES-256 (D)				454	7.30	33.15
TSMC 90 nm						
AES-128 (E)	13431	4759	95.3	243	2.32	68.72
AES-192 (E)				322	3.07	51.86
AES-256 (E)				371	3.54	45.01
AES-128 (D)				315	3.00	53.01
AES-192 (D)				400	3.81	41.75
AES-256 (D)				454	4.33	36.78
NanGate 15 nm						
AES-128 (E)	1150	5848	40.1	243	0.97	1196.10
AES-192 (E)				322	1.29	902.64
AES-256 (E)				371	1.49	783.43
AES-128 (D)				315	1.26	922.70
AES-192 (D)				400	1.60	726.63
AES-256 (D)				454	1.82	640.20
NanGate 45 nm						
AES-128 (E)	4141	5189	310.2	243	7.54	167.45
AES-192 (E)				322	9.99	126.37
AES-256 (E)				371	11.51	109.68
AES-128 (D)				315	9.77	129.18
AES-192 (D)				400	12.41	101.73
AES-256 (D)				454	14.08	89.63

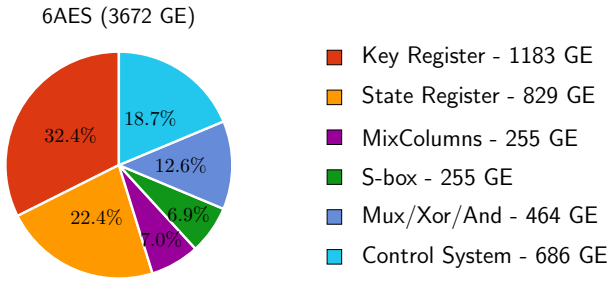

Figure 3.7 – The footprint of the sub-components of the circuit, obtained w.r.t. the technology library STM 90 nm.

3.11 Conclusion

In this chapter, we have presented an all-in-one AES circuit that supports six functionalities, i.e. these are both encryption and decryption for AES-128, AES-192 and AES-256. Compared to the smallest byte-serial AES-128 implementation reported by Banik et al. (with 2060 GE in STM 90 nm technology) [BBR16b], our results show that upgrading this circuit to support further AES-192 and AES-256 can be done with relatively small cost. The main design principle behind is that, starting from a serial AES-128 circuit with a goal to realize AES-192 and AES-256, the state pipeline without requiring many changes. On the other hand, the key pipeline needs to be extended by 128 flip-flops so that it can accommodate 256-bit key. Thanks to similarities in the key scheduling algorithms of AES family, most of the auxiliary circuit in the key pipeline of AES-128 can be reused. Hence, the main cost only comes from the additional flip-flops for the larger key, an observation we can make from the break down in Figure 3.7.

Naturally, with the circuit we presented, one could still derive a stand-alone combined or encryption-only AES-192 and AES-256 circuits. For instance, many of the candidates from NIST Post-Quantum Cryptography that rely on AES actually just need the encryption circuit for AES-256 [NISb]. For that, our results also provide an optimistic estimation for the size of encryption-only lightweight implementation of AES-256.

One of the takeaways is that the serialized implementation of AES in the form of ASIC spent larger portion of its silicon area for storage elements. Aligned with this observation, in Chapter 6, we will also see that, area-wise, the more internal storage elements a block cipher or AEAD scheme requires, the more heavyweight the implementation becomes.

4 Evaluation of ForkAES

In this chapter, we look at few architectural design choices for a lightweight implementation of the recently proposed forking cipher ForkAES, which is specifically conceived as a core primitive from which an AEAD scheme can be constructed [ARVV18].

The results presented in this chapter include the extension of the work done in collaboration with Subhadeep Banik [BB19a], which was presented in Indocrypt 2019. The authors were supported by the Swiss National Science Foundation (SNSF) through the Ambizione Grant PZ00P2_179921.

We briefly discuss the line of work which contributed to the design of ForkAES in Section 4.1 and lay out the details of our contributions in Section 4.2. By design, ForkAES requires storing an additional internal cipher state, and we discuss the pros and cons of introducing an extra register, as well as an alternative method to skip it, in Section 4.3. Then, we present the lightweight byte-serial implementation of ForkAES with small area goal in Section 4.4. In Section 4.5, we present the lightweight energy-efficient implementation of ForkAES. Finally, we conclude the chapter in Section 4.6.

The VHDL source code of our implementations can be found in a public git repository [FAE].

4.1 Related Work

In the past few years, lightweight cryptography has indeed become an important research discipline. A number of lightweight block ciphers like Clefia [SSA+07], and PRESENT [BKL+07] have become popular and have been well-studied with respect to their security and implementation. Both ciphers have been standardized in ISO/IEC 29192 [ISO12]. The Simon and Speck family of block ciphers [BSS+13] was proposed very recently by the researchers of the NSA with the goal of reducing hardware area. While the above ciphers have mostly targeted optimization of hardware area, there have been other block ciphers

aimed at optimizing other lightweight design metrics. One of the principal metrics among them is energy.

Through a lightweight standardization process, NIST is seeking AEAD schemes with one of the goals expressed as "optimized to be efficient for short messages (e.g., as short as 8 bytes)" [NISa]. Following the suit, ForkAES was proposed as the first example forkcipher construction by Andreeva et al. as an attempt to achieve a lightweight construction [ARVV18], which is in line with the design ideas from the TWEAKEY framework [JNP14]. Later, the round-reduced variant of ForkAES is cryptanalyzed by Banik et al. [BBJ+19]. Furthermore, the forking construction that is built on top of the SKINNY block cipher is also under submission to the NIST lightweight cryptography standardization project [ALP+19, NISa]. At the time of writing this thesis, this standardization is at the second round and ForkAE is one of the remaining candidates.

In a separate direction, the block cipher Midori was designed to specifically optimize energy consumption [BBI+15]. It has also been shown that for energy-efficient encryption of large quantities of data, stream-cipher-based constructions like Trivium perform much better [CP08, BMA+18]. The smallest combined encryption and decryption circuit, as well as the most energy-efficient round-based constructions of AES is studied by Banik et al. [BBR16b, BBR17].

4.2 Contributions

In this chapter, our contributions are as follows:

1. We show that it is possible to implement ForkAES without any additional storage elements other than those required to implement AES, if the AES circuit can perform both the encryption and decryption operations.

2. We extend upon the Atomic-AES architecture by Banik et al. [BBR16b], to realize ForkAES circuit that targets small area footprint. We implement ForkAES both with and without additional storage, and compare the area-latency trade-offs incurred in realizing the circuit. We conclude that area-wise, the smallest ForkAES implementation brings at least 35% more silicon footprint, compared to AES.

3. We extend upon the S3K2 architecture by Banik et al. [BBR17], in order to realize a ForkAES circuit with small energy consumption. Again, we implement ForkAES both with and without additional storage and compare the energy-latency trade-offs incurred in implementing the circuit. We further look at different component configurations to find out the most energy-efficient design. Our resulting implementations conclude that the energy-wise, ForkAES is at least two and half times more costly than AES.

4. We synthesize our ForkAES implementations with five different ASIC technology libraries, and report their area, latency, power and energy consumption measurements. Our original work had only taken the STM 90 nm library into account for measurements, but this thesis extends them to all five libraries for completeness.

As a result, we report the additional energy, latency and area cost introduced by ForkAES, in comparison to AES.

4.3 Removing Additional Storage

As described in detail in Section 2.2.6, ForkAES is a fork cipher, which is by design meant to be used as the core of an authenticated encryption primitive. In order to construct the latter, Andreeva et al. introduced two modes of operation, namely SAEF and PAEF, that can process arbitrary-size message and associated data blocks [ARVV18].

In order to execute the PAEF and SAEF modes of operation, a circuit must be able to execute the three algorithms: encryption E, decryption D^0 and reconstruction R^0. The remaining two algorithms, D^1 and R^1, are not used in these modes. Therefore, we omit the latter two algorithms in our circuit implementations.

ForkAES relies on a common state X^5 after five AES rounds, which is referred to as *the forking state*. We begin with a design that focuses on a smaller area footprint which completely avoids the extra register for storing the forking state. By doing so, we can handle encryption, decryption and reconstruction with single round function circuitry and avoid extra storage register, yet this comes at the expense of latency. In order to clearly understand the outcome of this trade-off, we then move on to another architecture that uses an additional register to store this 128-bit value temporarily.

With a more detailed look at encryption, decryption and reconstruction algorithms in Figure 2.2, we can observe that executing these three algorithms boils down to executing the round function and the key expansion of AES. Therefore, our hardware implementations can build on the state pipeline (Figure 3.1) and the key pipeline (Figure 3.2) implementations from our all-in-one AES architecture in Chapter 3, albeit with minor modifications.

Starting from an AES circuit that supports both encryption and decryption, let us first have a sketch idea of how to perform ForkAES encryption E, decryption D^0 and reconstruction R^0:

Encryption E As shown pictorially in Figure 4.1, encryption on an AES circuit would proceed as follows. In the first 5 rounds, the circuit would proceed in the forward direction, i.e. execute the forward key schedule function on the key register and the

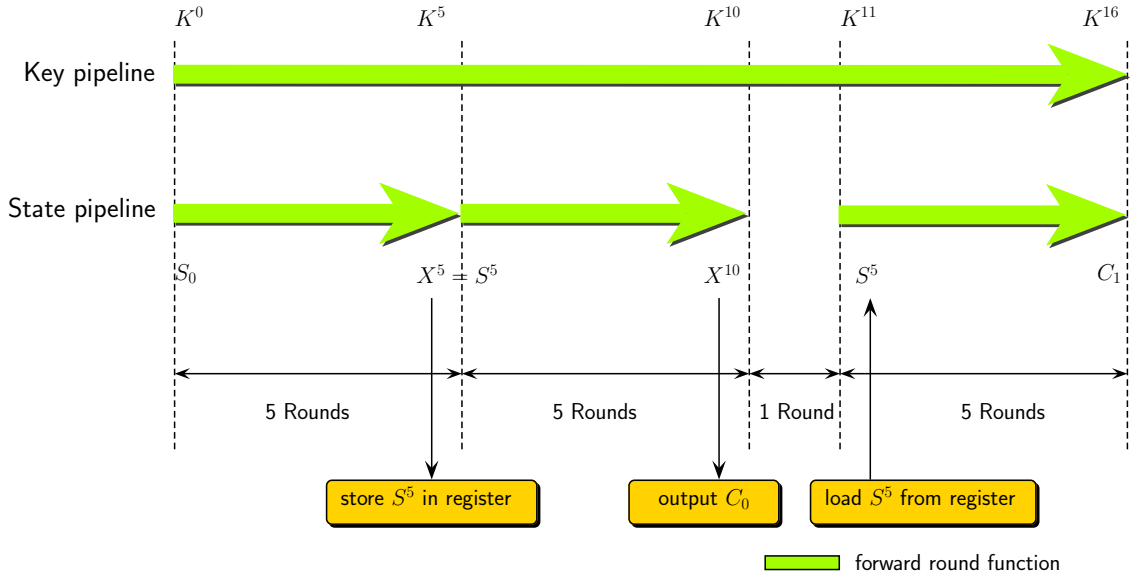

Figure 4.1 – Executing **E** on a generic AES circuit with an additional register.

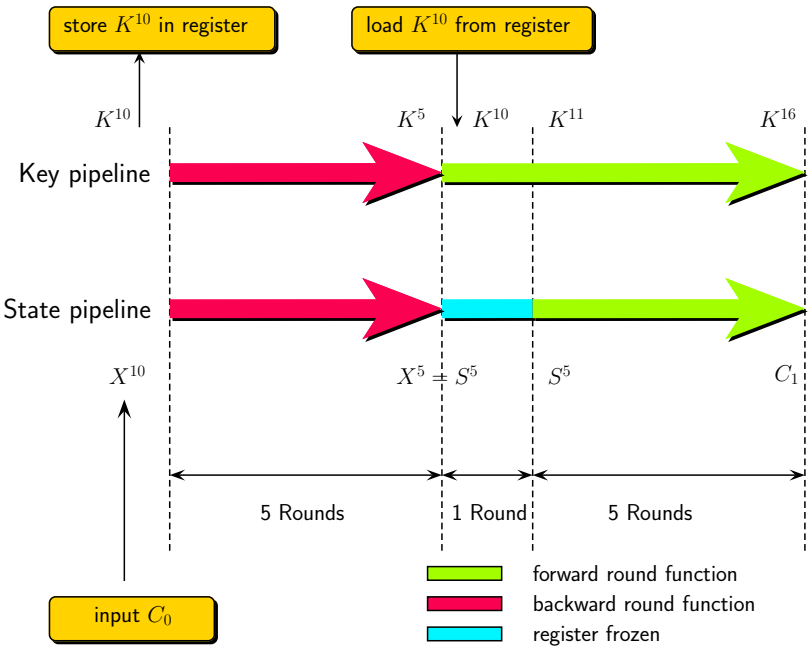

Figure 4.2 – Executing $\mathbf{R}^0$ on a generic AES circuit with an additional register.

forward AES round function on the state register. After this, the intermediate state $X^5 = S^5$ is stored in the additional register, parallelly while the circuit continues to execute the forward functions on both the key and state registers for another 5 rounds. At this point the the first ciphertext $C_0 = X^{10} \oplus K^{10} \oplus \tilde{T}$ is output from the state pipeline.

Thereafter, there needs to be one blank round in which the key registers execute the forward key schedule to compute the 12th round key K^{11}, during which the state registers could either be frozen using clock-gating techniques, or let to operate normally (it does not make any difference to the eventual circuit output). After this, the state S^5 that was stored in the extra register is loaded back on to the state registers and the circuit operates in the forward direction in both the state and key sides for another 5 rounds to output the second ciphertext block C_1.

Reconstruction R^0 The reconstruction function essentially outputs C_1 when the input is C_0. It would be executed as follows as per Figure 4.2. The initial inputs to the circuit are the ciphertext block $C_0 = X^{10} \oplus K^{10} \oplus \tilde{T}$ and the 11th roundkey K^{10}. We parallelly store K^{10} in the additional register and execute the inverse AES round functions and key schedule for 5 rounds. At this point the state and key registers store the intermediate states $X^5 = S^5$ and K^5 respectively. We freeze the state register for one round at this point and simultaneously load K^{10} that was stored in the additional register back on to key registers. After this round, the key registers compute the 12th roundkey K^{11} required to start the bottom branch of the reconstruction process. After this the state registers are unfrozen and both run in the forward direction for 5 more rounds to compute C_1.

We now try to show that both encryption and reconstruction can be performed on an AES circuit that additionally supports decryption.

Proposition 1. *Consider a (128/32/8/1-bit) serial circuit that can perform both* AES *encryption and decryption. Suppose that we complement this circuit by adding 64-bit register to store the 64-bit tweak value. Then it is possible to perform both* $\boldsymbol{E}$ *and* $\boldsymbol{R}^0$ *operations on such circuit without requiring any other additional storage elements.*

Sketch Proof. This proof idea is visualized in Figure 4.3. The AES circuit first runs for 10 rounds without interruption, and the ciphertext block $C_0 = X^{10} \oplus K^{10} \oplus \tilde{T}$ is output. Thereafter the circuit is operated in the backward direction for 5 rounds, i.e. the inverse AES round functions and key schedule operations are performed so that after five rounds, the circuit returns to having the forking state $X^5 = S^5$ in the state register and K^5 in the key register. At this point, we freeze the state registers for 6 rounds and let the key pipeline run in the forward direction for 6 rounds, so that the 12th roundkey K^{11} is computed. After this, both the state and key registers are run in the forward direction for 5 rounds, which ends up computing the ciphertext block C_1.

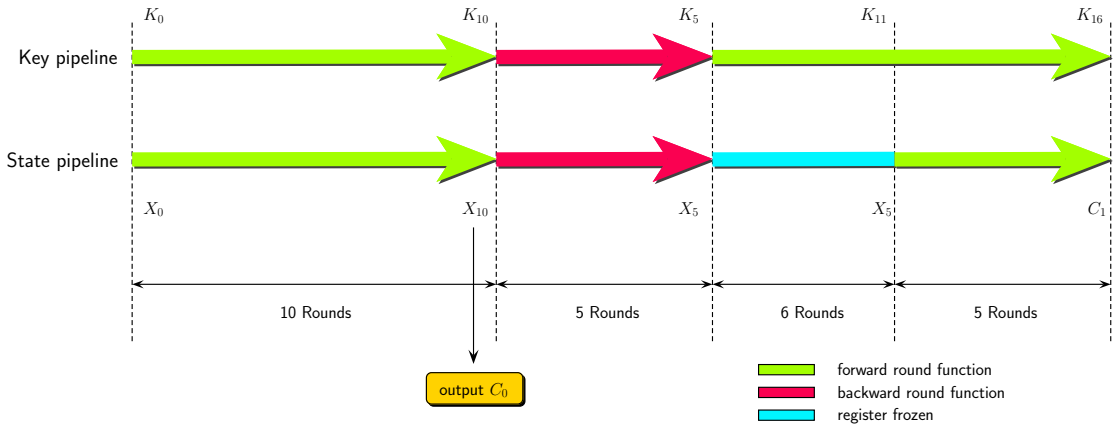

Figure 4.3 – Executing E on an AES circuit without an additional register

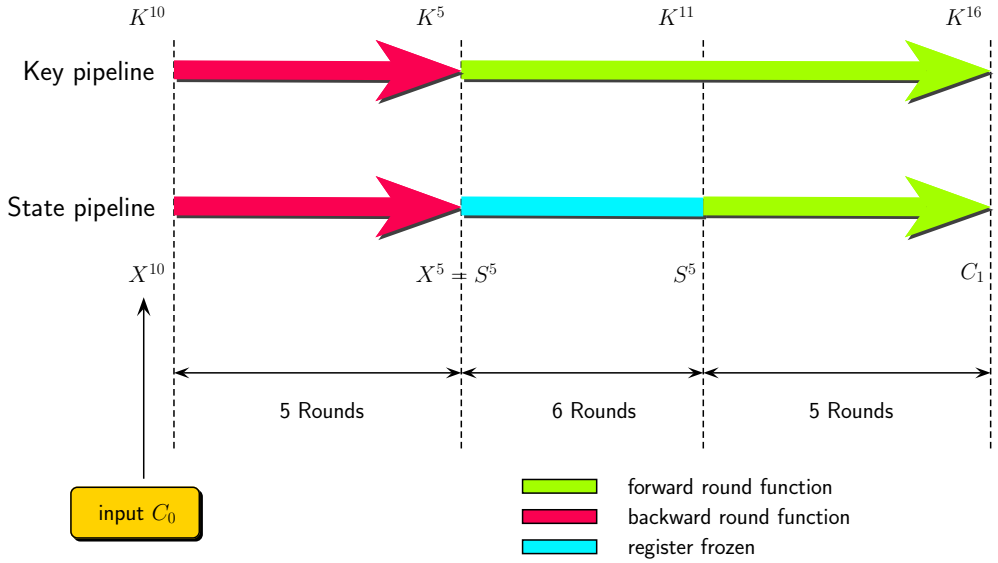

Figure 4.4 – Executing R^0 on an AES circuit without an additional register

Next, we look at reconstruction R^0. Reconstruction is essentially getting the circuit to output C_1, given C_0 and K^{10} as inputs. This is essentially how the circuit functions in the last 16 rounds in the encryption operation as is evident from Figures 4.3 and 4.4. □

4.4 Focusing on Area: Byte-serial ForkAES Architecture

The Atomic AES v2.0 architecture was proposed in [BBR16b]. It is an 8-bit serial circuit of AES that accommodates both encryption and decryption operations. For implementations whose goal is to minimize area, Atomic AES v2.0 is a good starting point, as it is the smallest byte-serial implementation in the literature.

The Atomic AES v2.0 has similar characteristics with our AES implementation in Chapter 3. One forward round is executed in 23 clock cycles and an inverse round is executed in 31 clock cycles. It occupies an area of only 2060 GE when implemented with the standard cell library of the STM 90 nm CMOS library and thus a very good candidate for a lightweight implementation of ForkAES both with and without the use of additional storage elements.

We first look at the design of ForkAES circuit without an additional register, and refer to this implementation as Conf_A. Before getting into circuit details of our implementation, let us look at the changes we need to make (starting from our AES circuit in Chapter 3) to accommodate the ForkAES operations:

1. The size of the key in ForkAES is fixed to 128 bits, therefore we remove the unnecessary flip-flops from the circuit, which were otherwise used for AES-192 and AES-256. Eventually, the key pipeline consists of 128 1-bit flip-flops.
2. We change the loading order of bytes for the key, plaintext and ciphertext blocks. Namely, we use row-major ordering. For example, given a 16-byte plaintext $B_0||B_1||\ldots||B_{15}$, the order for loading these bytes into the circuit is:

$$B_0, B_4, B_8, B_{12}, B_1, B_5, B_9, B_{13}, B_2, B_6, B_{10}, B_{14}, B_3, B_7, B_{11}, B_{15}$$

3. Additional 64-bit tweak register is added, in order to store the tweak and handle addition to the cipher state.

Besides these major changes, the bus and the controller are slightly modified. Those changes address the differences between the number of round in AES and ForkAES. The circuit is given in Figure 4.5.

We now look at register-level operations for a clearer picture of the movement of data in and out of the registers. Note that we do not delve into circuit-level details of how the AES round and key scheduling operate, as they were already explained in Chapter 3.

Now let us look at the sequence of operations necessary for E operation. Following the algorithmic description in Figure 2.2, we recall that S^i denotes the cipher state after i-th round, and in a similar fashion, X^i and Y^i represent the cipher states in diverged paths after the fork. In particular, $S^5 = X^5$ denotes the forking state, C_0 and C_1 denote the ciphertext blocks.

Cycles 0 to 15. The first 16 clock cycles are used for loading plaintext, and the key. The tweak is also loaded in the first 8 clock cycles simultaneously. AddRoundKey and SubBytes are also performed during loading.

Cycles 16 to 222. The next 207 clock cycles are used to execute 9 AES encryption

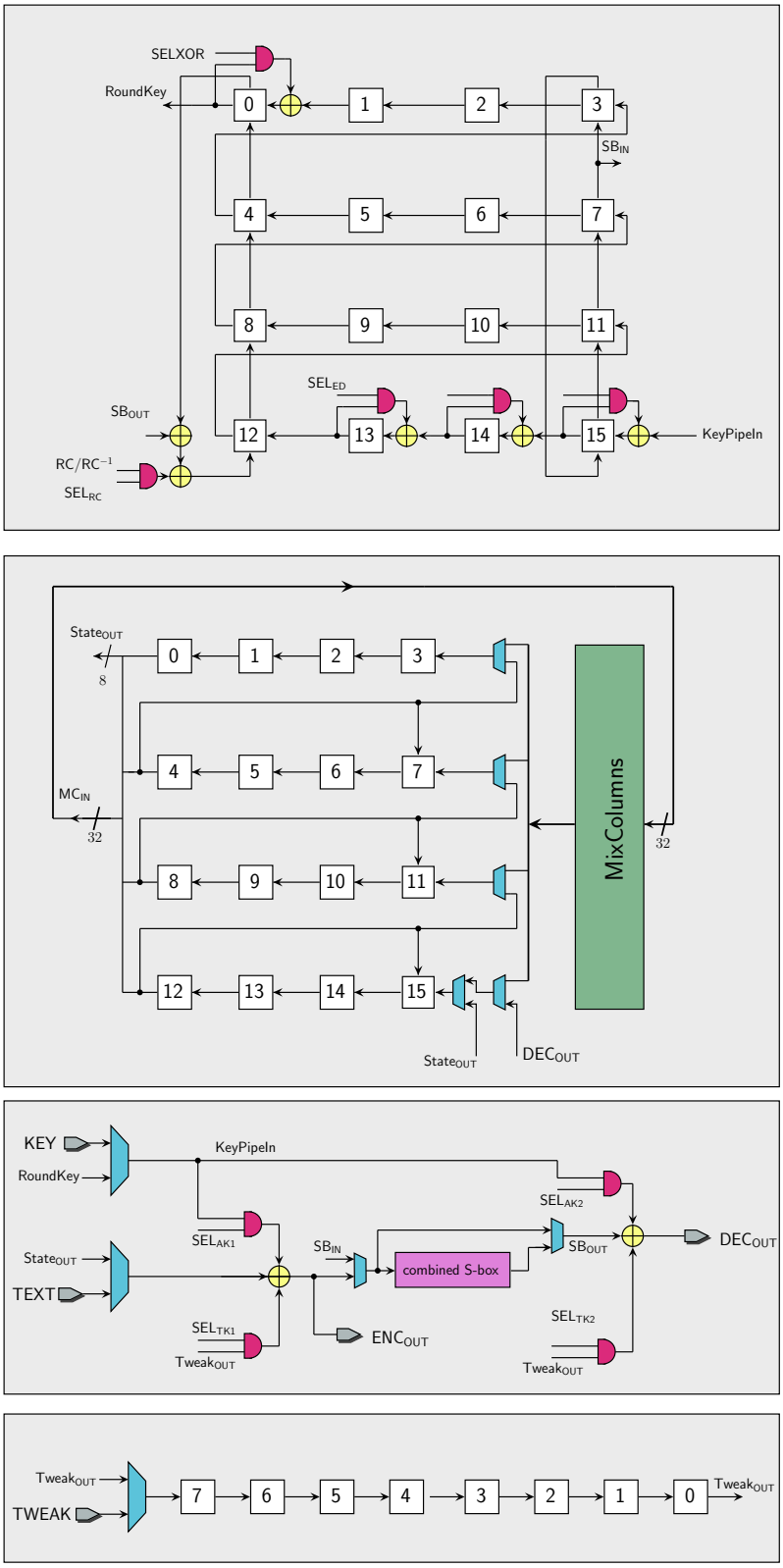

Figure 4.5 – ForkAES on the Atomic-AES v 2.0 circuit without an additional register.

rounds, with each round costing exactly 23 clock cycles. At the end, the state pipeline stores X^9.

Cycles 223 to 229. The next $3 + 4 = 7$ cycles are used to execute the 10th round ShiftRows (3 cycles) and the subsequent MixColumns (4 cycles). Thus the content of the state register at this point is $Z = \mathsf{MC} \circ \mathsf{SR} \circ \mathsf{SB}(X^9 \oplus K^9)$.

Cycles 230 to 245. These 16 cycles are used to do the final key addition to generate the first ciphertext block $C_0 = X^{10} = Z \oplus K^{10}$. At the same time the bytes coming out of the state register are fed back into the state register. At the same time, K^{10} is recycled back into the key pipeline.

Cycles 246 to 400. The next $31 \cdot 5 = 155$ cycles are used to perform 5 inverse AES round operations. Both the key and the state pipelines are reverted back to their values in the forking state.

Cycles 401 to 515. At this point of time, the state registers store the forking state $S^5 = X^5 = Y^5$. In order to compute the next state value Y^6, we need the round key K^{11}, which is not yet available in the key pipeline. Therefore the state pipeline is frozen through enabled flip-flops. The key registers store K^5, and so the next $5 \cdot 23 = 115$ cycles are used to operate the key schedule in the forward direction to compute K^{10}.

Cycles 516 to 538. The key registers function normally so that from cycles 523–538 the 12th round key K^{11} are available for key addition. The state registers are frozen until clock cycle 522. From cycles 523 to 538 the bytes are taken out of the state register, added to the individual bytes of K^{11}, passed through the S-box and driven back into the state registers. In this way, at the end of this set of cycles, the state registers hold $\mathsf{SB}(Y^5 \oplus K^{11})$, which is exactly the value required to operate the subsequent forward rounds.

Cycles 539 to 653. The next $5 \cdot 23 = 115$ cycles, 5 forward AES rounds are executed in a normal way, so that it is able to output the final ciphertext block C_1.

All the above description implicitly assumes that the tweak register essentially operates as a circularly shifting register that makes the tweak bytes available for addition when required. Note that decryption D^0 is also performed in a manner which is exactly the same as in our AES circuit, by merely following the sequence of operations in decryption and completing in exactly 326 clock cycles. As per Proposition 1, the reconstruction R^0 is simply achieved by executing the operations from clock cycles 230 to 653. Thus encryption, decryption and reconstruction takes 654, 326 and 424 cycles respectively. This completes the analysis for Conf_A.

The sequence of operations in the case with an extra register (call it Conf_B) is much simpler. The circuit does not need to recompute the forking state, as it can store it in

the extra 128-bit register. The encryption E can be done as follows.

Cycles 0 to 15. The first 16 clock cycles are used for loading plaintext, and the key. The tweak is also loaded in the first 8 clock cycles simultaneously. AddRoundKey and SubBytes are also performed during loading.

Cycles 16 to 130. The next $5 \cdot 23 = 115$ clock cycles are used to compute 5 rounds of AES encryption, until the forking state $S^5 = X^5$ is computed. Between clock cycles 115–130, the bytes leaving the state pipeline are also driven into the extra register, in order to store the forking state.

Cycles 131 to 245. The next $5 \cdot 23$ clock cycles are used to execute 5 more AES rounds, until the first ciphertext block C_0 is computed.

Cycles 246 to 268. In the following 23 clock cycles, the forking state is loaded into the state pipeline, and simultaneously, the key update is performed on the key pipeline.

Cycles 269 to 383. The next $5 \cdot 23$ clock cycles are used to compute C_1 from the forking state $S^5 = Y^5$. Between the clock cycles 368–383 the second ciphertext C_1 becomes available.

Note that the extra register can also be used to store the key in the reconstruction, therefore R also takes fewer clock cycles to complete. However, decryption remains exactly the same as in $\mathsf{Conf_A}$. Thus encryption, decryption and reconstruction takes 384, 326 and 309 clock cycles respectively. This completes the analysis for $\mathsf{Conf_B}$.

4.4.1 Byte-serial Implementation Results

With the presented byte-serial implementations, we achieve an implementation, namely $\mathsf{Conf_A}$ and $\mathsf{Conf_B}$ that cost 2781 GE and 3438 GE respectively, when implemented with the technology library STM 90 nm. The $\mathsf{Conf_A}$ is 35% larger compared to the Atomic AES v2.0 by Banik et al. [BBR16b]. The space for further area optimization is possible, with the use of manual-labour clock-gating technique. However, clock-gating comes at the expense of timing violations, which requires the implementor to handle those violations case-by-case for each technology library. The detailed measurements on these two byte-serial implementations are tabulated in Table 4.1.

4.5 Focusing on Energy: Round-based ForkAES Architecture

During its functionality, the energy spent by a circuit can be divided into two parts: leakage energy and dynamic energy. The former roughly scales with the number of gates

Table 4.1 – Performance comparison of our ForkAES architecture for 5 different technology libraries. Average power consumption is reported at a clock frequency of 10 MHz.

Circuit + Mode	Area (μm^2)	Area (GE)	Power (μW) @ 10 MHz	Latency (cycles)	Energy (nJ/128-bit)	Throughput (Mbit/s)
STM 90 nm						
$\mathsf{Conf_A}$ ($\mathbf{E}$)	12209	2781	138.3	654	9.04	21.96
$\mathsf{Conf_A}$ ($\mathbf{D}^0$)				326	4.51	44.05
$\mathsf{Conf_A}$ ($\mathbf{R}^0$)				424	5.86	33.87
$\mathsf{Conf_B}$ ($\mathbf{E}$)	15095	3438	155.7	384	5.98	32.94
$\mathsf{Conf_B}$ ($\mathbf{D}^0$)				326	5.08	38.80
$\mathsf{Conf_B}$ ($\mathbf{R}^0$)				309	4.81	40.94
UMC 90 nm						
$\mathsf{Conf_A}$ ($\mathbf{E}$)	10330	3294	108.7	654	7.11	20.79
$\mathsf{Conf_A}$ ($\mathbf{D}^0$)				326	3.54	41.70
$\mathsf{Conf_A}$ ($\mathbf{R}^0$)				424	4.61	32.06
$\mathsf{Conf_B}$ ($\mathbf{E}$)	12727	4058	130	384	4.99	34.07
$\mathsf{Conf_B}$ ($\mathbf{D}^0$)				326	4.24	40.13
$\mathsf{Conf_B}$ ($\mathbf{R}^0$)				309	4.02	42.34
TSMC 90 nm						
$\mathsf{Conf_A}$ ($\mathbf{E}$)	9330	3306	75.05	654	4.91	26.36
$\mathsf{Conf_A}$ ($\mathbf{D}^0$)				326	2.45	52.89
$\mathsf{Conf_A}$ ($\mathbf{R}^0$)				424	3.18	40.67
$\mathsf{Conf_B}$ ($\mathbf{E}$)	12014	4257	99.97	384	3.84	42.96
$\mathsf{Conf_B}$ ($\mathbf{D}^0$)				326	3.26	50.60
$\mathsf{Conf_B}$ ($\mathbf{R}^0$)				309	3.09	53.39
NanGate 15 nm						
$\mathsf{Conf_A}$ ($\mathbf{E}$)	841	4277	30.32	654	1.98	391.31
$\mathsf{Conf_A}$ ($\mathbf{D}^0$)				326	0.99	785.03
$\mathsf{Conf_A}$ ($\mathbf{R}^0$)				424	1.29	603.58
$\mathsf{Conf_B}$ ($\mathbf{E}$)	1066	5420	37.34	384	1.43	632.01
$\mathsf{Conf_B}$ ($\mathbf{D}^0$)				326	1.22	744.45
$\mathsf{Conf_B}$ ($\mathbf{R}^0$)				309	1.15	785.41
NanGate 45 nm						
$\mathsf{Conf_A}$ ($\mathbf{E}$)	2998	3757	267.05	654	17.46	79.77
$\mathsf{Conf_A}$ ($\mathbf{D}^0$)				326	8.71	160.03
$\mathsf{Conf_A}$ ($\mathbf{R}^0$)				424	11.32	123.04
$\mathsf{Conf_B}$ ($\mathbf{E}$)	3803	4766	327.07	384	12.56	131.91
$\mathsf{Conf_B}$ ($\mathbf{D}^0$)				326	10.66	155.38
$\mathsf{Conf_B}$ ($\mathbf{R}^0$)				309	10.11	163.93

constituting the circuit, where each gate is associated with a constant power leakage due to its implementation in the CMOS technology. The latter, on the other hand, essentially stems from state changes of wires, as each component of the circuit receives and further propagates glitches, until both its input and output values are stabilized. This repeats each time the inputs of components change that coincides with the rising edge of the clock signal.

Hence, minimizing the circuit size does not necessarily align with the goal of reducing energy consumption. Following the work of Banik et al. [BBR17], a circuit that performs one round of AES per clock cycle leads to the most energy efficient design. This particular round-based implementation is called S3K2 architecture. Then the question that follows is how can we transform that particular round-based AES circuit to obtain most energy-efficient implementation for ForkAES. As converting a plain AES architecture that supports both decryption and encryption into ForkAES circuit reveals a number of free design choices, we consider and compare each one of the possible designs in the following section.

4.5.1 Generic Architecture

On a higher level, we propose and implement few round-based ForkAES architectures. Following the approach taken by Banik et al. [BBR17], each variant is obtained by applying an incremental change on the particular component of the design, with the hope that a smaller energy consumption metric can be achieved. While the original work relied only on the STM 90 nm technology library, this thesis takes all five technology libraries mentioned in Section 2.3.1 into account. Our intuition behind the incremental-step strategy is to find the most energy-efficient design so that we can establish a fair comparison with the most energy-efficient AES circuit.

The following summary of the design refers to the most energy-efficient design on average and it is obtained through a combination of compartmentalized components SC#2, KC#1, TC, sg explained below. Further modifications we make lead to slight changes in the precise description of these components and as well as the main circuit as seen in Figure 4.6. We present the power and energy consumption results of the modifications in Table 4.2.

In comparison to Atomic AES v2.0 that uses 8-bit data and key path, the designs below utilize 128-bit data and key paths. With few exceptions, these round-based circuits consist of three components that respectively handle the cipher state, the key scheduling and the temporary storage for the forking state. Below, we summarize these components individually.

State component. We will use SC#1 from Figure 4.6 for the following functionality description. This component consists of three parts:

- At its core, 128-bit $\mathsf{Register}_{\mathsf{St}}$ is used to keep the cipher state after each ForkAES

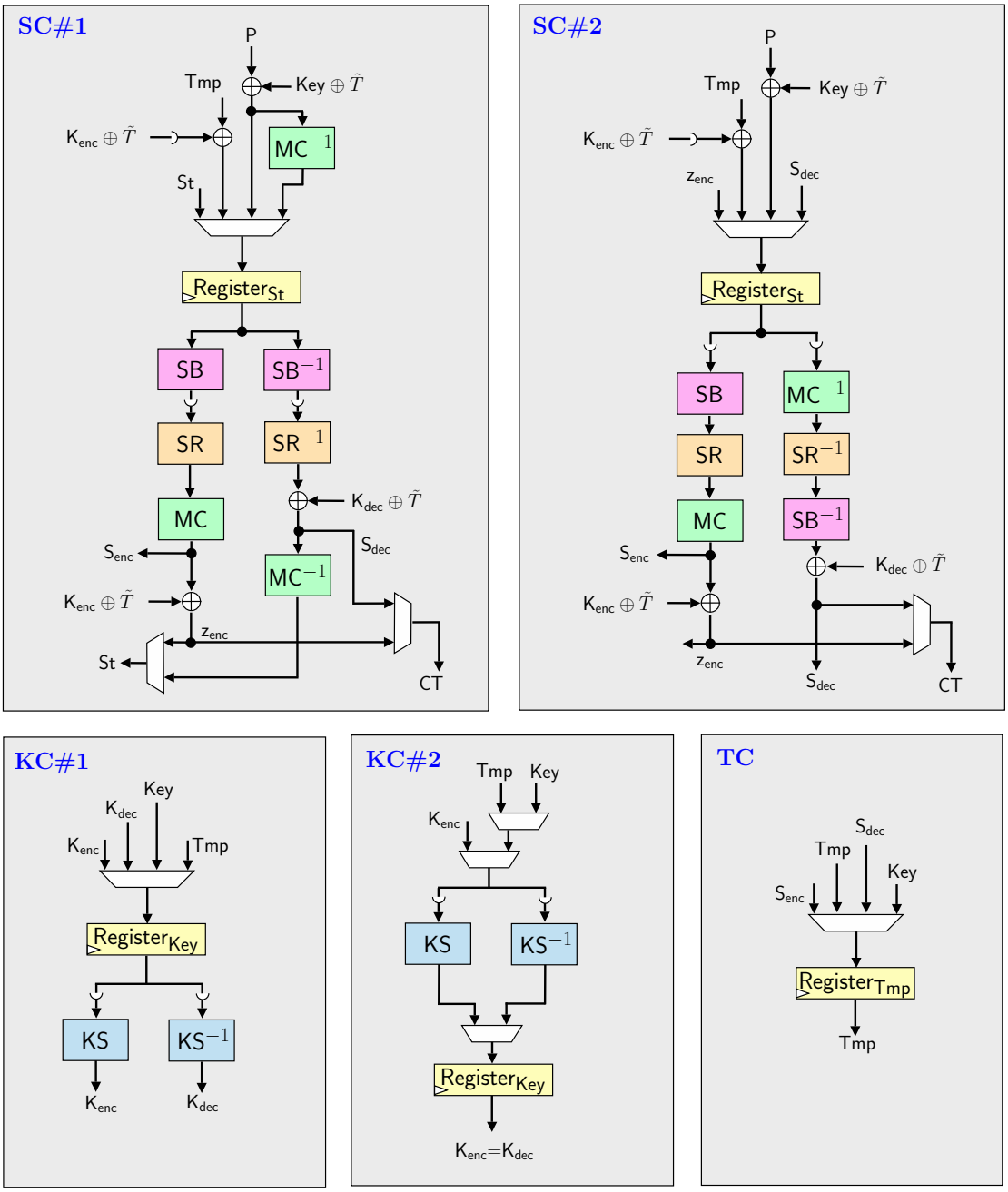

Figure 4.6 – The state components SC#1, SC#2; the key components KC#1, KC#2, and the temporary register component TC of ForkAES circuit.

round. At the rising edge of the clock, its content is updated to the next state with the help of the MUX described below.

- The MUX placed at the input of $\mathsf{Register}_{\mathsf{St}}$ serves three functions, by selecting which value should be loaded into this register. First, it can load the next plaintext/ciphertext state from the wire St, after it is computed by the round function circuit. Secondly, it can load the initial state, e.g. S^0 during encryption. And lastly, it can load the contents of the temporary register $\mathsf{Register}_{\mathsf{Tmp}}$.

- Round function bus consists of two series of 128-bit combinatorial circuits arranged to perform either the forward round function or its inverse, as well as the tweaked key addition. This dual-function circuit is complemented with masking AND gates (denoted with the symbol $\multimap\!\!\rightarrow$) that disable the unused part of the circuit, i.e. either the encryption or the decryption path, to reduce energy consumption. The final output of the circuit is selected by the output MUX.

Key Component (KC). The key component KC works in a quite similar fashion to the state component. In particular, we will describe KC#1 from Figure 4.6, which consists of three parts:

- 128-bit $\mathsf{Register}_{\mathsf{Key}}$ is used to keep the current round key (more precisely it keeps K^{i-1} at round i). It is updated with the rising edge of the clock.

- The MUX wired to the input of $\mathsf{Register}_{\mathsf{Key}}$ supports three different basic operations, by selecting which value to load into the register. First, it can load the next round key computed by the key schedule circuit. Secondly, it can initialize the register during cycle 0, e.g. load K^0 during encryption. And lastly, it can load the content of $\mathsf{Register}_{\mathsf{Tmp}}$.

- The key schedule consists of two series of 128-bit combinatorial circuits arranged to perform either the forward key schedule function KS or its inverse KS^{-1}. This dual circuit is also complemented with masking AND gates ($\multimap\!\!\rightarrow$) that disable the unused part of the circuit for energy efficiency. The actual round key that the state component needs is provided through either $\mathsf{K}_{\mathsf{enc}}$ or $\mathsf{K}_{\mathsf{dec}}$ based on the actual ForkAES operation the circuit is performing.

Temporary (Register) Component (TC). It consists of two parts (see TC in Figure 4.6):

- 128-bit $\mathsf{Register}_{\mathsf{TC}}$ is used to keep a temporary 128-bit value. This is either the state S^5 used at fork (see Figure 2.1) or the round key K^{10} loaded to the circuit during reconstruction operation.

- The MUX wired to the input of $\mathsf{Register_{TC}}$ supports three basic operations, by selecting which value to load into the register. First, it can maintain its content through reloading from itself. Secondly, it can initialize the register with the round key K^{10}. And lastly, it can load the forking state S^5 from the state component.

Below, we describe how encryption is done with the particular ForkAES architecture that combines components SC#1, KC#1, TC. We use the series of variables S^i, z^i, K^i, which respectively define the cipher state, the round key and the cipher state just after the round key and tweak addition in the i-th round (see the algorithmic descriptions in Figure 2.2).

Cycle 0. On SC#1, AddRoundKey (with tweak) is done on plaintext, and the result $z^1 = S^0 \oplus K^0 \oplus \tilde{T}$ is loaded into $\mathsf{Register_{St}}$ through MUX. On KC#1, the initial key K^0 is loaded into $\mathsf{Register_{Key}}$ without any operation.

Cycles 1 to 4. At the very beginning of cycle i, $\mathsf{Register_{St}}$ holds z^i. Then during cycle i, $S^i \leftarrow \mathsf{SB}(\mathsf{SR}(\mathsf{MC}(z^i)))$ is computed through encryption path and the round key addition follows it: $z^{i+1} \leftarrow S^i \oplus K^i \oplus \tilde{T}$. As $\mathsf{Register_{Key}}$ holds K^{i-1} at the beginning of clock cycle i, the round key K^i appears at the wire $\mathsf{K_{enc}}$ after being computed by KS circuit of KC#1 and the result is passed to the encryption path via $\mathsf{K_{enc}}$ as seen in Figure 4.6. Also, K^i is loaded into $\mathsf{Register_{Key}}$.

Cycle 5. Works similar to cycles 1 to 4. The only difference is that the forking state S^5 from the encryption path is stored into the temporary register $\mathsf{Register_{Tmp}}$.

Cycles 6 to 9. Similar to cycles 1 to 4.

Cycle 10. Works similar to cycles 1 to 4. The difference is that C_0 becomes available at the output wire CT during this clock cycle. Also, the control bits of MUX before $\mathsf{Register_{St}}$ is set to load the forking state S^5 for the next clock cycle from the temporary register $\mathsf{Register_{Tmp}}$.

Cycle 11. At the beginning of this cycle, $\mathsf{Register_{St}}$ receives $v^6 = S^5 \oplus K^{11} \oplus \tilde{T}$. Similar to cycle 1, the computation $Y^6 \leftarrow \mathsf{SB}(\mathsf{SR}(\mathsf{MC}(v_6)))$ is done first, and then the key addition: $v^7 \leftarrow Y^6 \oplus K^{12} \oplus \tilde{T}$. v^7 is stored back into $\mathsf{Register_{St}}$, and the round key K^{12} is stored into $\mathsf{Register_{Key}}$.

Cycles 12 to 15. Similar to cycles 1 to 4.

Cycle 16. Similar to cycle 10, with the difference that C_1 becomes available at CT.

Below we describe how ForkAES reconstruction is performed by the circuit, which involves some parts of encryption and decryption operations. We assume that at the beginning of the operation, the ciphertext C_0 is loaded from the wire P, and the round key K^{10} is loaded from the wire Key in Figure 4.6.

Cycle 0. On SC#1, AddRoundKey (with tweak) and the inverse MixColumns MC^{-1} are computed on the ciphertext C_0, and the result $X_{10,\mathsf{SR}}$ is loaded into $\mathsf{Register}_{\mathsf{St}}$ through MUX. On KC#1, the initial key K_{10} is loaded both into $\mathsf{Register}_{\mathsf{Key}}$ and $\mathsf{Register}_{\mathsf{Tmp}}$ without any operation.

Cycles 1 to 4. At the beginning of cycle i, $\mathsf{Register}_{\mathsf{St}}$ holds $\mathsf{MC}^{-1}(X^{11-i})$. Then during cycle i, $u^{11-i} \leftarrow \mathsf{SB}^{-1}(\mathsf{SR}^{-1}(z^i))$ is first computed through decryption path[1] and the round key addition follows it: $X^{10-i} \leftarrow u^{11-i} \oplus K^{10-i} \oplus \tilde{T}$. And finally, $\mathsf{MC}^{-1}(X^{10-i})$ is computed and stored in the register. In the same fashion, at the beginning of the clock cycle i, $\mathsf{Register}_{\mathsf{Key}}$ holds K^{11-i}, hence the round key K^{10-i} is calculated with the combinatorial KS^{-1} circuit of KC#1 and the result is passed to the decryption path via $\mathsf{K}_{\mathsf{dec}}$ as seen in Figure 4.6; and also loaded back into $\mathsf{Register}_{\mathsf{Key}}$.

Cycle 5. Works similar to cycles 1 to 4. The difference is that the forking state S^5 from the decryption path appears at $\mathsf{S}_{\mathsf{dec}}$ and hence it is loaded into the temporary register $\mathsf{Register}_{\mathsf{Tmp}}$ at the end of this clock cycle. Moreover, the round key K^{10} is loaded back into $\mathsf{Register}_{\mathsf{Key}}$ from $\mathsf{Register}_{\mathsf{Tmp}}$.

Cycle 6. No decryption or encryption operation is done on SC#1, because an operation that must follow is a round key addition (see Figure 2.1). Therefore, the forking state S_5 is read from $\mathsf{Register}_{\mathsf{Tmp}}$ and the round key addition is done on the wire: $v_6 \leftarrow S^5 \oplus K^{11} \oplus \tilde{T}$, where the round key K^{11} is computed with KS circuit in KC#1. The result v^6 is loaded into $\mathsf{Register}_{\mathsf{St}}$.

Cycles 7 to 11. Works similar to cycles 12 to 16 of ForkAES encryption operation above, and the result C_1 becomes available at clock cycle 11.

We skip the description of decryption, as it can be easily constructed by repeating the cycles 1 to 4 of ForkAES reconstruction above.

4.5.2 Modified Implementations

We explore possible modifications to the generic circuit, and compare their results in Table 4.2. In order to derive a single metric for strict comparison, we take the equal-weight average of energy consumed by each ForkAES operation: encryption E, decryption D^0 and reconstruction R^0. In our measurements, we first compute the average latency of the circuit for encryption, decryption and reconstruction. This average latency is then used to compute the average energy as well as the average maximum throughput. The resulting metric is used for deciding which design performs better in terms of energy efficiency.

Our choice of this metric is justified by the fact that the proposed modes of operation SAEF and PAEF by Andreeva et al. make the following number of ForkAES calls for processing a message of m blocks and an associated data of a blocks [ARVV18]:

[1] Note that $\mathsf{SB}^{-1}(\mathsf{SR}^{-1}(x)) = \mathsf{SR}^{-1}(\mathsf{SB}^{-1}(x))$ for all x.

- encryption: $(m + a) \cdot \mathsf{E}$,
- decryption: $a \cdot \mathsf{E} + m \cdot \mathsf{D}^0 + m \cdot \mathsf{R}^0$.

Hence the average energy spent per message block roughly converges to our metric if $m \gg a$. This metric omits the additional higher-level circuitry such as control logic that handles multiple associated data and message blocks in SAEF and PAEF, as we only focus on the ForkAES implementation.

Self-gating. One might notice that during encryption the control bits and contents of $\mathsf{Register}_{\mathsf{Tmp}}$ is irrelevant for 12 clock cycles, and used as a storage for 4 clock cycles. Similarly, during reconstruction, the $\mathsf{Register}_{\mathsf{Tmp}}$ stores its value for many cycles without receiving a new value. The register preserves its value through a MUX that feeds the register's own value back into its input (see TC in Figure 4.6). Hence one might wonder whether using an enabled flip-flops for this register yields better results. In that case, we could use the enable signal to freeze the register, instead of reloading it with the same value multiple times. We implemented this version. This incremental change on top of the generic design (SC#1, KC#1, TC) is given in Table 4.2 as SC#1, KC#1, TC, sg. Overall, this results in a slight reduction in the average power consumption, which directly translates into reduction in the energy consumption too.

Reorganized decryption path. One of the benefits of the state component SC#1 (see Figure 4.6) is that both SB and SB^{-1} has the same input, which allows them to be implemented as a single circuit and share a demultiplexer. This idea is due to Banik et al. [BBR16a]. As a disadvantage, this design requires an extra MC^{-1} circuit attached to the input wire P, as ForkAES does not skip a MixColumns operation at the last round in contrast to the original AES-128. In order to understand this trade-off better, we compare it with another state component design, i.e. SC#2. The latter organizes MC^{-1}, SR^{-1}, SB^{-1} circuits in a more intuitive fashion in the decryption path, and eliminates the need to append an extra MC^{-1} to the input (see Figure 4.6). In conclusion, this leads to a slightly better implementation, as reported in Table 4.2, because the energy consumption caused by duplication of some S-box circuitry is smaller than what is brought by the additional MC^{-1} circuity.

Removing temporary register. We have shown in Proposition 1 that even without a temporary register to store the forking state S^5, one can still realize ForkAES operations. This would apparently require more clock cycles, and therefore more energy. In order to understand this trade-off, we consider the design that is a combination of SC#1, KC#1, sg without temporary component. We still micromanage the key and state registers through the use of self-gating, as there are few clock cycles in which they become inactive. It can be seen in the Table 4.2 that even though it has the least power consumption in two technology libraries, this design performs poorly in terms of energy efficiency than its counterparts because of the incurred latency. Following our intuition, this design decision

has the most meaningful impact, both in terms of latency and energy consumption.

Flipped key scheduler. Our final tweaked design is based on the following observation: during each clock cycle, the round key is computed either through KS or KS^{-1} circuit. Because it takes a brief amount of time for these circuits to compute the final round key, the output wires $\mathsf{K}_{\mathsf{enc}}$ and $\mathsf{K}_{\mathsf{dec}}$ propagate glitches into SC#1 circuit. That is due to the fact that $\mathsf{Register}_{\mathsf{Key}}$ actually stores the previous round key instead of the exact round key needed by the state component. In comparison, if the key component were to be updated as such that the particular round key was stored in the key register precisely when it was needed by the state component, then $\mathsf{K}_{\mathsf{enc}}$ and $\mathsf{K}_{\mathsf{dec}}$ would be glitch-free. The modified key component is given as KC#2 in Figure 4.6. Depending on the technology library, this modification might increase or decrease the average energy consumption.

4.5.3 Round-based Implementation Results

The area, power and energy measurements of all our implementations are reported in Table 4.2. In order to further reason with the underlying causes for excessive power consumption, we also indicate the amount of power dissipated due to leakage in addition to the total power consumption. The ratio of leakage power is typically dependent on the technology library, and indicates how carefully cells are optimized for low power. It is evident that NanGate 15 nm and NanGate 45 nm libraries suffer significantly when it comes to losing greater portion of power directly from cell leakages, which builds up proportionately as the area of the circuit grows. On the other hand, the commercial TSMC 90 nm library is able to maintain this leakage ratio close to %5.

It is also worth noticing that the round-based S3K2 implementation, upon which our results build on, consumes 0.484 nJ of energy for processing 128-bit plaintext block on average, according to UMC 90 nm library. Therefore, one of our take away contribution in this work is that ForkAES is almost two and half times more expensive when it comes to energy-efficiency (with 1.17 nJ per 128-bit), as the very idea of forking the cipher state greatly disrupts the arrangement of subcomponents in hardware.

4.6 Conclusion

From an efficiency point of view, the key insight behind the design of forking cipher is that the produced secondary ciphertext block can be directly used for authentication purposes. Therefore, it removes an extra mechanism in the mode of operation to handle authentication. In comparison, AES-GCM uses expensive multiplications over $\mathsf{GF}(2^{128})$ to authenticate blocks. In short, one can say that the authentication is handled with this amortized cost of half block cipher call. This chapter looked at whether this intuition translates into lightweight metrics of energy and area, when forking cipher is realized as

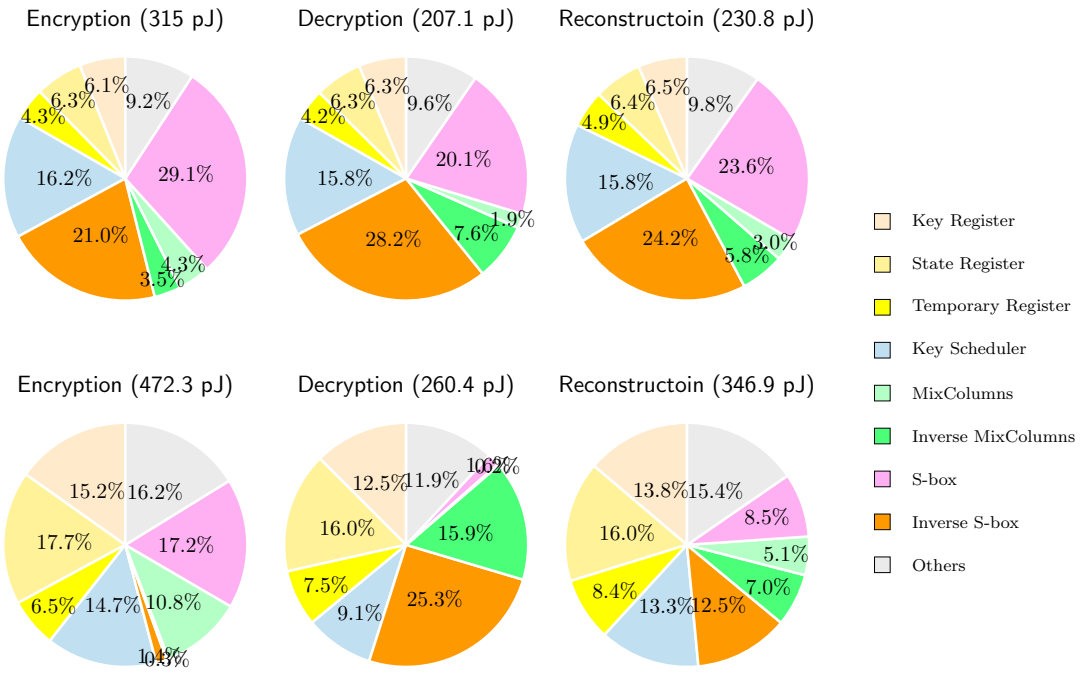

Figure 4.7 – Component-wise breakdown of the energy consumption of the most energy-efficient architecture (SC#2, KC#1, TC, sg configuration) during ForkAES encryption, decryption and reconstruction operations. The breakdown of power consumption is obtained from the NanGate 15 nm (top) and TSMC 90 nm (bottom) technology libraries.

Table 4.2 – Performance comparison of our energy-efficient round-based ForkAES architecture for 5 different technology libraries. Average power consumption is reported at a clock frequency of 10 MHz.

Circuit + Mode	Area		Power (μW)		Latency	Energy	Throughput
	(μm^2)	(GE)	Total	Leakage	(cycles)	(nJ/128-bit)	(Mbit/s)
STM 90 nm							
SC#1, KC#1, TC	119483	27214	775.8	172.8	40/3	1.03	1770.9
SC#1, KC#1, TC, sg	120880	27533	752.1	173.3	40/3	1.00	1816.6
SC#2, KC#1, TC, sg	137289	31270	**729.8**	211.8	40/3	**0.97**	1952.1
SC#1, KC#1, sg	**117657**	26799	732.3	170.4	17	1.24	1624.6
SC#1, KC#2, TC	119378	27191	760.2	172.9	40/3	1.01	2222.2
UMC 90 nm							
SC#1, KC#1, TC	109302	34854	940.8	432.6	40/3	1.25	1182.9
SC#1, KC#1, TC, sg	110422	35211	939.3	433.6	40/3	1.25	1195.2
SC#2, KC#1, TC, sg	120987	38580	**878.6**	427.6	40/3	**1.17**	1334.6
SC#1, KC#1, sg	**107941**	34420	898.1	425.8	17	1.53	957.4
SC#1, KC#2, TC	108550	34614	892.4	432.8	40/3	1.19	1469.6
TSMC 90 nm							
SC#1, KC#1, TC	89296	31638	301.5	13.6	40/3	0.40	1845.9
SC#1, KC#1, TC, sg	90192	31956	298.5	13.7	40/3	0.40	2244.0
SC#2, KC#1, TC, sg	100127	35476	**273.7**	13.7	40/3	**0.36**	1899.5
SC#1, KC#1, sg	**87646**	31054	278.1	13.3	17	0.47	1405.2
SC#1, KC#2, TC	88939	31512	282.6	13.6	40/3	0.38	2403.0
NanGate 15 nm							
SC#1, KC#1, TC	7063	35926	192.7	132.8	40/3	0.26	29065.1
SC#1, KC#1, TC, sg	7184	36539	194.0	134.1	40/3	0.26	28700.7
SC#2, KC#1, TC, sg	7796	39651	189.1	133.9	40/3	**0.25**	30018.1
SC#1, KC#1, sg	**6936**	35278	**185.4**	130.6	17	0.32	23238.8
SC#1, KC#2, TC	7060	35911	196.3	133.3	40/3	0.26	36331.4
NanGate 45 nm							
SC#1, KC#1, TC	26122	32734	2578.3	2348.6	40/3	3.44	4238.7
SC#1, KC#1, TC, sg	26454	33150	2583.5	2351.8	40/3	3.44	4180.6
SC#2, KC#1, TC, sg	29399	36841	2564.7	2357.6	40/3	**3.42**	4488.0
SC#1, KC#1, sg	**25606**	32087	**2523.2**	2308.6	17	4.29	3435.8
SC#1, KC#2, TC	26167	32791	2620.4	2376.9	40/3	3.49	5417.5

ASIC.

As pointed out in Section 4.4.1, $\mathsf{Conf_A}$ (resp. $\mathsf{Conf_B}$) implementation only brings extra 35% (resp. 67%) footprint into the circuit. Therefore, according to the ASIC area metric, we can verify that the aforementioned design intuition holds. However, on the energy direction, as pointed out in Section 4.5.3, the energy consumption grows to more than two and half that of AES. Therefore, forking paradigm is not a very good candidate for energy concerned lightweight applications.

5 Introduction to Swap-and-Rotate Technique

The results presented in this chapter is based on the work done in collaboration with Subhadeep Banik, Francesco Regazzoni and Serge Vaudenay [BBRV20]. It was presented in FSE 2020. The authors Fatih Balli and Subhadeep Banik were supported by the Swiss National Science Foundation (SNSF) through the Ambizione Grant PZ00P2_179921.

We summarize the previous work on byte and bit-serial block cipher implementations in Section 5.1. We give the details of our contributions in Section 5.2. In Section 5.3, we remind the preliminary definitions and notations (along with a brief sketch of the proofs presented in [Con13]). The main mathematical background is built on top of the single-swap setting, which is presented in Section 5.4. The theory built up in this section is done in various stages. In each stage, we try to decrease the number of permutations required to describe the PRESENT bit permutation. The same techniques are also applied to GIFT in parallel. Then, we move on to multi-swap setting in Section 5.5. And lastly a final optimization step is performed to match the number of clock cycles exactly with the block size in Section 5.6. With these results, we conclude the chapter and defer the actual circuit descriptions to Chapter 6.

5.1 Related Work

The block cipher family Katan [CDK09] (whose precursor was the stream cipher Trivium [CP08]) and then later Simon [BSS+] were in some sense aimed to achieve a lower limit of lightweight encryption in terms of area occupied in silicon. Both these ciphers have shift-register-based update functions, which is efficient to implement in ASIC when the length of datapath is reduced to one bit.

In CHES 2017, Jean et al. presented the concept of *bit-sliding* [JMPS17], in which byte and nibble oriented block ciphers like AES [NIS01], PRESENT [BKL+07] and SKINNY [BJK+16] were implemented in hardware by updating only one bit per clock cycle. The main idea behind these constructions was to reformulate the linear layer for these ciphers

so that they require fewer scan flip-flops, which have built-in multiplexer functionality at the input port, meaning that they are larger in area compared to D flip-flops. In particular, the PRESENT linear layer, which is essentially a bit permutation over the state, was decomposed as $P_2 \circ P_1^4$, where P_1 is a local permutation that operated on each 16-bit block of the 64-bit state and P_2 is some other global permutation. This decomposition allowed Jean et al. to implement the linear layer using only 25 scan flip-flops and 39 D flip-flops [JMPS17], whereas previous implementations have required all 64 flip-flops of the state to have additional multiplexer at their input ports [RPLP08].

5.2 Contributions

The contributions of this chapter can be summarized in the following salient points:

1. The main idea behind [JMPS17] was that the fewer scan flip-flops one uses to construct the circuit is likely to translate into a lowering of the total hardware area of the circuit. Taking this idea forward, in this chapter, we try to answer the following question:

   Question. *Is it possible to construct a 64-bit pipeline with only 2 additional MUXes such that it can execute the linear layer of PRESENT and GIFT?*

   The answer is yes, and we can easily extend it to any given permutation through the use of classical permutation theory [Con13]. However, even after applying various optimizations to the preliminary ideas from the permutation theory, the amount of time required to implement a PRESENT and GIFT round function takes at least thousands of clock cycles, meaning that it is still very slow. As latency is also an important lightweight metric, we explore a second direction in which we try to decrease latency by minimally increasing the number scan flip-flops.

2. Naturally, we then investigate if adding more scan flip-flops to the circuit can significantly reduce the number of clock cycles, possibly at the slight expense of area. Intuitively this makes sense because more scan flip-flops would allow us to execute more transposition operations on the state register in a single clock cycle, and hence it could reduce the total number of clock cycles to implement the complete bit permutation layer. Moreover, this could lead to a much smaller size of control bits required to control swaps and keep the area to a minimum. In fact, we found that adding 2 or 4 additional scan flip-flops provides us with a reasonable balance between area and throughput.

3. As a result of the theoretical foundations built in this chapter, we construct lightweight encryption-only and combined encryption+decryption implementations for a number of block ciphers, including AES, PRESENT, GIFT, SKINNY in Chapter 6. These implementations are not only the smallest of their kind (with respect

to ASIC area metric), but they also have the side benefit of achieving the optimum latency a serialized circuit can have.

5.3 Permutation Preliminaries

We recall the notation on permutations from Section 2.1. S_n denotes the permutation group over n elements. A k-cycle $\pi \in S_n$ (for $1 \leq k \leq n$) is generally expressed as the k-tuple $(i_1, i_2, \ldots, i_k)$ which implies

1. $\forall j \in [k-1], \quad \pi(i_j) = i_{(j+1) \bmod k}$,
2. $\forall i \in [n-1] \setminus \{i_1, \ldots, i_k\}, \quad \pi(i) = i$.

This is a permutation of order equal to k. A fixed point of the permutation can be considered as 1-cycle. A swap (or interchangeably a transposition) $\tau \in S_n$ is a 2-cycle. The permutation composition operation is denoted with $\circ$.

In our notation, we consider the following order of application among the permutations of a given composition. $f \circ g(x)$ corresponds to $f(g(x))$. In contrast, when we give a list of permutations, the application is assumed to start from the leftmost element, i.e. the list $\{g, f, h\}$ corresponds to $h \circ f \circ g$.

It is easy to see all disjoint cycles commute under the composition operation $\circ$. Furthermore, it is well known that every permutation in S_n can be expressed as a composition of disjoint k-cycles, uniquely up to ordering of the k-cycles. To begin discussions, we use a couple of results from Conrad [Con13].

Lemma 1. [Con13, Theorem 2.1] *For $n \geq 2$, S_n is generated by the set of its transpositions.*

Sketch Proof. First, the identity permutation (i.e. the neutral element) can be obtained by τ^2 where τ is an arbitrary transposition. As stated above, any permutation can be expressed as compositions of multiple cycles, and each of its k-cycle $(i_1, i_2, \ldots, i_k)$ can be constructed with the use of swaps $(i_1, i_2) \circ (i_2, i_3) \circ \cdots \circ (i_{k-1}, i_k)$ and so the result follows.

Lemma 2. [Con13, Theorem 2.5] *For $n \geq 2$, S_n is generated by the transposition $(0, 1)$ and the n-cycle $(0, 1, \ldots, n-1)$.*

Sketch Proof. A rigorous proof of the above lemma may be found in the work of Conrad [Con13], but for the benefit of the reader we give the sketch idea. First note that the set $G_1 = \{(0,1),\ (1,2),\ \cdots, (n-2, n-1)\}$ also generates S_n. That is because any arbitrary transposition (i, j) can be obtained by the composition $(i, i+1) \circ (i+1, j) \circ (i, i+1)$, where the first and third transpositions are already in G_1. If $|i+1-j| > 1$, then $(i+1, j)$

can be further written as $(i+1, i+2) \circ (i+2, j) \circ (i+1, i+2)$, and so on, until the term in the middle is in G_1. Given the following equality

$$\pi \circ (i_1, i_2, \ldots, i_k) \circ \pi^{-1} = (\pi(i_1), \pi(i_2), \ldots, \pi(i_k)),$$

for all k-cycles and $\pi \in S_n$, it is possible to show that any transposition of the form $(i, i+1)$ can be generated by $(0, 1)$ and the n-cycle $(0, 1, \ldots, n-1)$. Namely, if we denote $\sigma = (0, 1, \ldots, n-1)$, then we have

$$\sigma^i \circ (0, 1) \circ \sigma^{-i} = (\sigma^i(0), \sigma^i(1)) = (i, i+1)$$

This completes the proof. □

5.4 Single-swap Setting

5.4.1 Analysis of the Permutation Layer

The bit-permutation layer in PRESENT specifies that the i-th state bit is moved to the $P(i)$-th position after application of the permutation layer. This permutation is given in Table 2.2 in Chapter 2.

Let us look at the unique decomposition of P into its disjoint cycles. The disjoint decomposition of P consists in total of twenty 3-cycles, where the remaining four points are fixed. The 3-cycles are listed as follows:

- $(1, 16, 4)$, $(2, 32, 8)$, $(3, 48, 12)$, $(5, 17, 20)$, $(6, 33, 24)$,
- $(7, 49, 28)$, $(9, 18, 36)$, $(10, 34, 40)$, $(11, 50, 44)$, $(13, 19, 52)$,
- $(14, 35, 56)$, $(15, 51, 60)$, $(22, 37, 25)$, $(23, 53, 29)$, $(26, 38, 41)$,
- $(27, 54, 45)$, $(30, 39, 57)$, $(31, 55, 61)$, $(43, 58, 46)$, $(47, 59, 62)$.

For brevity, let the above 3-cycles be labeled by the symbols c_0 to c_{19}, in given order. Note that since all the c_i's are disjoint, the composition of all of them in any order will result in P. Each c_i may be further expressed as a composition of two swaps: $c_i = s_i \circ t_i$ (note that s_i and t_i do not commute). Table 5.1 lists all such decompositions in an explicit form.

Note that if we were to compose a permutation consisting of application of all the t_i's (in any order) followed by application of all the s_i's (again in any order), we would still obtain P. That is to say

$$P = s_{b_0} \circ s_{b_1} \circ \cdots \circ s_{b_{19}} \circ t_{a_0} \circ t_{a_1} \circ \cdots \circ t_{a_{19}}$$

Table 5.1 – Decomposition of the 3-cycle c_i's into swaps for the PRESENT permutation

i	c_i	$s_i \quad \circ \quad t_i$	i	c_i	$s_i \quad \circ \quad t_i$
0	$(1, 16, 4)$	$(4, 16) \circ (1, 4)$	10	$(14, 35, 56)$	$(14, 35) \circ (35, 56)$
1	$(2, 32, 8)$	$(8, 32) \circ (2, 8)$	11	$(15, 51, 60)$	$(15, 51) \circ (51, 60)$
2	$(3, 48, 12)$	$(12, 48) \circ (3, 12)$	12	$(22, 37, 25)$	$(25, 37) \circ (22, 25)$
3	$(5, 17, 20)$	$(5, 17) \circ (17, 20)$	13	$(23, 53, 29)$	$(29, 53) \circ (23, 29)$
4	$(6, 33, 24)$	$(24, 33) \circ (6, 24)$	14	$(26, 38, 41)$	$(26, 38) \circ (38, 41)$
5	$(7, 49, 28)$	$(28, 49) \circ (7, 28)$	15	$(27, 54, 45)$	$(45, 54) \circ (27, 45)$
6	$(9, 18, 36)$	$(9, 18) \circ (18, 36)$	16	$(30, 39, 57)$	$(30, 39) \circ (39, 57)$
7	$(10, 34, 40)$	$(10, 34) \circ (34, 40)$	17	$(31, 55, 61)$	$(31, 55) \circ (55, 61)$
8	$(11, 50, 44)$	$(44, 50) \circ (11, 44)$	18	$(43, 58, 46)$	$(46, 58) \circ (43, 46)$
9	$(13, 19, 52)$	$(13, 19) \circ (19, 52)$	19	$(47, 59, 62)$	$(47, 59) \circ (59, 62)$

where $a_0, a_1, \ldots, a_{19}$ and $b_0, b_1, \ldots, b_{19}$ are arbitrary ordering of the set $\{0, 1, \ldots, 19\}$. We will prove a generalized form of the above statement in the following lemma.

Lemma 3. *Let $u_1, u_2, \ldots, u_{2k+1}$ denote a series of arbitrary permutations from S_n with $\mathbb{A}_1, \mathbb{A}_2, \ldots, \mathbb{A}_{2k+1}$ denoting their activity sets, respectively. Suppose that each pair of u_{2i}, u_{2j-1} are pair-wise disjoint permutations for $1 \leq i \leq k$ and $1 \leq j \leq k+1$. Let $\pi = u_1 \circ u_2 \circ \cdots \circ u_{2k+1}$ and $\theta = u_2 \circ u_4 \circ \cdots \circ u_{2k}$. Let $\mathbb{A}_{\mathsf{even}} = \cup_{i=1}^{k} \mathbb{A}_{2i}$. For every $x \in A_{\mathsf{even}}$, it holds that $\Theta(x) = \Pi(x)$.*

Proof. The proof follows the idea of applying each permutation u_i starting from the right-side and going towards left in the descriptions of π and θ. Namely, for every $x \in \mathbb{A}_{\mathsf{even}}$, we can trace the computation of $\Theta(x)$ and $\Pi(x)$ separately. Let $\mathbb{A}_{\mathsf{odd}} = \cup_{j=1}^{k+1} \mathbb{A}_{2j-1}$. Given that even, odd indexed permutations are pair-wise disjoint, then $\mathbb{A}_{\mathsf{even}} \cap \mathbb{A}_{\mathsf{odd}} = \emptyset$.

We can start by showing that $u_{2k}(x) = u_{2k} \circ u_{2k+1}(x)$. Here, the condition $x \in \mathbb{A}_{\mathsf{even}}$ naturally implies that $x \notin \mathbb{A}_{2k+1}$, therefore $u_{2k+1}(x) = x$. If we let $x_k = u_{2k}(x)$, then it also holds that $x_k \in \mathbb{A}_{\mathsf{even}}$.

Then we can continue by applying recursion from $j = k$ and conclude at $j = 1$. In the similar fashion, we define $x_j = u_{2j}(x_{j+1})$ and it follows that:

- $x_{j+1} \in \mathbb{A}_{\mathsf{even}} \implies x_{j+1} \notin \mathbb{A}_{2j+1} \implies x_j = u_{2j} \circ u_{2j+1}(x_{j+1})$,
- $x_{j+1} \in \mathbb{A}_{\mathsf{even}} \implies x_j \in \mathbb{A}_{\mathsf{even}}$.

At the final step, we can also deduce that $u_1(x_1) = x_1 = \pi(x) = \theta(x)$, as $x_1 \notin \mathbb{A}_1$. □

Lemma 4. *Let π be a permutation in S_n whose disjoint cycle decomposition consists of the cycles $c_0, c_1, \ldots, c_{m-1}$ each with orders $i_0, i_1 \ldots, i_{m-1}$ respectively such that $i_0 \leq i_1 \leq$*

$\cdots \leq i_{m-1}$, *i.e.*

$$\pi = c_0 \circ c_1 \circ \cdots \circ c_{m-1}$$

Let each c_j be expressed as a composition of $i_j - 1$ swaps $s_j(1), s_j(2), \ldots, s_j(i_j - 1)$, in a manner such that any pair of $s_j(a)$, $s_{j'}(b)$ are also disjoint for all choices of a, b and $j \neq j'$.

$$\begin{array}{rcccccccl}
s_{m-1}(i_{m-1}-1) \circ & \cdots & \circ & \cdots & \circ \cdots \circ & s_{m-1}(2) & \circ\ s_{m-1}(1) & = & c_{m-1} \\
 & & & \vdots & & & & & \\
 & s_j(i_j-1) \circ & \cdots & & \circ \cdots \circ & s_j(2) & \circ\ s_j(1) & = & c_j \\
 & & & \vdots & & & & & \\
 & & & s_0(i_0-1) \circ & \cdots \circ & s_0(2) & \circ\ s_0(1) & = & c_0 \\
\hline
\textit{Sets:} & \chi_{i_{m-1}-1} & \chi_{i_j-1} & \chi_{i_0-1} & \cdots & \chi_2 & \chi_1 & &
\end{array}$$

Define the set $\chi_k = \{s_{m-1}(k), s_{m-2}(k), \ldots\}$ (for $1 \leq k < i_{m-1}$) as explained above. Let θ_k be the composition of all transpositions in χ_k in any arbitrary order. Then we must have :

1. *Each θ_k is invariant of the order in which the swaps in χ_k are applied,*
2. *We must have $\pi = \theta_{i_{m-1}-1} \circ \cdots \theta_{i_j-1} \circ \cdots \theta_2 \circ \theta_1$.*

Proof. The proof follows as an application of the Lemma 4, as it allows us to interleave decomposed swaps of c_j with other disjoint swaps from other c_i ($i \neq j$) permutations.

Namely, we can first define the activity sets $\mathbb{A}_0, \mathbb{A}_1, \ldots, \mathbb{A}_{m-1}$ for each of the cycles $c_0, c_1, \ldots, c_{m-1}$. Since, they are pair-wise disjoint, it holds that $A_i \cap A_j = \emptyset$ for every $i \neq j$. We can further define $\mathbb{A}_m = [1, n] \setminus \cup_{i=0}^{m-1} \mathbb{A}_i$.

Let $\theta = \theta_{i_{m-1}-1} \circ \cdots \theta_{i_j-1} \circ \cdots \theta_2 \circ \theta_1$. Suppose that we first fix a value for i such that $1 \leq i \leq m-1$. For each $x \in \mathbb{A}_i$, we can show that $\theta(x) = \pi(x)$ with the help of Lemma 4. We simply need to find a series of disjoint permutations u_j. Here, we construct the even-indexed permutations as $s_i(1), s_i(2), \ldots, s_i(i_j - 1)$ in reverse order, and the odd-indexed permutations are defined as the rest of the interleaving permutations. Then, it is clear that $\theta(x) = \pi(x)$ for all $x \in \mathbb{A}_i$.

Exception to the above statement is the case $x \in \mathbb{A}_m$. Then, $\pi(x) = x = \theta(x)$. This concludes the proof. □

Corollary 1. *The PRESENT permutation P satisfies that*

$$P = s_{b_0} \circ s_{b_1} \circ \cdots \circ s_{b_{19}} \circ t_{a_0} \circ t_{a_1} \circ \cdots \circ t_{a_{19}}$$

for the swaps s_i, t_i given in Table 5.1.

5.4.2 Pipeline with Swap $(1, 0)$

Lemma 2 already states that any permutation in S_n can be generated by the cycles $(0, 1, \ldots, n-1)$ and $(0, 1)$. In a typical serial implementation, the cycle $(0, 1, \ldots, n-1)$ naturally appears as the rotation operation of the pipeline constructed from n flip-flops. The swap $(0, 1)$ can be realized by simply replacing two of these flip-flops with scan flip-flops. Therefore, Lemma 2 implies the existence of PRESENT permutation realization with only 2 scan flip-flops. Therefore, we explore the number of necessary clock cycles required for executing PRESENT permutation, i.e. by deriving the decomposition sequence with a straightforward application of the above formalism.

We claim that S_{64} is generated by the cycles $w = (1, 0)$ and $r = (63, 62, \ldots, 0)$. The idea is to implement all the transpositions t_i followed by all the s_i's. In order to do so, let us first see how any arbitrary transposition can be implemented only using r and w.

Implementing $(x, x-1)$. From Lemma 1, it suffices to find a permutation π such that $(\pi(1), \pi(0)) = (x, x-1)$. To be more precise, we look for a positive integer a that satisfies $\pi = r^a$. It is easy to note that $r^a(x) = (64 - a + x) \bmod 64$. Therefore $a = 1 - x$ is a proper solution. Therefore, from Lemma 1, it follows that $(x, x-1) = r^{1-x} \circ w \circ r^{-(1-x)}$.

Implementing a transposition (x, y). We need to find a swap sequence for a tuple (x, y) such that $x > y$ and $x, y \in [0, 63]$. Following the results from the Lemmas 1 and 2, we can deduce the following sequence:

$$
\begin{aligned}
(x, y) &= (x, x-1) \circ (x-1, y) \circ (x, x-1) \\
&= (x, x-1) \circ (x-1, x-2) \circ (x-2, y) \circ (x-1, x-2) \circ (x, x-1) \\
&= (x, x-1) \circ (x-1, x-2) \circ \cdots \circ (y+1, y) \circ \cdots \circ (x-1, x-2) \circ (x, x-1) \\
&= (r^{1-x} \circ w \circ r^{x-1}) \circ (r^{2-x} \circ w \circ r^{x-2}) \circ \cdots \circ (r^{-y} \circ w \circ r^{y}) \circ \cdots \circ \\
&\quad (r^{2-x} \circ w \circ r^{x-2}) \circ (r^{1-x} \circ w \circ r^{x-1}) \\
&= r^{1-x} \circ w \circ (r \circ w)^{x-y-1} \circ (r^{-1} \circ w)^{x-y-1} \circ r^{x-1} \\
&= r^{(1-x) \bmod 64} \circ w \circ (r \circ w)^{x-y-1} \circ (r^{63} \circ w)^{x-y-1} \circ r^{(x-1) \bmod 64}
\end{aligned}
$$

Given the decomposition of (x, y) in terms of r and w as given above, the next question naturally arises is how to implement it using 2 scan flip-flops (or by adding 2 MUXes on top of regular flip-flops). Consider the circuit in Figure 5.1. It consists of an array of 64 flip-flops, with the two additional MUXes placed before the leftmost two flip-flops, and they are both controlled by the Sel signal. When Sel is logic 0, the data in the flip-flops simply rotate bit-wise towards the left. When Sel is logic 1, the b_0 bit stored in FF_0 is held in place, the bit b_1 is forwarded to FF_{63} and all the other remaining 62 bits are rotated left by 1 position. Implementing a particular permutation $\pi \in S_{64}$ on this circuit, essentially boils down to the following question.

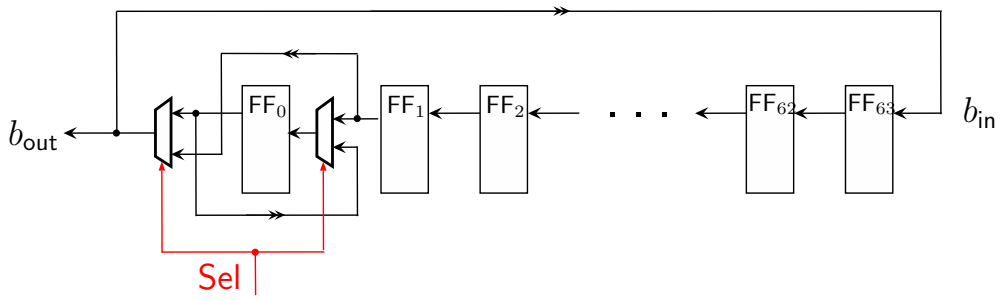

Figure 5.1 – Shift register circuit with (subsequent) 2 scan flip-flops

Question. *Suppose that* $\mathsf{FF}_i(t)$ *denotes the value of the bit stored in flip-flop* FF_i *at* t*-th clock cycle, for* $t \geq 0$ *and* $i \in [0, 63]$*. Does there exist some sequence of* Sel *signals* $s_0, s_1, \ldots, s_{T-1}$ *such that, after* T *clock cycles the values stored in the pipeline become* $\mathsf{FF}_{\pi(i)}(T) = \mathsf{FF}_i(0)$*?*

Lemma 5. *Considering the circuit in Figure 5.1, implementing an arbitrary swap operation* (x, y) *with* $x > y$ *requires at most* $64(x - y)$ *clock cycles.*

Proof. Again, note that $w = (1, 0)$ and $r = (63, 62, \ldots, 0)$. In Figure 5.1, setting the select signal Sel to logic 0 causes the shift register to implement the r function, as data follows the simple rotation path. Setting Sel to logic 1 brings about the following transformation:

$$(b_0, b_1, b_2, \ldots, b_{63}) \rightarrow (b_0, b_2, b_3, \ldots, b_{63}, b_1)$$

This is the same as applying the function $(r \circ w)$. As the Lemma 5 suggests, a sufficiently long sequence of r and $(r \circ w)$ can perform any permutation from S_{64}. Thus, by controlling the Sel signal, we can make the shift register circuit alternate between r and $v = (r \circ w)$ functions. Therefore, the following sequence can be used for setting the Sel signal in order to execute an arbitrary swap (x, y):

$$0^{(-x) \bmod 64} || 1 || 1^{x-y-1} || (0^{62} || 1)^{x-y-1} || 0^{(x-1) \bmod 64}$$

□

Note that the circuit must execute the Sel signal starting from the rightmost bit.

Corollary 2. *Employing the shift register circuit in Figure 5.1, one round of the* PRESENT *bit permutation can be executed in 36480 clock cycles.*

Proof. The idea is to execute the PRESENT permutation P by executing each of the transpositions t_i and then s_i sequentially. Denoting $t_i = (x_i, y_i)$ and $s_i = (x_{20+i}, y_{20+i})$ for

$i \in [0, 19]$, (with $x_i > y_i$) the number of clock cycles can be calculated as $\sum_{i=0}^{39} 64(x_i - y_i) = 36480$. □

This result is a pessimistic one as it shows that executing one round of PRESENT in the form of 1-bit serial pipeline circuit (with 2 scan flip-flops given in Figure 5.1) leads to heavy loss of throughput. In the following subsections, we will show how this number can be reduced further.

5.4.3 Pipeline with Swap $(\kappa, 0)$

Before we outline the method used to reduce the number of operations, let us look at the following definition.

Definition 1. *As in Lemma 5, let π be a permutation in S_n whose disjoint cycle decomposition consists of the cycles $c_0, c_1, \ldots, c_{m-1}$ each with orders $i_0, i_1 \ldots, i_{m-1}$ respectively. Let each c_j be expressed as composition of $i_j - 1$ transpositions $s_j(1), s_j(2), \ldots, s_j(i_j - 1)$. Denote the transposition $s_j(k) = (x_j(k), y_j(k))$ with $x_j(k) > y_j(k)$. π is said to be a special permutation of the type κ, if κ is the largest integer for which the following holds:*

$$x_j(k) - y_j(k) \equiv 0 \bmod \kappa, \quad \forall\ j \in [0, m-1], \forall\ k \in [0, i_j - 1]$$

It is easy to see from Table 5.1, that the PRESENT permutation P is a special permutation of type 3. Before we proceed, let us look at a result concerning special permutations of type κ.

Lemma 6. *Let G_κ denote the set of all the special permutations of S_{64} of type κ. Then G_κ can be generated by the permutations $w_\kappa = (\kappa, 0)$ and $r = (63, 62, \ldots, 0)$.*

Proof. The only thing we need to show is that any transposition (x, y) with $x > y$ and $x \equiv y \bmod \kappa$, can be generated using w_κ and r. For brevity, let $z = \frac{x-y}{\kappa}$. Then the following sequence can be obtained in the same manner as before:

$$\begin{aligned}
(x, y) &= (x, x-\kappa) \circ (x-\kappa, y) \circ (x, x-\kappa) \\
&= (x, x-\kappa) \circ (x-\kappa, x-2\kappa) \circ (x-2\kappa, y) \circ (x-\kappa, x-2\kappa) \circ (x, x-\kappa) \\
&= (x, x-\kappa) \circ (x-\kappa, x-2\kappa) \circ \cdots \circ (y+\kappa, y) \circ \cdots \circ (x-\kappa, x-2\kappa) \circ (x, x-\kappa) \\
&= (r^{\kappa-x} \circ w_\kappa \circ r^{x-\kappa}) \circ (r^{2\kappa-x} \circ w_\kappa \circ r^{x-2\kappa}) \circ \cdots \circ (r^{-y} \circ w_\kappa \circ r^{y}) \circ \cdots \circ \\
&\quad (r^{2\kappa-x} \circ w_\kappa \circ r^{x-2\kappa}) \circ (r^{\kappa-x} \circ w_\kappa \circ r^{x-\kappa}) \\
&= r^{\kappa-x} \circ w_\kappa \circ (r^{\kappa} \circ w_\kappa)^{z-1} \circ (r^{-\kappa} \circ w_\kappa)^{z-1} \circ r^{x-\kappa} \\
&= r^{(\kappa-x) \bmod 64} \circ w_\kappa \circ (r^{\kappa} \circ w_\kappa)^{z-1} \circ (r^{64-\kappa} \circ w_\kappa)^{z-1} \circ r^{(x-\kappa) \bmod 64}
\end{aligned}$$

□

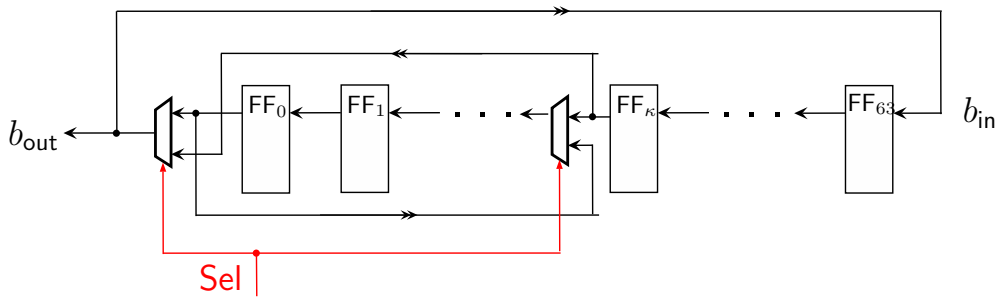

Figure 5.2 – Shift register circuit with $(\kappa, 0)$ swap

Naturally, the next inquisitive step is to see how much reduction we can get in number of clock cycles by implementing type 3 swap operation on circuit.

Corollary 3. *The circuit in Figure 5.2 can execute an arbitrary swap operation* (x, y) *with* $x > y$ *and* $x \equiv y \bmod \kappa$ *in* $\frac{64(x-y)}{\kappa}$ *clock cycles.*

Proof. As before, setting Sel to 0 executes the rotate function r. Setting Sel to 1, achieves the following transformation:

$$(b_0, b_1, b_2, \ldots, b_{62}, b_{63}) \rightarrow (b_1, b_2, \ldots b_{\kappa-1}, b_0, b_{\kappa+1}, \ldots, b_{62}, b_{63}, b_\kappa)$$

This is same as applying the transformation $v_\kappa = r \circ w_\kappa$. Then we can equally express the permutation σ with the following formulation:

$$\sigma = r^{(\kappa-x-1) \bmod 64} \circ v_\kappa \circ (r^{\kappa-1}1)^{z-1} \circ (0^{63-\kappa} \circ v_\kappa)^{z-1} \circ r^{(x-\kappa) \bmod 64}$$

Thus, as before, controlling Sel makes the circuit alternate between r and v_κ operations. We can then execute (x, y) (with $x \equiv y \pmod 3$) with the following binary sequence:

$$0^{(\kappa-x-1) \bmod 64}||1||(0^{\kappa-1}1)^{z-1}||(0^{63-\kappa}||1)^{z-1}||0^{(x-\kappa) \bmod 64}$$

□

Corollary 4. *The circuit in Figure 5.2 can execute the bit permutation* P *of* PRESENT *in 12160 clock cycles.*

Proof. We have already noted that P is a special permutation of type 3. As in the previous corollary, let $t_i = (x_i, y_i)$ and $s_i = (x_{20+i}, y_{20+i})$ for $i \in [0, 19]$, (with $x_i > y_i$). For performing all the t_i's followed by all the s_i's sequentially, the number of clock cycles can be calculated as $\sum_{i=0}^{39} 64 \cdot \frac{(x_i-y_i)}{3} = 12160$. □

By using the modified shift register structure, we obtain a threefold increase of throughput in computation of the PRESENT permutation. However, this is still way too slow, and in

the subsequent sections, we will try to find if the speed of computations can be further increased.

5.4.4 Control Bit Concatenation

So far, we were executing each transposition operation sequentially, i.e. one after the other. However in the interest of speeding up computations, let us investigate if it is at all possible to execute some of the swap operations concurrently.

Definition 2. *Given a swap (transposition) $\sigma = (x, y)$ in S_{64} with $x > y$, the selection vector $\overrightarrow{\mathsf{Sel}}_\sigma$ is defined as the binary vector that follows from the Corollary 4, for its execution by the circuit in Figure 5.2.*

First note that the length of $\overrightarrow{\mathsf{Sel}}_\sigma$ is exactly $\frac{64(x-y)}{\kappa}$. For example, let $\kappa = 3$, as in PRESENT. Consider $\sigma = (12, 3)$, for which $z = 3$. In this case, we would have (following from the proof of Corollary 3):

$$\begin{aligned}
\sigma &= r^{(\kappa-x) \bmod 64} \circ w_\kappa \circ (r^\kappa \circ w_\kappa)^{z-1} \circ (r^{64-\kappa} \circ w_\kappa)^{z-1} \circ r^{(x-\kappa) \bmod 64} \\
\sigma &= r^{(\kappa-x-1) \bmod 64} \circ v_\kappa \circ (r^{\kappa-1} \circ v_\kappa)^{z-1} \circ (r^{63-\kappa} \circ v_\kappa)^{z-1} \circ r^{(x-\kappa) \bmod 64} \\
\overrightarrow{\mathsf{Sel}}_\sigma &= 0^{54}||1||0^2 10^2 1||0^{60}10^{60}1||0^9
\end{aligned}$$

Let us now re-write the permutations r and v_κ in a functional form:

$$r(\alpha) = (\alpha - 1) \bmod 64, \quad v_\kappa(\alpha) = \begin{cases} \kappa - 1, & \text{if } \alpha = 0, \\ 63, & \text{if } \alpha = \kappa, \\ (\alpha - 1) \bmod 64, & \text{otherwise.} \end{cases}$$

We can see that r and v_κ differ on only two inputs 0 and κ. By stretching our selection vector notation, let $\overrightarrow{\mathsf{Sel}}_\rho$ also denote an arbitrary 64-bit binary vector that implements the permutation ρ when fed to the Sel port of the circuit in Figure 5.2 over 64 consecutive clock cycles. Let $\mathbb{B}_\rho$ be the set of elements that denote the positions of 1's in $\overrightarrow{\mathsf{Sel}}_\rho$. The bits are indexed in a manner that the rightmost bit corresponds to the lowest index, i.e. $\overrightarrow{\mathsf{Sel}}_\sigma = \mathsf{Sel}_{63}||\ldots||\mathsf{Sel}_1||\mathsf{Sel}_0$.

From the functional equations of r and v_κ, we can deduce that ρ's set of active elements $\mathbb{A}_\rho = \mathbb{U}_\rho \cup \mathbb{V}_\rho$, where

$$\mathbb{U}_\rho = \{\alpha : \alpha \in \mathbb{B}_\rho\}, \ \mathbb{V}_\rho = \{\alpha + \kappa \bmod 64 : \alpha \in \mathbb{B}_\rho\}$$

It is also possible to deduce ρ from $\mathbb{B}_\rho$. If $\mathbb{B}_\rho$ contains a subset of elements $\{b, b+\kappa, b+2\kappa, \ldots, b+(\ell-1)\kappa\}$ which are in an arithmetic sequence with common difference κ, then we have

$$\rho(b + i\kappa) = b + (i-1)\kappa, \ \forall i \in [1, \ell], \text{ and } \ \rho(b) = b + \ell\kappa$$

For all other elements $\hat{b} \in \mathbb{B}_p$ that are not part of any arithmetic sequence with common difference κ, we have $\rho(\hat{b}) = \hat{b} + \kappa$ and $\rho(\hat{b} + \kappa) = \hat{b}$. For all $x \in [0, n-1] \setminus \mathbb{A}_\rho$, we have $\rho(x) = x$.

Example 1. Suppose that $\mathbb{B}_p = \{6, 9, 19, 29, 53, 56, 60, 61\}$ with $\kappa = 3$. The subsets of arithmetic sequences with $\kappa = 3$ difference are 6, 9 and 53, 56. Hence, we have $\mathbb{A}_p = \{6, 9, 12, 19, 22, 29, 32, 53, 56, 59, 60, 61, 63, 0\}$. We have $\rho = (12, 9, 6) \circ (59, 56, 53) \circ (22, 19) \circ (32, 29) \circ (63, 60) \circ (61, 0)$.

Suppose that we chop $\overrightarrow{\mathsf{Sel}}_\sigma$ into 64-bit blocks. Let π_i (for $i = 0$ to $z-1$) be the composition of all the permutations in the i-th 64-bit block. Let us use the notation

$$\overrightarrow{\mathsf{Sel}}_\sigma = \overrightarrow{\mathsf{Sel}}_{\pi_{z-1}} || \overrightarrow{\mathsf{Sel}}_{\pi_{z-2}} || \cdots || \overrightarrow{\mathsf{Sel}}_{\pi_2} || \overrightarrow{\mathsf{Sel}}_{\pi_1} || \overrightarrow{\mathsf{Sel}}_{\pi_0}$$

Naturally, we have $\sigma = \pi_{z-1} \circ \pi_{z-2} \circ \cdots \circ \pi_2 \circ \pi_1 \circ \pi_0$. In the above example, for $\sigma = (12, 3)$ the selection vector can be partitioned into three 64-bit blocks:

$$\overrightarrow{\mathsf{Sel}}_\sigma = 0^{54}10^210^210^3 \quad || \quad 0^{57}10^6 \quad || \quad 0^{54}10^9$$

thus, we have $\mathbb{B}_{\pi_0} = \{9\}$, $\mathbb{B}_{\pi_1} = \{6\}$, $\mathbb{B}_{\pi_2} = \{3, 6, 9\}$.

If we generalize the above formulation for an arbitrary swap (x, y) such that $x > y$ and $x \equiv y \pmod{\kappa}$, then we would always obtain a series of permutations $\pi_0, \pi_1, \ldots, \pi_{z-1}$ with their sets $\mathbb{B}_0, \mathbb{B}_1, \ldots, \mathbb{B}_{z-1}$ respectively, where $z = \frac{x-y}{\kappa}$. Furthermore, it holds that $B_{z-1} \supseteq B_i$ for all $i \in [0, z-2]$. In particular, $\mathbb{B}_{\pi_2} = \{y, y + \kappa, \ldots, x - \kappa\}$, where all elements have the same residue modulo κ. From the analysis presented above, it can be deduced that for all i,

$$\pi_i(\alpha) = \alpha, \ \forall \alpha \ \not\equiv x \bmod \kappa.$$

This is because the 1's (equivalently v_κ's) in this block appear at distances of κ. If we apply each function in π_i one by one, for any input $\alpha \ \not\equiv x \bmod \kappa$, the corresponding input to v_κ is never 0 or κ, and so a plain rotation is effectively executed. Therefore all the π_i's perform shuffling on only a subset of elements that are congruent to $x \bmod \kappa$ and leave the others untouched. From the equation $\mathbb{A}_\rho = \mathbb{U}_\rho \cup \mathbb{V}_\rho$, we can also deduce that $\mathbb{A}_{\pi_{z-1}} = \{y, y + \kappa, \ldots, x - \kappa, x\}$. Thus each π_i is effectively a permutation function on only a subset of $\{0, 1, 2, 3, \ldots, 63\}$ that are congruent to $x \bmod \kappa$.

Lemma 7. *Let $\overrightarrow{\mathsf{Sel}}_{\rho_1}$ and $\overrightarrow{\mathsf{Sel}}_{\rho_2}$ be two 64 bit signal vectors implementing permutations ρ_1 and ρ_2 on the circuit of Figure 5.2. If $\mathbb{A}_{\rho_1} \cap \mathbb{A}_{\rho_2} = \emptyset$, then $\rho = \rho_1 \circ \rho_2$ can be concurrently executed on this circuit using the signal vector $\overrightarrow{\mathsf{Sel}}_{\rho_1} \hat{|} \overrightarrow{\mathsf{Sel}}_{\rho_2}$, where $\hat{|}$ denotes a bit-wise OR operation on the vectors.*

Proof. First, ρ_1 and ρ_2 are disjoint, as $\mathbb{A}_{\rho_1} \cap \mathbb{A}_{\rho_2} = \emptyset$. Note that this implies $\mathbb{B}_{\rho_1} \cap \mathbb{B}_{\rho_2} = \emptyset$. This means that the 1's in the $\overrightarrow{\mathsf{Sel}}_{\rho_1}$ and $\overrightarrow{\mathsf{Sel}}_{\rho_2}$ vectors are not aligned. This is equivalent to saying that $\overrightarrow{\mathsf{Sel}}_{\rho_1} \hat{|} \overrightarrow{\mathsf{Sel}}_{\rho_2}$ has 1's in all the locations in which either $\overrightarrow{\mathsf{Sel}}_{\rho_1}$ or $\overrightarrow{\mathsf{Sel}}_{\rho_2}$ has

1. Let $\overrightarrow{\mathsf{Sel}}_\rho = \overrightarrow{\mathsf{Sel}}_{\rho_1} \hat{|} \overrightarrow{\mathsf{Sel}}_{\rho_2}$. We already know that $\mathbb{B}_\rho$ would contain all elements of $\mathbb{B}_{\rho_1}$ and $\mathbb{B}_{\rho_2}$. Thus the arithmetic sequence structures of both $\mathbb{B}_{\rho_1}$ and $\mathbb{B}_{\rho_2}$ are preserved in $\mathbb{B}_\rho$. Furthermore, $\mathbb{A}_{\rho_1} \cap \mathbb{A}_{\rho_2} = \emptyset$ ensures that no new arithmetic sequence of common difference κ is created $\mathbb{B}_\rho$ that is not already present in $\mathbb{B}_{\rho_1}$ or $\mathbb{B}_{\rho_2}$.

Let us prove the latter part by contradiction. Suppose that $\exists b_1 \in \mathbb{B}_{\rho_1},\ b_2 \in \mathbb{B}_{\rho_2}$ such that $b_2 = b_1 + \kappa$. Then by definition $b_1, b_1 + \kappa \in \mathbb{A}_{\rho_1}$ and $b_2, b_2 + \kappa \in \mathbb{A}_{\rho_2}$. However, $b_1 + \kappa = b_2$, and so this contradicts the fact that $\mathbb{A}_{\rho_1} \cap \mathbb{A}_{\rho_2} = \emptyset$. Since the arithmetic structures are preserved, ρ essentially executes ρ_1 and ρ_2 concurrently: we have $\forall \alpha \in \mathbb{A}_{\rho_1}$, $\rho(\alpha) = \rho_1(\alpha)$ and $\forall \alpha \in \mathbb{A}_{\rho_2}$, $\rho(\alpha) = \rho_2(\alpha)$. Also $\rho(\alpha) = \alpha$ for all $\alpha \notin \mathbb{A}_{\rho_1} \cup \mathbb{A}_{\rho_2}$. Thus we have $\rho = \rho_1 \circ \rho_2$. □

Lemma 8. *Let $\sigma_1 = (x_1, y_1)$ and $\sigma_2 = (x_2, y_2)$ be two transpositions in S_{64} of type κ satisfying $x_i > y_i$ for $i = 1, 2$. Without loss of generality, let $\ell_1 = (x_1 - y_1) \geq (x_2 - y_2) = \ell_2$, and $z_i = \frac{\ell_i}{\kappa}$. Let the respective decompositions of these two permutations be denoted by the symbols π_i and θ_i:*

$$\sigma_1 = \pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_2 \circ \pi_1 \circ \pi_0$$
$$\sigma_2 = \theta_{z_1-1} \circ \theta_{z_2-2} \circ \cdots \circ \theta_2 \circ \theta_1 \circ \theta_0$$

In order to ensure that $\overrightarrow{\mathsf{Sel}}_{\sigma_1}$ and $\overrightarrow{\mathsf{Sel}}_{\sigma_2}$ are of the same length, we use the identity permutation as the padding $\theta_{z_1-1} = \theta_{z_1-2} = \cdots = \theta_{z_2} = r^{64}$. If $\mathbb{A}_{\pi_{z_1-1}} \cap \mathbb{A}_{\theta_{z_2-1}} = \emptyset$, then it is possible to execute σ_1 and σ_2 concurrently on the circuit in Figure 5.2 and achieve $\sigma_1 \circ \sigma_2$ in $64 \cdot z_1$ clock cycles. The sequence $\overrightarrow{\mathsf{Sel}}_{\sigma_1 \circ \sigma_2} = \overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|} \overrightarrow{\mathsf{Sel}}_{\sigma_2}$ realizes this permutation.

Proof. Let $\sigma = \sigma_1 \circ \sigma_2$. Since $\mathbb{A}_{\pi_{z_1-1}} \cap \mathbb{A}_{\theta_{z_2-1}} = \emptyset$, from the result of Lemma 7, we can certainly use $\overrightarrow{\mathsf{Sel}}_{\pi_0} \hat{|} \overrightarrow{\mathsf{Sel}}_{\theta_0}$ to get $\pi_0 \circ \theta_0$. Since all $\mathbb{A}_{\pi_i}$'s and $\mathbb{A}_{\theta_i}$'s are subsets of $\mathbb{A}_{\pi_{z_1-1}}$ and $\mathbb{A}_{\theta_{z_2-1}}$ respectively, we also have $\mathbb{A}_{\pi_i} \cap \mathbb{A}_{\theta_i} = \emptyset$ for all $0 \leq i \leq z_1 - 1$. We can then use $\overrightarrow{\mathsf{Sel}}_{\pi_i} \hat{|} \overrightarrow{\mathsf{Sel}}_{\theta_i}$ to get $\pi_i \circ \theta_i$ for all $0 \leq i \leq z_1 - 1$. Thus if $\overrightarrow{\mathsf{Sel}}_\rho = \overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|} \overrightarrow{\mathsf{Sel}}_{\sigma_2}$, we naturally have

$$\rho = (\pi_{z_1-1} \circ \theta_{z_1-1}) \circ (\pi_{z_1-2} \circ \theta_{z_1-2}) \circ \cdots \circ (\pi_1 \circ \theta_1) \circ (\pi_0 \circ \theta_0)$$

In order to show that $\sigma = \rho$, we can apply Lemma 3 twice. We recall that this lemma allows us to reorder the decomposition on the condition that they are pair-wise disjoint. We can individually look at the sets of elements $x \in \mathbb{A}_{\pi_{z_1-1}}$ and $x \in \mathbb{A}_{\theta_{z_2-1}}$ and show that $\rho(x) = \sigma(x)$ in these two disjoint sets. The remaining elements are trivially satisfied as they are fixed points of $\rho(x) = \sigma(x) = x$. □

The above result may be extended to a set of any number of special swaps σ_i ($i = 1$ to k) of the type κ, provided that the respective $\mathbb{A}_{\pi_{z-1}}$ sets are pair-wise disjoint. In that case

Table 5.2 – Concurrent execution of the t_i and s_i's in the PRESENT permutation

Group	mod3	t_i	$\max(x_i - y_i)$	# of cycles
1	0	$(57, 39),\ (36, 18),\ (12, 3)$	33	704
	1	$(61, 55),\ (52, 19),\ (4, 1)$		
	2	$(62, 59),\ (44, 11),\ (8, 2)$		
2	0	$(60, 51),\ (45, 27),\ (24, 6)$	21	448
	1	$(46, 43),\ (40, 34),\ (28, 7)$		
	2	$(56, 35),\ (29, 23),\ (20, 17)$		
3	1	$(25, 22)$	3	64
	2	$(41, 38)$		

Group	mod3	s_i	$\max(x_i - y_i)$	# of cycles
1	0	$(51, 15)$	36	768
	1	$(55, 31),\ (19, 13)$		
	2	$(53, 29),\ (17, 5)$		
2	0	$(48, 12)$	36	768
	1	$(58, 46),\ (34, 10)$		
	2	$(59, 47),\ (32, 8)$		
3	0	$(54, 45),\ (39, 30),\ (18, 9)$	21	448
	1	$(49, 28),\ (16, 4)$		
	2	$(50, 44),\ (35, 14)$		
4	0	$(33, 24)$	12	256
	1	$(37, 25)$		
	2	$(38, 26)$		

we have

$$\overrightarrow{\mathsf{Sel}}_{\sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_k} = \overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|} \ \overrightarrow{\mathsf{Sel}}_{\sigma_2} \hat{|} \ \cdots \hat{|} \ \overrightarrow{\mathsf{Sel}}_{\sigma_k}$$

Corollary 5. *Let $\sigma_1 = (x_1, y_1)$ and $\sigma_2 = (x_2, y_2)$ be two transpositions ($x_i > y_i$, $i = 1, 2$) in S_{64} such that $x_1 - y_1 \equiv x_2 - y_2 \bmod \kappa$, and $x_1 \not\equiv x_2 \bmod \kappa$. Without loss of generality let $\ell_1 = (x_1 - y_1) \geq (x_2 - y_2) = \ell_2$, and $z_i = \frac{\ell_i}{\kappa}$. As before, let the respective decompositions are denoted by the symbols π_i and θ_i and append $64(z_1 - z_2)$ zeroes to $\overrightarrow{\mathsf{Sel}}_{\sigma_2}$ to make the two $\overrightarrow{\mathsf{Sel}}$ vectors of the same length. It is possible to execute σ_1 and σ_2 concurrently on the circuit in Figure 5.2 and achieve $\sigma_1 \circ \sigma_2$ in $64 \cdot z_1$ clock cycles by using $\overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|} \ \overrightarrow{\mathsf{Sel}}_{\sigma_2}$ as the select signal vector.*

Proof. We have already seen that for any transposition $\sigma = (x, y) = \pi_{z-1} \circ \cdots \circ \pi_0$, we have $\mathbb{A}_{\pi_{z-1}} = \{x, x - \kappa, x - 2\kappa, \ldots, y\}$. Thus $\mathbb{A}_{\pi_{z-1}}$ contains elements that are only congruent to $x \bmod \kappa$. As (x_1, y_1) and (x_2, y_2) belong to different equivalence classes modulo κ, $\mathbb{A}_{\pi_{z_1-1}} \cap \mathbb{A}_{\theta_{z_2-1}} = \emptyset$. Thus the result follows. □

Corollary 6. *Let $\sigma_1 = (x_1, y_1)$ and $\sigma_2 = (x_2, y_2)$ be two transpositions ($x_i > y_i$, $i = 1, 2$) in S_{64} such that $y_1 > x_2$. Let $\ell_1 = (x_1 - y_1) \geq (x_2 - y_2) = \ell_2$, and $z_i = \frac{\ell_i}{\kappa}$. Let the respective decompositions are denoted by the symbols π_i and θ_i. Then after making the $\overrightarrow{\mathsf{Sel}}$ vectors of the same length by appending zeroes, it is possible to execute σ_1 and σ_2 concurrently on the circuit in Figure 5.2 and achieve $\sigma_1 \circ \sigma_2$ in $64 \cdot z_1$ clock cycles by using $\overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{\mid} \overrightarrow{\mathsf{Sel}}_{\sigma_2}$ as the select signal vector.*

Proof. We have $\mathbb{A}_{\pi_{z_1-1}} = \{x_1, x_1 - \kappa, x_1 - 2\kappa, \ldots, y_1\}$ and $\mathbb{A}_{\theta_{z_2-1}} = \{x_2, x_2 - \kappa, x_2 - 2\kappa, \ldots, y_2\}$. Since $y_1 > x_2$, clearly $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} = \emptyset$. Thus the result follows. □

We can use the results in the above two corollaries to further reduce the execution time of the PRESENT permutation. We have to execute all the transpositions t_i followed by the transpositions s_i. The idea is to execute as many permutations concurrently as possible so long as they have pairwise disjoint $\mathbb{A}_{\pi_{z-1}}$'s. In order to simplify this process, we can easily partition the swaps modulo $\kappa = 3$. Then, the swaps that are in different classes modulo 3 can obviously be executed concurrently, following Corollary 5. Also transpositions in the same class modulo 3 that have disjoint $\mathbb{A}_{\pi_{z-1}}$'s can also be executed together, following Lemma 8. For the t_i's we can think of the following solution given in Table 5.2, that takes $(11 + 7 + 1) \cdot 64 = 704 + 448 + 64 = 1216$ cycles. All the swaps in i-th group can be executed concurrently, thereby reducing the number of cycles.

A similar construction for the s_i's will take $(12 + 12 + 7 + 4) \cdot 64 = 2240$ cycles. So a total of $1216 + 2240 = 3456$ cycles are required which is already much better than our previous construction of 12160 cycles from Section 5.4.3.

5.4.5 Application to GIFT-64

We can apply the outlined steps of optimization on the GIFT block cipher as well. The permutation G_{64} is given in Table 2.3 in Chapter 2. We can make the following observations on the permutation function G_{64}:

1. It is a special permutation of type $\kappa = 4$.
2. It can be decomposed into fourteen 4-cycles and two 2-cycles all of which are pairwise disjoint. Additionally it has 4 fixed points.
3. Each 4-cycle can be decomposed into three transpositions $s_i \circ t_i \circ u_i$. The decomposition is shown in Table 5.3.

Corollary 7. *Let a_i, c_i (resp. b_i) denote a series that is an arbitrary ordering of elements from $[0, 13]$ (resp. from $[0, 15]$). The GIFT permutation G_{64} satisfies that*

$$G_{64} = s_{c_0} \circ \cdots \circ s_{c_{13}} \circ t_{b_0} \circ \cdots \circ t_{b_{15}} \circ u_{a_0} \circ \cdots \circ u_{a_{13}}$$

Table 5.3 – Decomposition of the c_i's in the GIFT permutation

i	c_i	$s_i \circ t_i \circ u_i$
0	$(0, 48, 60, 12)$	$(12, 48) \circ (48, 60) \circ (0, 12)$
1	$(2, 18, 22, 6)$	$(6, 18) \circ (18, 22) \circ (2, 6)$
2	$(3, 35, 43, 11)$	$(11, 35) \circ (35, 43) \circ (3, 11)$
3	$(4, 32, 56, 28)$	$(28, 32) \circ (32, 56) \circ (4, 28)$
4	$(5, 49, 13, 17)$	$(17, 49) \circ (13, 49) \circ (5, 17)$
5	$(7, 19, 39, 27)$	$(19, 27) \circ (19, 39) \circ (7, 27)$
6	$(8, 16, 52, 44)$	$(16, 44) \circ (16, 52) \circ (8, 44)$
7	$(9, 33)$	$(9, 33)$
8	$(10, 50, 30, 38)$	$(38, 50) \circ (30, 50) \circ (10, 38)$
9	$(14, 34, 26, 54)$	$(34, 54) \circ (26, 34) \circ (14, 54)$
10	$(15, 51, 47, 59)$	$(51, 59) \circ (47, 51) \circ (15, 59)$
11	$(20, 36, 40, 24)$	$(24, 36) \circ (36, 40) \circ (20, 24)$
12	$(21, 53, 61, 29)$	$(29, 53) \circ (53, 61) \circ (21, 29)$
13	$(25, 37, 57, 45)$	$(37, 45) \circ (37, 57) \circ (25, 45)$
14	$(31, 55)$	$(31, 55)$
15	$(42, 58, 62, 46)$	$(46, 58) \circ (58, 62) \circ (42, 46)$

for the swaps s_i, t_i, u_i given in Table 5.3.

We can again build our formalism on top of the circuit described in Figure 5.2, but with $\kappa = 4$ in the case of GIFT. Namely, we let $w_\kappa = (4, 0)$ and $r = (63, 62, \ldots, 0)$, and recall that, through application of Lemma 6, any swap (x, y) with $x > y$ and $x \equiv y \bmod 4$ can be constructed with the following sequence of basic operations:

$$(x, y) = r^{(\kappa - x) \bmod 64} \circ w_\kappa \circ (r^\kappa \circ w_\kappa)^{z-1} \circ (r^{64-\kappa} \circ w_\kappa)^{z-1} \circ r^{(x-\kappa) \bmod 64}$$

The optimization that follows after application of this formula is to arrange the swaps into groups for concurrent execution, as Lemma 8 clearly permits this arrangement. The final result of this regrouping can be found in Table 5.4. Overall, the G_{64} can be executed in 4096 clock cycles.

5.5 Multiple-swap Setting

In this section, we look at trade-off between the number of scan flip-flops and the latency of the permutation layer. In other words, we employ multiple scan flip-flops to complete the permutation layer operation in at most few hundreds of clock cycles.

Table 5.4 – Concurrent execution of the t_i and s_i's in the PRESENT permutation

Group	mod4	u_i	$\max(x_i - y_i)$	# of cycles
1	0	$(8, 44)$	44	704
	1	$(25, 45), (5, 17)$		
	2	$(14, 54), (2, 6)$		
	3	$(15, 59), (3, 11)$		
2	0	$(4, 28)$	28	448
	1	$(21, 29)$		
	2	$(10, 38), (42, 46)$		
	3	$(7, 27)$		
3	0	$(0, 12), (20, 24)$	12	192

Group	mod4	s_i	$\max(x_i - y_i)$	# of cycles
1	0	$(16, 52)$	36	576
	1	$(13, 49), (53, 61)$		
	2	$(30, 50), (18, 22)$		
	3	$(19, 39), (47, 51)$		
2	0	$(32, 56)$	24	384
	1	$(37, 57)$		
	2	$(26, 34), (58, 62)$		
	3	$(35, 43)$		
3	0	$(48, 60), (36, 40)$	12	192

Group	mod4	s_i	$\max(x_i - y_i)$	# of cycles
1	0	$(12, 48)$	36	576
	1	$(17, 49)$		
	2	$(34, 54)$		
	3	$(11, 35)$		
2	0	$(16, 44)$	28	448
	1	$(29, 53)$		
	2	$(6, 18)$		
	3	$(31, 55)$		
3	0	$(24, 36)$	24	384
	1	$(9, 33)$		
	2	$(38, 50)$		
	3	$(19, 27)$		
4	0	$(28, 32)$	12	192
	1	$(37, 45)$		
	2	$(46, 58)$		
	3	$(51, 59)$		

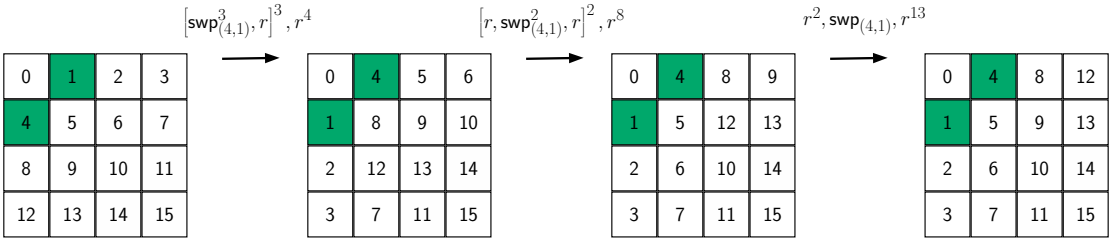

Figure 5.3 – A transposition can be done with r and $\mathsf{swp}_{(4,1)}$ in 3×16 cycles. The operations separated by comma are executed in leftmost-first fashion. The cells corresponding to fixed swap positions of $\mathsf{swp}_{(4,1)}$ are marked with green.

In order to understand how more scan flip-flops can be accommodated, let us start with the basics. Again, we let r denote the simple rotation operation of the pipeline, that is $r \in S_{64}$ such that $r(i) = i - 1 \bmod 64$. Then we additionally introduce swap-then-rotate operations to this pipeline. A swap-then-rotate operation is denoted with $\mathsf{swp}_{(x,y)}$, and it first swaps x and y, and then rotates the pipeline. Namely,

$$\mathsf{swp}_{(x,y)}(x) = (y-1) \bmod 64, \qquad \mathsf{swp}_{(x,y)}(y) = (x-1) \bmod 64$$

and the others elements remain untouched.

We have already seen that a swap-then-rotate operation can be done in the pipeline quite efficiently, i.e. it requires only two extra MUXes before inputs of flip-flops $x-1$ and $y-1$. The technique presented in this section targets reduction of the number of clock cycles, at the expense of few other freshly introduced swap operations into the circuit. For this reason, we extend the notion into multiple-swaps-then-rotate in a natural way. Namely, assuming that $\{x, y\} \neq \{z, t\}$, then $\mathsf{swp}_{(x,y),(z,t)}$ corresponds to application of $r \circ (z, t) \circ (x, y)$. Therefore, first the list of given swaps are executed starting from the leftmost pair, and a final rotation is executed after all swaps are done.

We invite the reader's attention to the difference between the *swap operations*, e.g. $\mathsf{swp}_{(x,y)}$, and *swap permutations*, e.g. 2-cycle (x, y). The former includes a self rotation by definition, and can also include many pairs of swaps at once.

5.5.1 4×4 matrix transposition with swaps

For simplicity, first imagine a 16-bit pipeline whose flip-flop are arranged as 4×4 matrix (see Figure 5.3). Suppose that the pipeline supports only r and $\mathsf{swp}_{(4,1)}$ operations. These two permutations are given in their mathematical forms in Table 5.5, where $r(i)$ denotes the final position of the bit i after r is executed.

Our claim is that the usual 4×4 matrix transposition τ can be written in terms of r and

Table 5.5 – Mathematical forms of some permutations over S_{16}

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$r(i)$	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathsf{swp}_{(4,1)}(i)$	15	3	1	2	0	4	5	6	7	8	9	10	11	12	13	14
$\tau(i)$	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
$\sigma(i)$	0	5	10	15	12	1	6	11	8	13	2	7	4	9	14	3

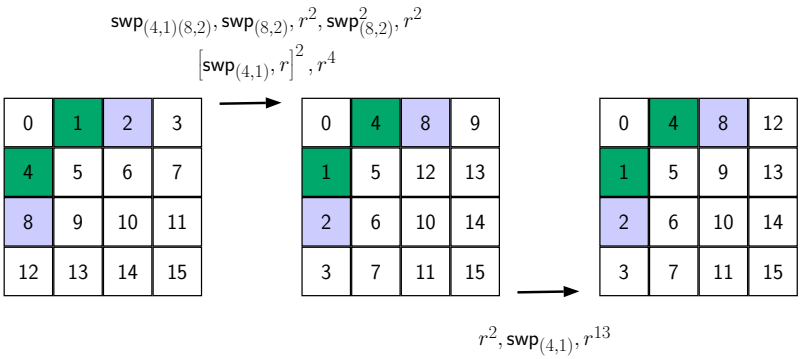

Figure 5.4 – Transposition τ can also be done in 128 clock cycles with 2 swap operations.

$\mathsf{swp}_{(4,1)}$ permutations. Namely, our formula is $\tau = \pi_3 \circ \pi_2 \circ \pi_1$ where

$$\pi_1 = r^4 \circ \left[r \circ \mathsf{swp}^3_{(4,1)}\right]^3$$
$$\pi_2 = r^8 \circ \left[r \circ \mathsf{swp}^2_{(4,1)} \circ r\right]^2$$
$$\pi_3 = r^{13} \circ \mathsf{swp}_{(4,1)} \circ r^2$$

This is demonstrated in Figure 5.3. In conclusion, performing a transposition τ requires three full rotations of the pipeline, i.e. it takes 3×16 cycles, with a single swap operation.

In order to optimize the number of clock cycles spent for each τ application, we can add one or two more swaps into the pipeline. Hence, there is a trade-off between the number of cycles and the circuit area required to execute the permutation. The sequences of operations with two and three swap operations are demonstrated in Figures 5.3, 5.4 and 5.5.

5.5.2 From Transpositions to PRESENT Permutation

Now we give an alternative decomposition of PRESENT's permutation P that is more convenient to realize with a few swaps. The crucial observation is that the permutation

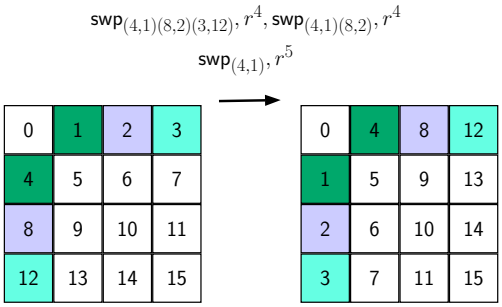

Figure 5.5 – Transposition τ can also be done in 64 clock cycles with 3 swap operations.

P from S_{64} can be written in terms of eight applications of τ from S_{16} so long as we chop 64 bits into four 16-bit matrices in a careful manner. We further use the pipeline rotation to use the same swap operation to perform τ operation on different sub-matrices of the pipeline.

In the first pass, we divide 64 positions $\{0, \ldots, 63\}$ into four *vertical* disjoint matrices, that is we construct M_0, M_1, M_2, M_3 such that i-th row, j-th column of M_r is $16i + j + 4r$ (where columns/rows are indexed from 0 to 3). Then we apply τ on each M_r. In the second pass, we construct *horizontal* matrices N_r such that i-th row, j-th column of N_r is $4i + j + 16r$. Again, τ is applied over each N_r. The choices of 16 indices for M_3 and N_0 are demonstrated in Table 5.6. Below, we give the positions of the bits that are use to fill in M_r and N_r matrices:

$$M_r \leftarrow \begin{bmatrix} 4r & 4r+1 & 4r+2 & 4r+3 \\ 4r+16 & 4r+17 & 4r+18 & 4r+19 \\ 4r+32 & 4r+33 & 4r+34 & 4r+35 \\ 4r+48 & 4r+49 & 4r+50 & 4r+51 \end{bmatrix}$$

$$N_r \leftarrow \begin{bmatrix} 16r & 16r+1 & 16r+2 & 16r+3 \\ 16r+4 & 16r+5 & 16r+6 & 16r+7 \\ 16r+8 & 16r+9 & 16r+10 & 16r+11 \\ 16r+12 & 16r+13 & 16r+14 & 16r+15 \end{bmatrix}$$

More formally, let Z denote an ordered subset $\{z_0, z_1, \ldots, z_{15}\}$ of $\{0, \ldots, 63\}$ (equivalently Z can be considered as a 4×4 matrix). Then we define the permutation $\tau_Z \in S_{64}$ as applying $\tau \in S_{16}$ over Z while keeping the other 48 bits untouched. Which is to say, given $i \in \{0, \ldots, 63\}$, if $i = z_j$ for some j then $\tau_Z(i) = \tau_Z(z_j) = z_{\tau(j)}$, and otherwise (if $i \notin Z$) then $\tau_Z(i) = i$. Our claim is that $P = \tau_{N_0} \circ \tau_{N_1} \circ \tau_{N_2} \circ \tau_{N_3} \circ \tau_{M_0} \circ \tau_{M_1} \circ \tau_{M_2} \circ \tau_{M_3}$.

For a particular choice of Z, we need to consider that $\tau_Z \in S_{64}$ can differ from $\tau \in S_{16}$ in two ways: 1) the positions of (x, y) in $\mathsf{swp}_{(x,y)}$ operation and 2) the number of $r \in S_{64}$ applications necessary to complete a full rotation in Z. For the former, we need to choose (z_4, z_1) as swap positions instead of $(4, 1)$. For the latter, we need to update our schedule

Table 5.6 – The movement of bits according to the PRESENT's permutation function P, where $\pi_{i\to j}$ denotes $\pi_i \circ \pi_{i-1} \circ \cdots \circ \pi_j$.

$\mathcal{I}$																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$\pi_{3\to1}(\mathcal{I})$	0	16	32	48	4	20	36	52	8	24	40	56	12	28	44	60
	1	17	33	49	5	21	37	53	9	25	41	57	13	29	45	61
	2	18	34	50	6	22	38	54	10	26	42	58	14	30	46	62
	3	19	35	51	7	23	39	55	11	27	43	59	15	31	47	63
$\pi_{6\to4}(\mathcal{I})$	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
	1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
	2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
	3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63

of operations. For instance, M_3 requires 64 cycles of r to complete its full rotation instead of 16. That means during τ_{M_r} operations, $r \in S_{64}$ that rotates the pipeline is actually different than the one we used previously, i.e. $r \in S_{16}$, to formulate $\tau \in S_{16}$. In particular, since the pipeline consists of 64 bits, it takes 16 cycles for the second row of M to move to its first row. Hence, we need to update our decomposition sequences to interleave τ_{M_r} operations.

We interleave τ_M operations as follows. Given

$$\begin{aligned}
\pi_1 &= r^{16} \circ \left[r \circ \mathsf{swp}^3_{(16,1)} \right]^{12} \\
\pi_2 &= r^{32} \circ \left[r \circ \mathsf{swp}^2_{(16,1)} \circ r \right]^{8} \\
\pi_3 &= r^{48} \circ \left[r \circ \mathsf{swp}_{(16,1)} \circ r^2 \right]^{4}
\end{aligned}$$

then $\pi_3 \circ \pi_2 \circ \pi_1 = \tau_{M_0} \circ \tau_{M_1} \circ \tau_{M_2} \circ \tau_{M_3}$. And for τ_N operations, given

$$\begin{aligned}
\pi_4 &= \left[r^4 \circ \left[r \circ \mathsf{swp}^3_{(4,1)} \right]^{3} \right]^{4} \\
\pi_5 &= \left[r^8 \circ \left[r \circ \mathsf{swp}^2_{(4,1)} \circ r \right]^{2} \right]^{4} \\
\pi_6 &= \left[r^{13} \circ \mathsf{swp}_{(4,1)} \circ r^2 \right]^{4}
\end{aligned}$$

then $\pi_6 \circ \pi_5 \circ \pi_4 = \tau_{N_0} \circ \tau_{N_1} \circ \tau_{N_2} \circ \tau_{N_3}$. Finally, $P = \pi_6 \circ \pi_5 \circ \pi_4 \circ \pi_3 \circ \pi_2 \circ \pi_1$. The full worked-out schedules and decomposition of PRESENT permutation is given in Table 5.8.

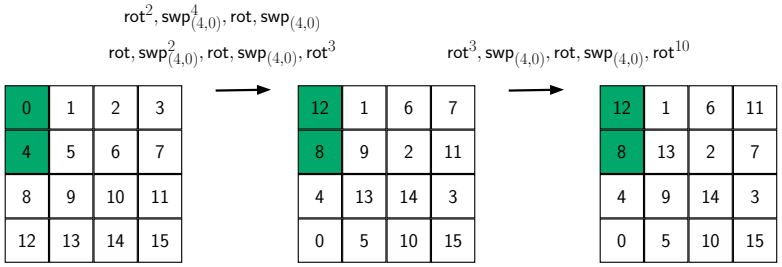

Figure 5.6 – Performing σ with $\mathsf{swp}_{(4,0)}$ and r in 2 full rounds, i.e 2×64 cycles.

Note that we require 2 different swaps (therefore 4 scan flip-flops) to work this out. Each π_{i} requires 64 cycles and hence the permutation can be realized with 6×64 cycles.

5.5.3 From Transpositions to GIFT-64 Permutation

The decomposition of GIFT permutation G_{64} is slightly different than P. We choose our matrices such that the first operation becomes transposition τ over nibbles instead of bits, and the second one consists of series of ad hoc swaps described as the permutation σ in Table 5.5. In the same fashion, σ is a permutation over S_{16}, but we can extend it to S_{64} by defining σ_Z for Z being an ordered subset of $\{0, \ldots, 63\}$ as before.

Performing G takes four applications of τ followed by four applications of σ. We choose our matrices as follows. The i-th row, j-th column of M_r is $4i + 16j + r$. Then we apply τ on M_r matrices. In the second pass, the i-th row, j-th column of N_r is $16i + j + 4r$. Again, σ is applied over N_r matrices. Our finding is that $G_{64} = \sigma_{N_0} \circ \sigma_{N_1} \circ \sigma_{N_2} \circ \sigma_{N_3} \circ \tau_{M_0} \circ \tau_{M_1} \circ \tau_{M_2} \circ \tau_{M_3}$. The structure of this decomposition is illustrated in Table 5.7 and the sequence of operations to realize G_{64} with minimum number of swaps are presented in Table 5.9.

5.5.4 Inverse Permutations for Decryption

One might notice that neither G_{64} nor P are involution, that is $P(P(i)) = i$ does not hold. This means that the permutation logic for encryption cannot be readily used in decryption. A straightforward idea for decryption that avoids adding extra gates could be based on the fact that P^3 and G^4 are identity permutations. Hence, one can repeat P and G two and three times respectively to get their inverse permutation. However, this is not an optimal solution, as it double or triples the number of cycles required for the inverse permutation layer, making decryption significantly more costly than encryption.

Our technique also enables more elegantly solution for the inverse permutations. We draw attention to the fact that, because of the manner we decomposed both permutations, τ and σ are in fact involutions. This can be easily deduced from the fact that they are,

Table 5.7 – The movement of bits according to the GIFT's permutation function G_{64}, where $\pi_{i\to j}$ denotes $\pi_i \circ \pi_{i-1} \circ \cdots \circ \pi_j$.

$\mathcal{I}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$\pi_{3\to1}(\mathcal{I})$	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51
	4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55
	8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59
	12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63
$\pi_{6\to4}(\mathcal{I})$	12	1	6	11	28	17	22	27	44	33	38	43	60	49	54	59
	8	13	2	7	24	29	18	23	40	45	34	39	56	61	50	55
	4	9	14	3	20	25	30	19	36	41	46	35	52	57	62	51
	0	5	10	15	16	21	26	31	32	37	42	47	48	53	58	63

in return, compositions of multiple disjoint swap operations. Hence, for decryption, it suffices to change the order of executions. As an example, for PRESENT, we only need to run τ_{N_r} permutations in the first pass, and τ_{M_r} in the second pass for decryption. The number of cycles and trade-offs remain exactly the same. No extra gates or cycles are required. In conclusion, the advantage of decomposing a permutation with our swap-based technique is twofold: it adds quite small amount of gates (2 extra MUXes for each swap), and it readily supports decryption with no extra cost, even if the composed permutation is not an involution and might seem to require some extra gates for its inverse.

5.6 Final Interleaving Optimization

This section is dedicated to the goal of decreasing the latency even further. In Sections 5.5.2 and 5.5.3, we have shown how PRESENT and GIFT implementations can be realized with very small additional cost, i.e. 4 to 12 scan flip-flops (i.e. 2 to 6 swaps). Even though our approach achieves roughly 20 % reduction in the circuit area, it causes the latency of the circuit to increase to threefold. Hence, in this section we show that by carefully arranging all swap operations to run concurrently, we can beat the state-of-the-art implementations of PRESENT and GIFT [JMPS17], in terms of both latency and circuit-size.

Building upon our finding in Section 5.5.2, we provide realization of PRESENT and GIFT permutations with 6 swaps that require no additional clock cycles. While encryption/decryption rounds take precisely 64 cycles to complete for each round (AddRoundKey and SubBytes), our permutation layer operates on the state pipeline seamlessly to ensure that each bit leaving the pipeline is already moved to its permuted position. There is no need to freeze the state pipeline or allocate extra clock cycles to the permutation layer either.

Table 5.8 – The sequence of operations for PRESENT's permutation layer, and the trade-off between the number of clock cycles vs. the number of swaps.

# swaps	round	cycles	decomposition
2	1	0-47	$[\mathsf{swp}^3_{(16,1)}, r]^{12}$
		48-63	r^{16}
	2	0-31	$[r, \mathsf{swp}^2_{(16,1)}, r]^8$
		32-63	r^{32}
	3	0-15	$[r^2, \mathsf{swp}_{(16,1)}, r]^4$
		16-63	r^{48}
	4	0-63	$([\mathsf{swp}^3_{(4,1)}, r]^3, r^4)^4$
	5	0-63	$([r, \mathsf{swp}^2_{(4,1)}, r]^2, r^8)^4$
	6	0-63	$[r^2, \mathsf{swp}_{(4,1)}, r^{13}]^4$
4	1	0-15	$[\mathsf{swp}_{(16,1),(32,2)}, \mathsf{swp}_{(32,2)}, r^2]^4$
		16-47	$[\mathsf{swp}^2_{(32,2)}, r^2]^4, [\mathsf{swp}_{(16,1)}, r]^8$
		48-64	r^{16}
	2	0-63	$[r^2, \mathsf{swp}_{(16,1)}, r]^4, r^{48}$
	3	0-3, 16-19, 32-35, 48-51	$\mathsf{swp}_{(4,1),(8,2)}, \mathsf{swp}_{(8,2)}, r^2$
		4-15, 20-31, 36-47, 52-63	$\mathsf{swp}^2_{(8,2)}, r^2, [\mathsf{swp}_{(4,1)}, r]^2, r^4$
	4	0-63	$[r^2, \mathsf{swp}_{(4,1)}, r^{13}]^4$
6	1	0-15	$[\mathsf{swp}_{(16,1),(32,2),(15,60)}, r^3]^4$
		16-31	$[r, \mathsf{swp}_{(16,1),(32,2)}, r^2]^4$
		32-63	$[r^2, \mathsf{swp}_{(16,1)}, r]^4, r^{16}$
	2	0-7, 16-23, 32-39, 48-55	$\mathsf{swp}_{(4,1),(8,2),(51,60)}, r^4, \mathsf{swp}_{(4,1),(8,2)}, r^2$
		8-15, 24-31, 40-47, 56-63	$r^2, \mathsf{swp}_{(4,1)}, r^5$

Table 5.9 – The sequence of operations for GIFT's permutation layer, and the trade-off between the number of clock cycles vs. the number of swaps.

# swaps	round	cycles	decomposition
2	1	0-63	$[\mathsf{swp}^{12}_{(16,4)}, r^4]^3, r^{16}$
	2	0-63	$[r^4, \mathsf{swp}^{8}_{(16,4)}, r^4]^2, r^{32}$
	3	0-63	$r^8, \mathsf{swp}^{4}_{(16,4)}, r^{52}$
	4	0-31	$[r^2, \mathsf{swp}^{2}_{(16,0)}]^4, [\mathsf{swp}^{2}_{(16,0)}, r, \mathsf{swp}_{(16,0)}]^4$
		32-63	$[r, \mathsf{swp}^{2}_{(16,0)}, r]^4, [\mathsf{swp}_{(16,0)}, r^3]^4$
	5	0-63	$[r^3, \mathsf{swp}_{(16,0)}]^4, [r, \mathsf{swp}_{(16,0)}, r^2]^4, r^{32}$
4	1	0-23	$\mathsf{swp}^{4}_{(16,4),(32,8),(48,12)}, r^{16}, \mathsf{swp}^{4}_{(16,4),(32,8)}$
		25-63	$r^{16}, \mathsf{swp}^{4}_{(16,4)}, r^{20}$
	2	0-31	$[r^2, \mathsf{swp}^{2}_{(16,0)}]^4, [\mathsf{swp}^{2}_{(16,0)}, r, \mathsf{swp}_{(16,0)}]^4$
		32-63	$[r, \mathsf{swp}^{2}_{(16,0)}, r]^4, [\mathsf{swp}_{(16,0)}, r^3]^4$
	3	0-63	$[r^3, \mathsf{swp}_{(16,0)}]^4, [r, \mathsf{swp}_{(16,0)}, r^2]^4, r^{32}$
6	1	0-23	$\mathsf{swp}^{4}_{(16,4),(32,8),(48,12)}, r^{16}, \mathsf{swp}^{4}_{(16,4),(32,8)}$
		25-63	$r^{16}, \mathsf{swp}^{4}_{(16,4)}, r^{20}$
	2	0-31	$[\mathsf{swp}_{(48,0)}, r, \mathsf{swp}_{(16,0),(33,1)}, r]^4, [\mathsf{swp}_{(16,0),(33,1)}, r^3]^4$
		32-63	$[r^2, \mathsf{swp}_{(16,0)}, r]^4, r^{16}$

In comparison, the smallest known implementation from Jean et al. [JMPS17] requires 4 additional cycles each round, leading to a loss of more than a hundred cycles in latency. This is because the additional circuitry that handles the permutation layer requires four cycles to complete the permutation, during which AddRoundKey and SubBytes layers must be stalled. Our implementation of permutation layer, on the other hand, reaches to maximum utilization in a bit-serial implementation architecture, as it brings no additional cycles.

The intuition behind interleaving is the fact that disjoints permutations can be executed in arbitrary order. As swaps are simply 2-cycles, given (a, b), (c, d) checking whether they are disjoint is straightforward. On the contrary, if two overlapping swaps are given, e.g. (a, b), (b, c), then we must preserve the order between them. If such a choice of swaps emerge, one needs to be cautious about which exact swap is run at some given clock cycle, as swp operations on the hardware actually operate at different 2-cycles. For instance, $\mathsf{swp}_{(11,14)}$ can execute any permutation from the set $\{(11 + i \bmod 64, 14 + i \bmod 64)\}$ for the clock cycles i in which it is active. Hence, if we expand the operation sequences that leads to two rounds from Tables 5.8, 5.9 into a series of actual 2-cycle applications by replacing each $\mathsf{swp}_{(x,y)}$ at active clock cycle i with 2-cycle $(x + i \bmod 64, y + i \bmod 64)$, the following question arises:

Question. *Can the expanded sequence of swaps (which takes 128 clock cycles according*

to Sections 5.5.2 and 5.5.3) be squeezed into fewer number of clock cycles (close to 64) so that we can complete the permutation layer in one pass (single round)?

Fortunately, the answer to this question is affirmative. Furthermore, if we make our choices for the initial swap operations wisely, we can even use the exact same swap operations in a combined encryption and decryption circuit for PRESENT permutation. For GIFT permutation, we need to add two more swaps for the combined circuit. The fully worked-out schedule and the carefully chosen six swaps are given in Table 5.10, for both encryption and decryption circuits. In summary, we achieve the following permutation layer implementations:

1. PRESENT encryption-only circuit with 6 swaps can compute the permutation layer in 64 clock cycles,
2. PRESENT encryption and decryption combined circuit with 6 swaps can compute the permutation layer in 64 clock cycles,
3. GIFT encryption-only circuit with 6 swaps can compute the permutation layer in 64 clock cycles,
4. GIFT decryption-only circuit with 6 swaps can compute the permutation layer in 64 clock cycles,
5. GIFT encryption and decryption combined circuit with 8 swaps can compute the permutation layer in 64 clock cycles.

5.7 Conclusion

In this chapter, we looked at swap-based optimization techniques for the execution of permutation layers of block ciphers in serial architectures. The Chapter 6 will build on top of these results to achieve a large number of block ciphers, which again have the smallest reported latency-area trade-off reported in the literature. For instance, according to STM 90 nm CMOS logic process, our work presented at FSE 2020 reports implementations of GIFT-64 and PRESENT-80 costing 907 GE and 694 GE for encryption-only circuits, and 1055 GE and 786 GE for combined encryption and decryption circuits respectively [BBRV20]. These are the smallest known implementations of these two block ciphers.

More specifically, we tried to answer the question if bit-permutations like the one used in the linear layers of block ciphers PRESENT and GIFT can be executed in a flip-flop array using only two scan flip-flops. With the help of permutation theory, the response is affirmative, however straightforward application of the ideas that follow the proofs take lots of clock cycles, and thus affect the throughput of the resulting circuit drastically.

Table 5.10 – Realization of GIFT and PRESENT permutations in 64 clock cycles.

cipher	mode	swap	active clock cycles
PRESENT	enc	(58, 43)	$\{0, 4, 22, 26, 30, 34, 39, 43, 47, 51, 56, 60\}$
		(59, 29)	$\{2, 37, 41, 45, 49, 54, 58, 62\}$
		(60, 15)	$\{0, 52, 56, 60\}$
		(6, 3)	$\{3, 8, 14, 19, 24, 30, 35, 40, 46, 51, 56, 62\}$
		(8, 2)	$\{0, 5, 16, 21, 32, 37, 48, 53\}$
		(10, 1)	$\{2, 18, 34, 50\}$
PRESENT	dec	(58, 43)	$\{3, 7, 11, 15, 33, 37, 41, 45, 50, 54, 58, 62\}$
		(59, 29)	$\{1, 5, 9, 13, 48, 52, 56, 60\}$
		(60, 15)	$\{3, 7, 11, 63\}$
		(6, 3)	$\{3, 9, 14, 19, 25, 30, 35, 41, 46, 51, 57, 62\}$
		(8, 2)	$\{0, 11, 16, 27, 32, 43, 48, 59\}$
		(10, 1)	$\{13, 29, 45, 61\}$
GIFT	enc	(51, 39)	$\{5, 6, 7, 8, 29, 30, 31, 32, 49, 50, 51, 52\}$
		(50, 26)	$\{2, 3, 4, 5, 46, 47, 48, 49\}$
		(49, 13)	$\{0, 1, 2, 63\}$
		(18, 2)	$\{0, 4, 8, 12, 14, 18, 22, 26, 32, 36, 40, 44\}$
		(33, 1)	$\{2, 6, 10, 14, 16, 20, 24, 28\}$
		(48, 0)	$\{0, 4, 8, 12\}$
GIFT	dec	(19, 7)	$\{0, 1, 2, 3, 20, 21, 22, 23, 40, 41, 42, 43\}$
		(32, 8)	$\{3, 4, 5, 6, 23, 24, 25, 26\}$
		(49, 13)	$\{2, 3, 4, 5\}$
		(18, 2)	$\{49, 53, 57, 61\}$
		(33, 1)	$\{37, 41, 45, 49, 51, 55, 59, 63\}$
		(48, 0)	$\{1, 21, 25, 29, 33, 35, 39, 43, 47, 53, 57, 61\}$

The great portion of the chapter is then dedicated to reducing the number of operations required to execute the bit permutation in this setting. Eventually we are able to show that as few as six swaps are sufficient to implement the permutations layers of PRESENT and GIFT as fast as it can be expected from any serial circuit. For the sake of consistency, the circuit implementation details are left to Chapter 6.

6 The Area-Latency Symbiosis through Swap-and-Rotate

The results presented in this chapter are based on the work done in collaboration with Andrea Caforio and Subhadeep Banik [BCB21], which will be presented in TCHES 2021. The authors Fatih Balli and Subhadeep Banik were supported by the Swiss National Science Foundation (SNSF) through the Ambizione Grant PZ00P2_179921.

In Section 6.1, we first give an extract on the authenticated encryption candidates from NIST LWC, and we also summarize our motivation. In Section 6.2, we give the details of our contributions. We use Section 6.3 to explain our generic high-level approach. Sections 6.4, 6.5, 6.6 and 6.7 are respectively dedicated to detailed descriptions of lightweight circuits for AES, SKINNY, GIFT* and GIFT respectively. Each of these sections include the implementation details for both 1-bit and 4/8-bit serial circuits. In Section 6.8, we turn our attention to serial implementation of four AEAD candidates: SUNDAE-GIFT, SAEAES, Romulus and SKINNY-AEAD. In Section 6.9, we explain how our circuit implementations can be tweaked to support decryption functionality, and what is the associated cost. Lastly, Section 6.10 is the conclusion.

The VHDL source code of our implementations can be found in a public git repository [ALS], and they also appear as artifact in TCHES 2021 [BCB20].

6.1 Related Work

Mode of operation. As of the second round, 13 out of 32 candidates in NIST LWC are based on block ciphers [NISa]. These candidates simply design a mode of operation around a given block cipher to function as an authenticated encryption scheme. From Table 6.1, note that:

- 4 candidates use either the standard or a tweaked version of AES as the primary choice. These are COMET, ESTATE, mixFeed, SAEAES [GJN19, CDJ+19a, CN19, NMMaS+19].

Table 6.1 – The list of second-round NIST LWC candidates that are based on block ciphers. GIFT refers to the original block cipher from [BPP+17], and GIFT* refers to the version that assumes different ordering of the input bits and the key [BCI+19, BBP+19].

Candidate	Primary		Alternative		
	cipher	size (block+key)	cipher	size (block+key)	cipher calls
COMET	AES	128+128	CHAM, Speck	64/128+128	enc
ESTATE	TweAES	128+128	TweGIFT	128+128	enc+dec
ForkAE	ForkSKINNY	128+288	ForkSKINNY	64/128+192/256/288	enc+dec
GIFT-COFB	GIFT*	128+128	-	-	enc
HYENA	GIFT	128+128	-	-	enc
LOTUS-AEAD	TweGIFT	64+128	-	-	enc
mixFeed	AES	128+128	-	-	enc
Pyjamask	Pyjamask	128+128	Pyjamask	96+128	enc
Romulus	SKINNY	128+384	SKINNY	128+256/384	enc
SAEAES	AES	128+128	-	-	enc
Saturnin	Saturnin	256+256	-	-	enc
SKINNY-AEAD	SKINNY	128+384	SKINNY	128+256/384	enc
SUNDAE-GIFT	GIFT*	128+128	-	-	enc

- 3 candidates use either the standard or a forked version of SKINNY as the primary choice. These are ForkAE, Romulus, SKINNY-AEAD [ALP+19, IKMP19, BJK+19].
- 2 candidates use either the standard or a tweaked version of 128-bit variant of GIFT. Namely, ESTATE uses the tweaked version of GIFT as an alternative choice [CDJ+19a], whereas HYENA uses the original version [CDJN19, BPP+17]. Besides, LOTUS-AEAD also employs a tweaked 64-bit variant of GIFT [CDJ+19b, BPP+17].
- 2 candidates use GIFT*. These two candidates are GIFT-COFB and SUNDAE-GIFT [BCI+19, BBP+19]. The difference between GIFT* and GIFT is that the former assumes a different indexing of the input and output bits. We denote their modified version with GIFT*, as it leads to a significant difference from a design and implementation perspective (but remains equivalent in terms of cryptanalysis).
- Pyjamask and Saturnin are the exceptions to the popular approach, as they bring their own dedicated block cipher designs into the standardization [GJK+19, CDL+19].

We should note that only Romulus and SKINNY-AEAD are the two AE designs that made it to the final round [NISa].

Given the modular approach taken by these candidates, one can pose the two following questions, which relates directly to their lightweight performance:

1. How lightweight is the block cipher employed at the core?
2. What is the cost of the surrounding mode of operation?

This chapter responds to these two significant questions.

Bottleneck of storage. Most low-area implementations of SPN-based block ciphers eventually face a bottleneck of storage, quite similar to the fashion in which they are observed in hash functions [BLP+08]. Namely, all implementations (except those that are fully unrolled) need to store the key and the cipher state during the encryption operation. For a block cipher with ℓ_b-bit block and ℓ_k-bit key, this typically requires the use of $\ell_b + \ell_k$ flip-flops. More concretely, the area-cost of storing 256-bit corresponds to 1088 GE, 832 GE and 1451 GE for the cell libraries UMC 90 nm, STM 90 nm and NanGate 45 nm respectively. It naturally follows that for the smallest implementations of AES (resp. SKINNY-128-128, GIFT), the 73% (resp. 83%, 69%) of the circuit is due to merely D flip-flops [JMPS17, BPP+17]. Therefore, the state-of-the-art ultra-small ASIC implementations of block ciphers contain mostly storage elements, and space for further area optimizations is limited. We take this as an indicator that we should divert our focus to the other aspects of the circuit while remaining in the same area budget.

Two previous papers that immediately evoke comparison with our work are the ones by Jean et al. and Banik et al. [JMPS17, BBRV20]. The former was the first paper to propose a 1-bit datapath implementation of AES, PRESENT and SKINNY family. However none of the implementations reported in this paper achieved a latency per round figure equal to the block size of the underlying block cipher. One of the reasons for this is that they approach the entire round function as a monolithic entity, i.e. all the algebraic operations in the round were completed in the time period allotted for the round.

Large amounts of the engineering in this chapter is devoted towards investigating what happens if we flirt with these boundaries by executing some operations of round i in the time period allotted to round $i+1$ while maintaining correct functionality. As it turns out, when we do this efficiently, we can limit latency per round (in clock cycles) to the block size and that is exactly the mathematical challenge we faced in this work. To make things clearer, note that Jean at al. reports a bit-serial implementation of AES which completes a round in 168 cycles [JMPS17]. 128 clock cycles are used to rotate bits across the state pipeline and perform AddRoundKey and SubBytes operations simultaneously. Precisely 8 cycles are used for ShiftRows and 32 more for MixColumns. However, we observed that we could be more flexible in the scheduling of operations, which is to say neither do we have to wait for the AddRoundKey and SubBytes operation to be completed on the entire 128-bit state to begin ShiftRows, nor wait for the ShiftRows to complete to begin MixColumns. When a group of bits in the state have undergone AddRoundKey and SubBytes, we can already begin ShiftRows on those bits immediately and the same holds for the scheduling of MixColumns vis-à-vis ShiftRows. In the process of developing this technique, we find that not all operations of a round is finished in the time allocated for the round, and so we improvise and try to get them done in the next round, while trying to maintain functionality at all times.

A preliminary version of the aforementioned technique, targeting specifically the permutation layers of GIFT and PRESENT, but without generalization to other family of block ciphers is already summarized in Chapter 5. This chapter extends on these results in order to generalize it to implementations of higher-bit datapaths and other family of block ciphers.

6.2 Contributions

We provide 1/4/8-bit-serial architectures for the popular 128-bit block size variants of the block ciphers AES, SKINNY, GIFT* and GIFT, which are popular among NIST candidates as of the second round. Our implementations can be employed by 10 candidates out of 13 listed in Table 6.1. Our approach has the following benefits, and the detailed comparison with the state of the art is summarized in Table 6.2:

- In terms of circuit area, each of our block cipher implementations is an evident contender to be the smallest implementation.
- Each implementation fully utilizes both the state and the key pipelines. With 1-bit datapath, each round consisting of 128-bit is executed exactly in 128 clock cycles. This ensures that we get the maximum throughput from 1-bit-serial implementation. This leads to an approximately 20% reduction in latency (in clock cycle units) over the circuits reported by the previous work [JMPS17, BPP+17] (note that the AES, SKINNY, GIFT circuits in these papers report a latency of 168, 168, 160 cycles per round respectively). Our circuit design is novel in the sense that both pipelines are continuously active.
- With 8-bit datapath, each round consisting of 128 bits is executed exactly in 16 clock cycles. This leads to a roughly 20% reduction in latency over the circuits reported in [JMPS17, BBR16a] (note that the AES, SKINNY, GIFT circuits in these papers report a latency of 21, 21, 20 cycles per round respectively).
- Each implementation respects the standard ordering of input and output bits. We do not make a non-standard assumption on the ordering of the bits to reduce the area and latency. Namely, we ensure that an implementation from our paper is readily usable from a NIST LWC candidates without having to modify and deal with the ordering the bits. Some implementations of AES, e.g. [MPL+11, JMPS17, BBR16a, BBR16b], assume that plaintext and the key is arranged in a row major fashion (which we call non-standard), even though the original specification of AES assumes a column-major arrangement [NIS01].
- We avoid techniques such as clock-gating, which might sometimes result in timing inconsistencies during synthesis phase and cost additional circuit area. This also brings the additional benefit of being compatible with the recently introduced

Table 6.2 – The comparison of our work with the state of the art in terms of latency, area and energy. The measurements respect to the use of the same library and clock frequency, NanGate 45 nm and UMC 90 nm for AES, SKINNY and STM 90 nm for GIFT*. It has been estimated in [MPL+11] that converting a non-standardized to a standardized circuit requires an additional 20 MUXes. The area figures in this row is obtained by adding the area of 20 MUXes to the figures in the previous row. [b]GIFT* refers to the slightly modified version of GIFT used in SUNDAE-GIFT [BBP+19].

Block cipher	Area (GE)		Latency (cycles)		Energy (nJ/128-bit)		Ref.
	NanGate 45	UMC 90	round	total	NanGate 45 @ 100 KHz	UMC 90 @ 100 KHz	
AES (standard)	1974	1600	**128**	**1408**	**1441.8**	**7.7**	Sec. 6.4
AES (non-standard)	1982	1596	168	1776	1779.6	11.9	[JMPS17]
AES (standardized)[a]	2029	1641	168	1904			[JMPS17]
SKINNY-128-128	1748	1355	**128**	**5248**	**4602.0**	**22.6**	Sec. 6.5
SKINNY-128-128	1740	1363	168	6976	6045.4	39.1	[JMPS17]
SKINNY-128-256	2502	1927	**128**	**6272**	**7837.5**	**38.9**	Sec. 6.5
SKINNY-128-256	2501	1937	168	8448	10432.4	97.2	[JMPS17]
SKINNY-128-384	3263	2518	**128**	**7296**	**11877.2**	**59.8**	Sec. 6.5
SKINNY-128-384	3260	2508	168	9920	15875.0	153.8	[JMPS17]
	STM 90	UMC 90			STM 90 @ 10 MHz	UMC 90 @ 10 MHz	
GIFT	1215	1531	**128**	**5248**	27.2	26.9	Sec. 6.7
GIFT	1213	-	160	6528	26.3		[BPP+17]
GIFT* [b]	1108	1332	**128**	**5248**	**26.1**	**25.5**	Sec. 6.6

glitch-resistant security model [BGI+18]. This paper provides framework for formal verification of masked designs in the presence of glitches and is thus a very useful tool to have for implementors. However, the model used in the paper assumes that all registers are triggered by a perfectly synchronized clock signal. In the case of clock-gating, this assumption does not hold, because gated clock has variable delay in comparison to the main clock source. Thus when our techniques are used to produce masked implementations by simply duplicating the combinatorial and storage circuitry, it has the added advantage of conforming to this security model, which would make it easier for the circuit designer to formally verify the security of the circuit in the presence of glitches.

In the second part of this chapter, we direct our attention to implementation of four AE schemes, one for each block cipher: SUNDAE-GIFT (1201 GE), SAEAES (1350 GE), Romulus (2399 GE) and SKINNY-AEAD (3589 GE), with their respective area costs in STM 90 nm technology library. We have chosen these candidates, because the mode of operation part of the circuit has the minimal storage requirement, thus leading to very compact implementations. To the best of our knowledge, these are the smallest block-cipher-based authenticated encryption schemes reported so far in the bit-serial and 4/8-bit-serial configurations. In Table 6.3, we summarize the synthesis figures for the

Table 6.3 – The comparison of our work with regards to 8-bit-serial AES, SKINNY and 4-bit-serial GIFT state-of-the-art implementations. [a]The number of clock cycles is incorrectly reported for the 128-bit version of GIFT in [BPP+17] (and confirmed by the authors). We report the rectified figures for the respective implementation. [b]GIFT* refers to the slightly modified version of GIFT used in SUNDAE-GIFT [BBP+19]

Block cipher	Area (GE) STM 90	Area (GE) UMC 180	Latency (cycles) round	Latency (cycles) total	Energy (nJ/128-bit) STM 90 @ 10 MHz	Energy (nJ/128-bit) UMC 180 @ 100 KHz	Ref.
AES (standard)	1785	-	**16**	**176**	1.84	-	Sec. 6.4.6
AES (non-standard)	-	2400	21	226	-	8.36	[MPL+11]
AES (non-standard)	2060	-	23	246	-	3.2	[BBR16b]
SKINNY-128-128	1326	-	**16**	**656**	**4.10**	-	Sec. 6.5
SKINNY-128-128	1638	-	21	840	6.64	-	[BPP+17]
SKINNY-128-128	-	1840	21	872	-	-	[BJK+16]
SKINNY-128-256	1880	-	**16**	**784**	**7.10**	-	Sec. 6.5
SKINNY-128-256	-	2655	21	1040	-	-	[BJK+16]
SKINNY-128-384	2431	-	**16**	**912**	**10.9**	-	Sec. 6.5
SKINNY-128-384	-	3474	21	1208	-	-	[BJK+16]
GIFT [a]	1455	-	33	1352	10.78	-	[BPP+17]
GIFT* [b]	1430	-	**32**	**1312**	**8.06**	-	Sec. 6.6

multi-bit implementations and similarly, Table 6.4 tabulates the synthesis figures for our AEAD constructions under the same technology library. The measurements for other libraries can be found in Section 6.8.5. The source code of our implementations are also publicly available [ALS, c4s].

6.3 Generic Approach

An SPN-based block cipher generally consists of three layers of operation in a round: key addition, substitution and a linear operation. The linear layer is often a combination of a permutation function and a matrix multiplication. For example in AES, the permutation function is the ShiftRows operation and matrix multiplication is done by the MixColumns operation. In the context of lightweight circuits, we can further classify these operations into 2 broad classes: (1) swap-based and (2) replacement-based. In AES, for example, the SubBytes and MixColumns operations can be seen as replacement-based operations, since they take a finite portion of the AES state and replace them with new data block of equal length. ShiftRows can be seen as a swap-based operation because it essentially swaps some bits at two different locations of the state vector. Our technique, for implementing an SPN-based block cipher, then consists of finding a good and short sequence of swap operations that corresponds to the swap-based operation, and interleave them with the replacement-based operations.

In particular, let us look at AES as an example. We recall that the same permutation τ

Table 6.4 – Synthesis figures for the implemented bit-serial AEAD schemes using the STM 90 nm process. Energy and throughput figures are based on the processing of 1024 bits of plaintext and 128 bits of associated data.

		Area (GE)	Power (μW @ 10 MHz)	Latency (cycles)	Energy (nJ/1152-bit)	Ref.
SUNDAE-GIFT	1-bit	**1201**	50.1	92544	463.6	Sec. 6.8.1
SUNDAE-GIFT	4-bit	1587	63.9	23136	147.8	Sec. 6.8.1
SAEAES	1-bit	1350	77.2	24448	188.7	Sec. 6.8.2
SAEAES	8-bit	1940	108.0	3056	33.0	Sec. 6.8.2
Romulus-N1	1-bit	2399	98.1	64647	634.2	Sec. 6.8.3
Romulus-N1	8-bit	2912	114.0	8080	92.6	Sec. 6.8.3
SKINNY-AEAD	1-bit	3589	134.3	72960	979.9	Sec. 6.8.4
SKINNY-AEAD	8-bit	3783	149.0	9856	146.9	Sec. 6.8.4

was considered in Section 5.5.1. Here, we look at it in more detail. At the byte level, the ShiftRows is a permutation over the set $[0, 15]$ which can be formulated as

$$(1, 13, 9, 5) \circ (2, 10) \circ (6, 14) \circ (3, 7, 11, 15)$$

Given that the AES byte order is $b_0, b_1, \ldots, b_{15}$, the above notation means that after ShiftRows, b_1 is moved to location 13, b_{13} is moved to location 9 etc. Note that each of the k-cycles correspond to a particular row of the AES state and they commute with each other, so the order of their execution is irrelevant. The above expression can be decomposed further as

$$[(9, 13) \circ (5, 9) \circ (1, 5)] \circ (2, 10) \circ (6, 14) \circ [(11, 15) \circ (7, 15) \circ (3, 15)]$$

What does this tell us? First, the permutation is special and of type 4, since for all the swaps (x, y) listed above we have $y - x \equiv 0 \pmod 4$.

Let us turn our attention to the first 4-cycle which decomposes as $(9, 13) \circ (5, 9) \circ (1, 5)$ (note that these swaps no longer commute). We will show how to implement this 4-cycle in 16 clock intervals. Let us choose to implement the $(11, 15)$ swap in the circuit for this purpose, for which we place scan-flip-flop-based byte registers in locations 10 and 14 as shown in Figure 6.1. The only reason we chose these locations was that they are 4 places apart. Later it would be easy to see that we could have chosen any 2 locations $(x, x + 4)$ for this purpose.

The first task is to execute $(1, 5)$. We do the rotate operation, denoted by r, a total of 6 times on the circuit, and so that bits arrive to positions shown in Figure 6.1(b). We now invoke the scan functionality so that in the next cycle bytes 1, 5 would be in positions 10 and 14 as shown in Figure 6.1(c). Note that in doing so, we effectively execute the

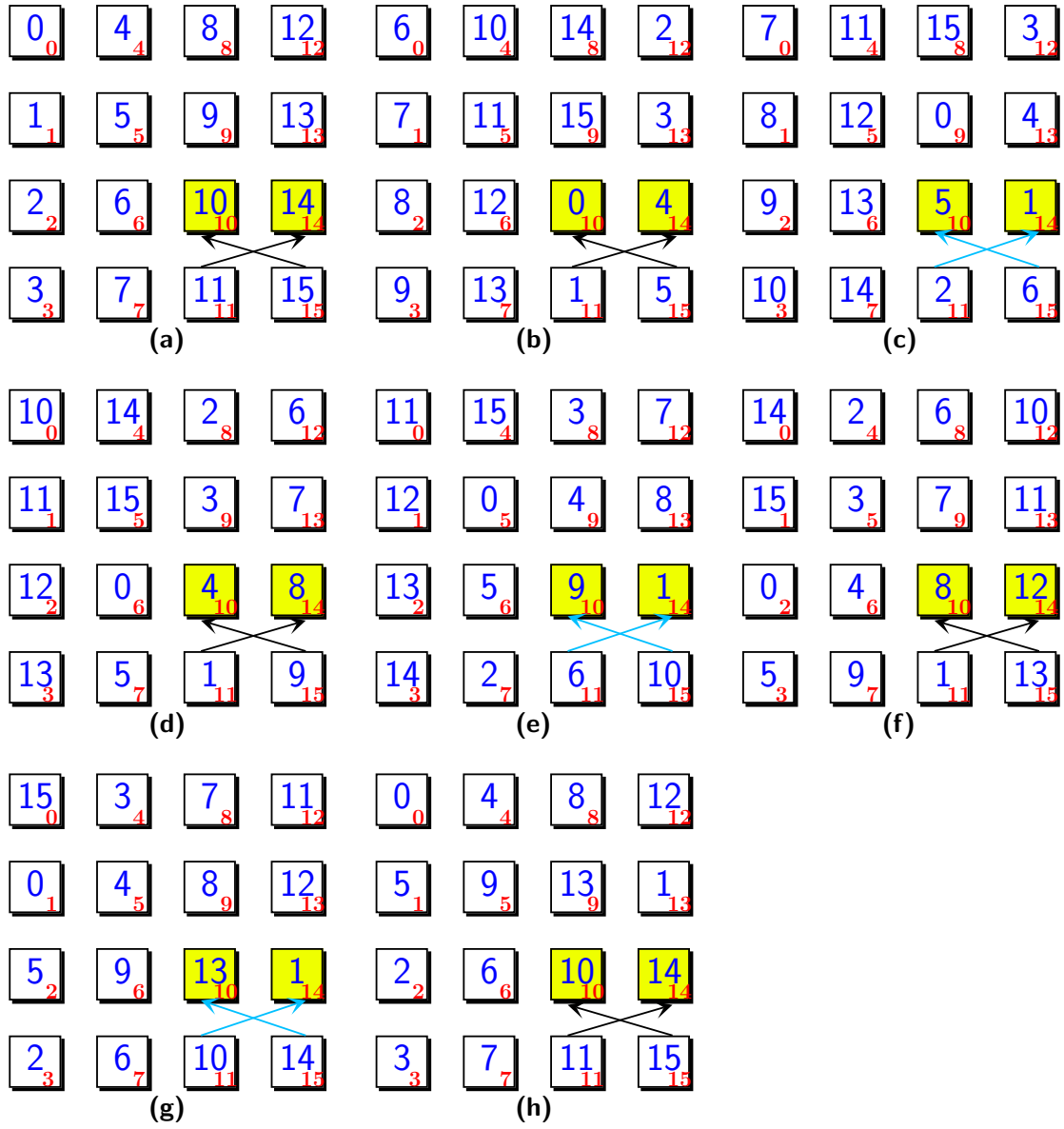

Figure 6.1 – The contents of pipeline (a) initially, (b) after r^6, (c) after $\theta \circ r^6$, (d) after $r^3 \circ \theta \circ r^6$, (e) after $\theta \circ r^3 \circ \theta \circ r^6$, (f) after $r^3 \circ \theta \circ r^3 \circ \theta \circ r^6$, (g) after $\theta \circ r^3 \circ \theta \circ r^3 \circ \theta \circ r^6$, (g) finally after $r \circ \theta \circ r^3 \circ \theta \circ r^3 \circ \theta \circ r^6$. Note the numbers in blue denote the byte index, i.e corresponds to b_i, and the subscripts in red denote the fixed register positions. As explained in Figure 2.4, the yellow boxes denote the byte registers implemented with scan flip-flops. Cyan and black arrows denote whether the operation θ or r is executed respectively.

permutation $\theta = r \circ (11, 15)$. The next swap to be executed is $(5, 9)$, which corresponds to $b_1 \leftrightarrow b_9$ in the current state. By rotating 3 more times, we reach to the state in Figure 6.1(d), where the bytes b_1, b_9 are in place to be swapped in the next cycle. After executing θ at this point, we reach to the state in Figure 6.1(e). The final swap to be performed is $(9, 13)$, which as per the previous logic is swapping bytes $b_1 \leftrightarrow b_{13}$. Again it is easy to see that performing $\theta \circ r^3$ over the next 4 cycles gives us the position in Figure 6.1(g), where all the bytes have now been swapped as required. We perform the rotate operation once more to get the position in Figure 6.1(h), where all bytes are back to the original position and the ShiftRows operation has been executed on the 1st row. In effect, the permutation we performed is $r \circ \theta \circ r^3 \circ \theta \circ r^3 \circ \theta \circ r^6$, which takes 16 cycles. Note that if we had chosen any other swap location of the form $(x, x + 4)$, it would still be possible to do the above sequence of operations. For example if we had chosen the swap $(9, 13)$ instead of $(11, 15)$, we would need to execute $\theta' \circ r^3 \circ \theta' \circ r^3 \circ \theta' \circ r^8$, where $\theta' = r \circ (9, 13)$. This already takes 17 cycles and so all the bytes will be indeed swapped correctly, but not return to their original positions as before. Conceptually, this means that if the AES round is executed in 16 cycles, then a few of the swap operations of the current round would take place in the next round, and we would have to tailor the other operations in the pipeline accordingly.

Following the same logic, let us now try to do the swaps $(2, 10) \circ (6, 14)$ of the next row. This time, let us choose two swap locations 8 places apart, in particular $(5, 13)$. The above swaps commute and so can be done in any order, so let us do $(2, 6)$ first. After r^{13}, the bytes b_2, b_6 are in place for swapping, and in the next cycle we execute the scan functionality to perform $\alpha = r \circ (5, 13)$. After 3 more cycles of r, the bytes b_6, b_{14} are in place, and then we execute α again. Thus by executing $\alpha \circ r^3 \circ \alpha \circ r^{13}$, we have again already spent 18 cycles. As explained before, this indicates that at this point, the bytes have again been correctly swapped in terms of their relative order in the pipeline and that in terms of data flow in the circuit, some of the swaps of the current AES round overflow into the next round.

The third set of swaps for the final row is $(11, 15) \circ (7, 15) \circ (3, 15)$. We can construct this sequence with 3 different swap locations also at distances 4, 8 and 12 apart. Let us choose the swaps $(11, 15), (5, 13)$ as before and $(2, 14)$ as the additional swap location. We have to execute $(3, 15)$ first, therefore we rotate once to bring the bytes b_3 and b_{15} in place and then execute $\beta = r \circ (2, 14)$. We will now use the swap locations $(11, 15), (5, 13)$, which have already been used to do swaps in the previous 2 rows. At this point b_7 and after the previous swap b_3 are already in place and so we execute α on the location $(5, 13)$ by invoking its scan functionality. For the last remaining swap $(11, 15)$, we have to wait till b_{11} returns to location 11, which requires 13 more rotations after which we can invoke θ.

Putting it together. We have just put together a set of swap sequences that enable the execution of the AES ShiftRows operation. We looked at each row separately and so it is conceivable that the swap sequences be performed one after the other, thereby

requiring a little over 48 cycles. But in the interest of latency, we wish to do them in 16 and if required within a few cycles of the next round. Since the k-cycles in each row that we executed commute with each other, the swaps can actually be executed concurrently. That is, following the above example, we

1. invoke scan functionality on the swap location $(2, 14)$ at clock cycle 1 (assuming we start with cycle 0);
2. invoke scan functionality on the swap location $(5, 13)$ at clock cycles 2, 13, 17;
3. invoke scan functionality on the swap location $(11, 15)$ at cycle 6, 10, 14, 16.

The point is that since the k-cycles commute, we execute the swaps concurrently on the given locations in 18 continuous cycles (numbered 0 to 17) and still achieve the ShiftRows functionality. Indeed it is a matter of a simple arithmetic exercise to see that the arrangement of bytes obtained after executing the above sequence of swaps concurrently in 18 cycles results in ShiftRows off by 2 extra rotations.

We have seen that we can execute AES ShiftRows and more generally any permutation of type 4, by judiciously choosing swap locations at distances 4, 8, 12 and tailoring the swap sequences around it. What about the other operations like SubBytes and MixColumns? That is where the engineering challenge lies. Since these are substitution type operations, they have to be accommodated in the pipeline preferably when the scan functionalities of the registers are not being invoked. There are of course precedence issues a designer would have to deal with, for example, the SubBytes and ShiftRows in any round must precede the MixColumns. Can this technique be applied to other block ciphers in general? For block ciphers that employ some kind of byte/nibble/word-based swap operations in their permutation function, the answer is affirmative. For example, SKINNY has a permutation function given by

$$(4, 5, 6, 7) \circ (9, 11) \circ (8, 10) \circ (12, 15, 14, 13)$$

This is a permutation of type 1, and has a similar form with AES, so it takes modest effort to construct it using swaps, in the same fashion explained above. For block ciphers such as GIFT that employ bit-based permutation function, the technique becomes slightly more involved.

From byte to bit-serial. When we reduce the datapath to 1 bit, we can no longer swap 2 bytes in one cycle and it would take exactly 8 cycles for every byte swap. At the bit level, ShiftRows of AES is essentially the composition of the following permutations over the set $[0, 127]$ for all $k \in [0, 7]$:

$$(8+k, 104+k, 72+k, 40+k) \circ (16+k, 80+k) \circ (48+k, 112+k) \circ (24+k, 56+k, 88+k, 120+k)$$

As it can be seen from this expansion, at the bit level, everything scales by a factor of 8. At the byte level, we used the sequence $r \circ \theta \circ r^3 \circ \theta \circ r^3 \circ \theta \circ r^6$ to execute 4-cycle $(1, 13, 9, 5)$ with the swap located at $(11, 15)$. At the bit level, let us choose the swap locations $(88, 120)$, located 32 places apart. Using the same logic as before, it is easy to see that $r^8 \circ \theta_1^8 \circ r^{24} \circ \theta_1^8 \circ r^{24} \circ \theta_1^8 \circ r^{48}$ can realize $\bigcup_{k=0}^{7}(8+k, 104+k, 72+k, 40+k)$, where $\theta_1 = (88, 120) \circ r$ (with $\bigcup$ denoting the composition operation). Similarly by choosing swap locations that are 64 and 96 places apart, we can permute the other rows using the same multiply-by-8 principle. Similarly the SKINNY permutations can be designed for the bit-serial datapath with swap locations 8, 16 and 24 places apart.

6.4 AES

For the rest of this section, we assume familiarity with the round function and the key scheduling algorithms of AES [NIS01], which is outlined in Section 2.2.1. Our circuit simply consists of the following components in the main hierarchy: (1) a state pipeline, (2) a key pipeline, (3) a controller, (4) a shared S-box.

6.4.1 State Pipeline

The state in our design uses the following components/techniques:

- nibble-level MixColumns circuit introduced by Jean et al. [JMPS17],
- the smallest known AES S-box "bonus" from Maximov and Ekdahl [ME19].

Given that state and key bits are stored in a pipelined fashion, one can easily notice that AddRoundKey can be performed without much hassle as long as each of the state and key pipelines produces the correct bit per clock cycle. Hence, the main challenges on the state pipeline part is to (1) execute all SubBytes, ShiftRows, MixColumns operations simultaneously, (2) complete the operations in 128 clock cycles, while (3) following the standard ordering of bits for the plaintext and the key. Below, we first describe each layer separately, and show how we can fuse them into one operation that executes over the state pipeline continuously.

6.4.2 ShiftRows with Swaps

Assume that the 128-bit pipeline is defined in the same fashion as in Section 2.3, i.e. the bits are loaded into FF_{127} and they are flushed out by FF_0. We use three swap operations to execute the ShiftRows layer: $(80, 112)$, $(56, 120)$ and $(25, 121)$. The timetable for scheduling these swaps are given in Table 6.5. Below, we explain how we came up with these swap sequences and the mechanism in which they work for shifting rows correctly.

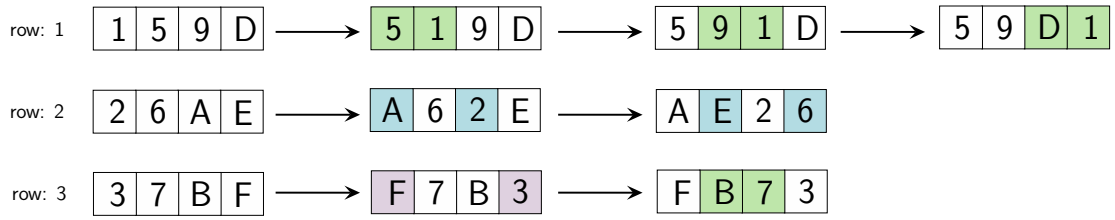

Figure 6.2 – The transition diagram for rows 1, 2, 3; where the colored cells denote the recently modified values. Note that there are three distinct swap operations, with distance 0, 1 and 2 cells in-between.

Table 6.5 – The timetable of operations for bit-serial AES encryption.

pipeline	operation	active cycles
state	swap (80, 112)	$[56, 64) \cup [88, 96) \cup [120, 127] \cup [8, 16)$
	swap (56, 120)	$[88, 96) \cup [120, 127] \cup [0, 8)$
	swap (25, 121)	$\{127\} \cup [0, 6]$
	load S-box	$\{8k + 7 : k \in [0, 15]\}$
	load Mix Col.	$[32, 40) \cup [64, 72) \cup [96, 104) \cup [0, 8)$
key	swap (96, 128)	$[0, 8)$
	swap (40, 72)	$[56, 64)$
	load S-box	$\{112\} \cup \{120\} \cup \{0\} \cup \{8\}$
	key XOR	$[0, 96)$
	add RC	(lookup table)

For simplicity, let us forget about the pipeline and shift operations for the moment, and focus on the nature of ShiftRows in the 16-byte state. We try to express ShiftRows in terms of byte swaps. Suppose that the values contained in the state are the hexadecimal characters $0, 1, \ldots, F$. Considering the standard byte arrangement for loading the initial data [NIS01], row 0 contains the values $0, 4, 8, C$; row 1 contains the values $1, 5, 9, D$ etc. We then devise a sequence of swap operations over the rows 1, 2, 3 to perform ShiftRows. Our three distinct swaps are denoted with distinct colors in Figure 6.2. This figure shows the movement of the bytes as they arrive to their final position implied by ShiftRows.

We point out two important observations: (1) each byte-swap operation can be executed by a bit-swap circuit through 8 consecutive calls interleaved by shift operations, (2) the swap operations denoted with the same color can actually be executed by a single swap operation as long as it is enabled in the correct clock cycle. Therefore, the choice of swaps and the timetable in Table 6.5 are straightforward extensions of this example into the 128-bit state pipeline.

To help understand how the structure helps perform the ShiftRows operation, we note that as the pipeline is always active, the shift operation is performed in every clock cycle. To additionally perform the swap $80 \leftrightarrow 112$, in any clock cycle, we need to place scan

flip-flops at locations 79 and 111 (and wire the output of 80 to the input of 111; wire the output of 112 to the input of 79) as is shown in Figure 6.3. So assuming that the bits indexed 0 to 127 enter the pipelines through the location 127, at clock cycle $k \leq 127$, the pipeline stores exactly k bits. For instance, in the 56-th clock cycle, the bits indexed 8, 40 are at locations 80, 112 respectively. Enabling swaps for cycles 56 to 63 therefore swaps bits $8, \ldots, 15$ with $40, \ldots, 47$, which are essentially bytes indexed by 1 and 5. It can be verified without difficulty that performing the same swap in cycles $[88, 96)$ actually swaps bytes 1 and 9. This exactly follows the explanation in Section 6.3 using $(80, 112)$ as swap locations instead of $(88, 120)$. Similarly the same swap in cycles $[120, 127)$ swaps bytes 1 with 13, which completes the ShiftRows operation on row 1. It is not too difficult to verify that the other swaps at cycles as listed in Table 6.2 faithfully perform the remaining ShiftRows operations.

6.4.3 The Nibble MixColumns

The nibble MixColumns was introduced by Jean et al. [JMPS17]. The multiplication over a single column is completed over 8 clock cycles, updating each nibble at a time. To simplify, we first represent a single column of bytes as 8 vertical nibble vectors as below. Namely, from the pipeline given in Figure 6.3, the vectors M_i are defined for $0 \leq i \leq 7$ as below:

$$M_i := \begin{bmatrix} \mathsf{FF}_i \\ \mathsf{FF}_{i+8} \\ \mathsf{FF}_{i+16} \\ \mathsf{FF}_{i+24} \end{bmatrix} \qquad \mathcal{R}(M_0) := \begin{bmatrix} \mathsf{FF}_8 \\ \mathsf{FF}_{16} \\ \mathsf{FF}_{24} \\ \mathsf{FF}_0 \end{bmatrix}$$

The nibble MixColumns architecture employs an additional set of 4 flip-flops to help with the serialized computation of this functionality. Define the vector M_8 to denote this additional internal 4-bit storage this architecture employs. During its 8 clock cycle operation, these flip-flops are used to keep the value of the leftmost bit of each one of the four bytes. We define a function *upward* rotation $\mathcal{R}$ that rotates the elements in a given vertical matrix by one position, as exemplified above. The circuit essentially performs the following sequence of operations to derive the new value of M_i for each $i = 0, 1, \ldots, 7$, starting from $i = 0$ respectively:

- if $i = 0$, store $M_8 \leftarrow M_0$ before any of the following computation,
- update $M_i \leftarrow \mathcal{R}(M_i) \oplus \mathcal{R}^2(M_i) \oplus \mathcal{R}^3(M_i) \oplus M_{i+1} \oplus \mathcal{R}(M_{i+1})$,
- if $i \in \{3, 4, 6\}$, further update $M_i \leftarrow M_i \oplus M_8 \oplus \mathcal{R}(M_8)$.

In other words, at each clock cycle, based on the internal 7-bit counter, we can execute a single slice of the previous computation. In total, it takes 8 clock cycles for a single

Figure 6.3 – The state (above) and key (below) pipelines of AES-128 encryption with colored scan flip-flops. S-box output ports are denoted with $\mathsf{S}_0||\mathsf{S}_1||\ldots||\mathsf{S}_7$.

column, and 32 clock cycles for the whole MixColumns layer. This serial circuit can be realized with 8 XOR, 8 NAND gates and 4 flip-flops (see Figure 1 of [JMPS17]).

6.4.4 Combined State Pipeline

In the controller, the circuit contains an 11-bit counter to keep both the round (4-bit) and the phase (7-bit). We split this counter into two parts and refer to them respectively by variables $0 \leq \mathsf{round} \leq 10$ for the upper 4-bit and $0 \leq \mathsf{count} \leq 127$ for the lower 7-bit.

In contrast to previous work [JMPS17], we follow the standard ordering of bits in our implementation. That is given a plaintext and a key, the bits are loaded into the circuit starting from the leftmost bits, and following the natural order [NIS01]. This becomes a crucial aspect of a block cipher implementation, if it is meant to be used in a mode of operation that needs to comply with a fixed standard.

At the beginning of its operation, the 11-bit counter is reset to zero. During initialization, i.e. $\mathsf{round} = 0$, the white-colored MUXes in Figure 6.3 are configured so that the next bit s of the state is received from the input port PT but after the XOR is performed with KEY, which is also being loaded at the same time. For $\mathsf{round} > 0$, we select the state bit to be loaded from the exit of the state pipeline.

SubBytes. Meanwhile, we proceed with executing the SubBytes layer, by enabling the S-box at every 8-th cycle. More precisely, the S-box is configured to take $\mathsf{FF}_{121}, \mathsf{FF}_{122}, \ldots, \mathsf{FF}_{127}$ and s as input, and the scan flip-flops $\mathsf{FF}_{120}, \ldots, \mathsf{FF}_{127}$ are instructed to load the output from the S-box if count mod $8 = 7$.

ShiftRows. Starting from count $= 56$, the swap operations become active. Many of the bits need to make a couple of jumps before they are located into their ultimate positions implied by ShiftRows, as demonstrated in Figure 6.2. Hence, position-wise, many bits are incorrectly located and look garbled as they pass through flip-flops $\mathsf{FF}_{24}, \ldots, \mathsf{FF}_{120}$. Nonetheless, as soon as they exit the last swap position FF_{24}, they are guaranteed to be in their final position. See Table 6.5 to notice that the last swap operation executed on a layer actually happens when count $= 15$ in the next round. In other words, performing ShiftRows over the i-th state uses the last 72 cycles of the round i and the first 16 cycles of the round $i + 1$, and it is not aligned with the counter round itself .

MixColumns. The input ports to the nibble MixColumns circuit are flip-flops FF_i for $i \in \{0, 1, 8, 9, 16, 17, 24, 25\}$, and the output ports are input to the exit MUX of the pipeline and $\mathsf{FF}_7, \mathsf{FF}_{15}, \mathsf{FF}_{23}$ respectively. The MixColumns of round i is performed at round $= i + 1$ and it is active during $0 \leq$ count mod $32 \leq 7$, except the last round where MixColumns must be skipped.

Resolving overlaps. Note that there are two clock cycles, i.e. count values, during which two operations modify the same FF simultaneously in Table 6.5. First, at clock cycle 127 both S-box and swap $(25, 121)$ attempts to overwrite FF_{120}. Here, the operation precedence is given to the S-box (as SubBytes comes before ShiftRows), meaning that the leftmost output bit of the S-box is fed to the swap operation (instead of FF_{120}). A second overlap occurs when count $= 3$, as MixColumns circuit attempts to read FF_{25} before its value is updated correctly by the swap $(25, 121)$. Here, the precedence is given to the swap operations, meaning that the output of the swap operation is fed as input to MixColumns circuit (instead of FF_{25}).

6.4.5 Key Pipeline

Suppose that $K_0, K_1, \ldots, K_{15}$ represent the key bytes of a particular round. Then the next round key sequence $K_{16}, \ldots, K_{31}$ is computed as follows:

$$\begin{bmatrix} K_{16} & K_{20} & K_{24} & K_{28} \\ K_{17} & K_{21} & K_{25} & K_{29} \\ K_{18} & K_{22} & K_{26} & K_{30} \\ K_{19} & K_{23} & K_{27} & K_{31} \end{bmatrix} \leftarrow \begin{bmatrix} K_0 & K_4 & K_8 & K_{12} \\ K_1 & K_5 & K_9 & K_{13} \\ K_2 & K_6 & K_{10} & K_{14} \\ K_3 & K_7 & K_{11} & K_{15} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S\text{-}box}(K_{13}) \oplus \mathsf{RC} & K_{16} & K_{20} & K_{24} \\ \mathsf{S\text{-}box}(K_{14}) & K_{17} & K_{21} & K_{25} \\ \mathsf{S\text{-}box}(K_{15}) & K_{18} & K_{22} & K_{26} \\ \mathsf{S\text{-}box}(K_{12}) & K_{19} & K_{23} & K_{27} \end{bmatrix}$$

where RC denotes the round constant byte.

In summary, the first column requires special treatment, because it involves S-box calls, and the remaining three columns can be updated smoothly (by simply XORing with a neighboring bytes). In particular, one can notice the disarrangement in the update of the first column, as it takes the current last columns bytes with a downward rotation (by one byte). If we implement this in a straightforward fashion by updating each byte when they arrive to position 0, we would have to choose the input of the S-box either from the position 13 (for computing K_{16}, K_{17}, K_{18}) or 9 (for computing K_{19}). This means that we would have to put an extra 8-bit MUX to choose which value needs to be fed to the S-box. Instead, we decided to temporarily move the byte K_{12} to position 13 before it is fed to S-box, and then return back to its original position after the S-box operation is done. Therefore, the pipeline performs the following operations in sequence:

- In the first 8 clock cycles, we activate the swap $(96, 128)$ so that the key byte K_{12} is temporarily moved such that it comes after K_{15}. Here, FF_{128} actually refers to the new key bit that is about to be loaded into the key pipeline. With this operation, the key pipeline contains $K_{13}, K_{14}, K_{15}, K_{12}$, in given order. Hence, it respects the order they are being used to update the first key column.
- In clock cycles 112, 120 (of the current round) and 0, 8 (of the next round); the S-box is used by the key pipeline. During these cycles, the S-box reads $K_{13}, K_{14}, K_{15}, K_{12}$ from $\mathsf{FF}_{120}, \ldots, \mathsf{FF}_{127}$ in given order. The output from the S-box is XORed with $\mathsf{FF}_{16}, \ldots, \mathsf{FF}_{23}$ and the result is loaded into $\mathsf{FF}_{15}, \ldots, \mathsf{FF}_{22}$.
- The round constant is added as the bit FF_{24} is loaded into FF_{23}. We use a lookup table to decide when the round constant bit is enabled. In total, this bit is enabled 16 times during the whole encryption.
- During the clock cycles $[56, 64)$, we activate the swap $(40, 72)$ to return K_{12} back to its original relative position. Hence the internal ordering of the bytes becomes K_{12}, K_{13}, K_{14}, K_{15} again.
- For the rest of the key bits, we handle the key scheduling by activating $\mathsf{FF}_{31} \leftarrow \mathsf{FF}_0 \oplus \mathsf{FF}_{32}$ during the clock cycles $[0, 96)$.

Table 6.6 tabulates the synthesis results for this AES circuit under 5 different standard cell libraries.

6.4.6 8-bit Datapath

As already stated, there are several implementations of AES with a byte-serial datapath that can execute one AES round in 21 cycles [MPL+11, BBR16a]. Since it is not possible to implement the circuit in less than 20 cycles if the number of S-boxes is limited to one, this represents a close-to-optimal latency for this datapath. However, note that

Table 6.6 – Synthesis figures for the AES-128 encryption-only circuits.

Library	Area (μm^2)	Area (GE)	Power (μW) @ 10 MHz	Latency (cycles) round	Latency (cycles) total	Energy (nJ/128-bit)	Throughput (Mbit/s)
1-bit							
STM 90 nm	5562.6	1267	73.6	128	1408	10.4	13.44
UMC 90 nm	5016.8	1600	65.2	128	1408	9.2	12.53
TSMC 90 nm	4692.2	1663	56.1	128	1408	7.9	14.50
NanGate 15 nm	441.8	2247	18.4	128	1408	2.6	293.89
NanGate 45 nm	1575.0	1974	143.0	128	1408	20.1	45.87
8-bit							
STM 90 nm	7838.0	1785	104.5	16	176	1.8	112.96
UMC 90 nm	6917.2	2206	85.6	16	176	1.5	115.60
TSMC 90 nm	6360.1	2256	68.9	16	176	1.2	121.89
NanGate 15 nm	564.2	2870	23.1	16	176	0.4	2160.68
NanGate 45 nm	2022.9	2535	192.4	16	176	3.4	376.94

these two circuits adopt a non-standard, row-first arrangement of bytes. One of our goals therefore was to design a circuit that uses standard byte ordering. As there already exists a 21-cycles-per-round circuit that achieves close to optimal latency, we did not attempt to design one that also achieves 20 cycles per round. Instead, we focus on an implementation that closely matches our bit-serial circuit, and achieves one round in 16 cycles, by using 2 S-box circuits.

As this circuit closely resembles the bit-serial circuit, all the calculations of swap locations and the time intervals when the swap functionality is invoked basically scale by a factor of 8. It is best to summarize it using the following salient points:

- The circuit has 32 byte-registers Reg_0 to Reg_{15} and Key_0 to Key_{15}, and we use the following swap operations to implement ShiftRows: **(a)** $(9, 13)$ in cycles 7, 11, 15, 0, **(b)** $(6, 14)$ in cycles 11, 15, 0, and **(c)** $(2, 14)$ in cycle 0.

- We use a $\{0,1\}^{32} \rightarrow \{0,1\}^{32}$ MixColumns circuit for this implementation. We chose to use the MixColumns implementation with 92 XOR gates from Maximov [Max19], which has the lowest gate count known in the literature. The operation is performed in cycles 0, 4, 8, 12. The inputs are taken from the byte registers in the first column and written in registers 1, 2, 3, 15 in the order from MSB to LSB. This closely resembles the bit-serial circuit.

- The key addition and S-box are done in every cycle.

- The key pipeline uses the swaps $(11, 15)$ in cycle 0 and $(4, 8)$ in cycle 7. The column addition in the key update is done by calculating $\mathsf{Key}_3 \leftarrow \mathsf{Key}_0 \oplus \mathsf{Key}_4$, in cycles 0 to 11.

Table 6.7 – The timetable of operations for bit-serial SKINNY-128-384 encryption circuit.

pipeline	operation	active cycles
state	swap (112, 120)	$[112, 120) \cup [120, 127] \cup [0, 8) \cup [64, 72)$
	swap (104, 120)	$[64, 72) \cup [88, 96) \cup [96, 104)$
	swap (96, 120)	$[64, 72)$
	load S-box	$\{8k : k \in [0, 15]\}$
	rc addition.	(lookup table + LFSR)
	load Mix Col.	$[0, 32)$
tweakey 1,2,3	swap (56, 120)	$[72, 127] \cup [0, 8)$
	swap (48, 56)	$[120, 127]$
	swap (24, 56)	$[112, 120) \cup [120, 127] \cup [0, 8)$
	swap (8, 24)	$[120, 127] \cup [0, 8) \cup [24, 32)$
tweakey 2	swap (0, 1)	$[0, 6] \cup [8, 14] \cup [16, 22] \cup [24, 30] \cup [32, 38] \cup [40, 46] \cup [48, 54] \cup [56, 62]$
	LFSR XOR	$\{8k : k \in [0, 7]\}$
tweakey 3	LFSR (8-bit)	$\{8k : k \in [0, 7]\}$

Table 6.6 tabulates the synthesis results for the 8-bit circuit for the same 5 different standard cell libraries.

6.5 SKINNY

SKINNY provides six different variants [BJK+16]. In this thesis, we consider the variants that are used by NIST LWC candidates, i.e. these are the members with 128-bit block size, as given in Table 6.1. In these variants, the tweakey size is variable, i.e. it can consist of $128z$ bits for $z = 1, 2, 3$. These three versions are SKINNY-128-128, SKINNY-128-256 and SKINNY-128-384 respectively.

From circuit designer perspective, SKINNY is quite similar to AES, but it employs more lightweight operations for the round function. Prominently, S-box and MixColumns can be realized with much smaller circuitry compared to AES (see Appendix A.1.3). The round function consists of SubCells, AddConstants, AddRoundTweakey, ShiftRows, MixColumns. For the finer details of these layers, we refer the reader to Section 2.2.2.

Our design follows a similar architecture to that of AES. The circuit simply consists of the following parts in the main hierarchy: (1) a state pipeline (which includes a dedicated S-box), (2) a key pipeline, (3) a controller. Below, we will explain the 1-bit implementation, and modifying the circuit into 8-bit implementation is quite straightforward.

6.5.1 Combined State Pipeline

In the controller, the circuit contains a 13-bit counter to keep both the round (6-bit) and the phase (7-bit). We split this counter into two parts and refer to them respectively by

variables $0 \leq \mathsf{round} \leq 56$ for the upper 6 bits and $0 \leq \mathsf{count} \leq 127$ for the lower 7 bits.

Because SKINNY is already designed with hardware-friendliness in mind, we load the bits into the circuit starting from the leftmost bits, by following the standard [BJK+16]. In our implementations the key blocks and the plaintext are loaded simultaneously and completed in 128 cycles. This applies to all three versions of SKINNY-128-128, SKINNY-128-256, SKINNY-128-384.

At the beginning of its operation, the 13-bit counter is reset to zero. Then during initialization, i.e. $\mathsf{round} = 0$, the plaintext is loaded through 1-bit input port, and the key is loaded through z-bit input port into their respective pipelines without modification. Each tweakey block has its own dedicated input port. These ports are denoted with PT (for plaintext) and KEY1, KEY2, KEY3 for the tweakey. Below, we describe the layers of operations executed on the state pipeline, in an order observed by the incoming bits. The high-level view of the circuit is given in Figure 6.4.

SubCells. SubCells layer is executed by enabling the S-box at every 8-th cycle. More precisely, the S-box is configured to read $\mathsf{FF}_{120}, \mathsf{FF}_{121}, \ldots, \mathsf{FF}_{127}$ as input, and the scan flip-flops $\mathsf{FF}_{119}, \ldots, \mathsf{FF}_{126}$ are instructed to be loaded with the S-box output if $\mathsf{count} \bmod 8 = 0$.

AddConstants. The round constants are added right after the S-box operation. An XOR gate is placed between FF_{119} and FF_{120}, and the round constant bit rc is added. We use a 7-bit LFSR circuit (not shown in the figure) to produce the round constant bit.

AddRoundTweakey. The key bits are added at the same position with the round constant bit, i.e. between FF_{119} and FF_{120}. In order to synchronize this with the key pipeline, the key bits k_0, k_1, k_2 are read from FF_{120} of the key pipeline. The key addition is active during $8 \leq \mathsf{count} < 72$. This corresponds to adding the first half of each tweakey.

ShiftRows. This layer is executed with 3 swap operations, similar to AES, and the timetable of swaps is given in Table 6.7. Position-wise, bits are incorrectly located and look garbled as they pass through flip-flops $\mathsf{FF}_{95}, \ldots, \mathsf{FF}_{119}$, but as soon as they exit the last swap position FF_{95}, they are guaranteed to be in their intended final position.

MixColumns. The input ports to the nibble MixColumns circuit are flip-flops FF_i for $i \in \{0, 32, 64, 96\}$, and the output ports are input to the exit MUX of the pipeline and $\mathsf{FF}_{31}, \mathsf{FF}_{63}, \mathsf{FF}_{95}$ respectively. The MixColumns operation is active during the first 32 clock cycles of each round.

Resolving overlaps. Note that during clock cycles $64 \leq \mathsf{count} < 72$ three swaps $(112, 120)$, $(104, 120)$, $(96, 120)$ are active at the same time and overlap at the same

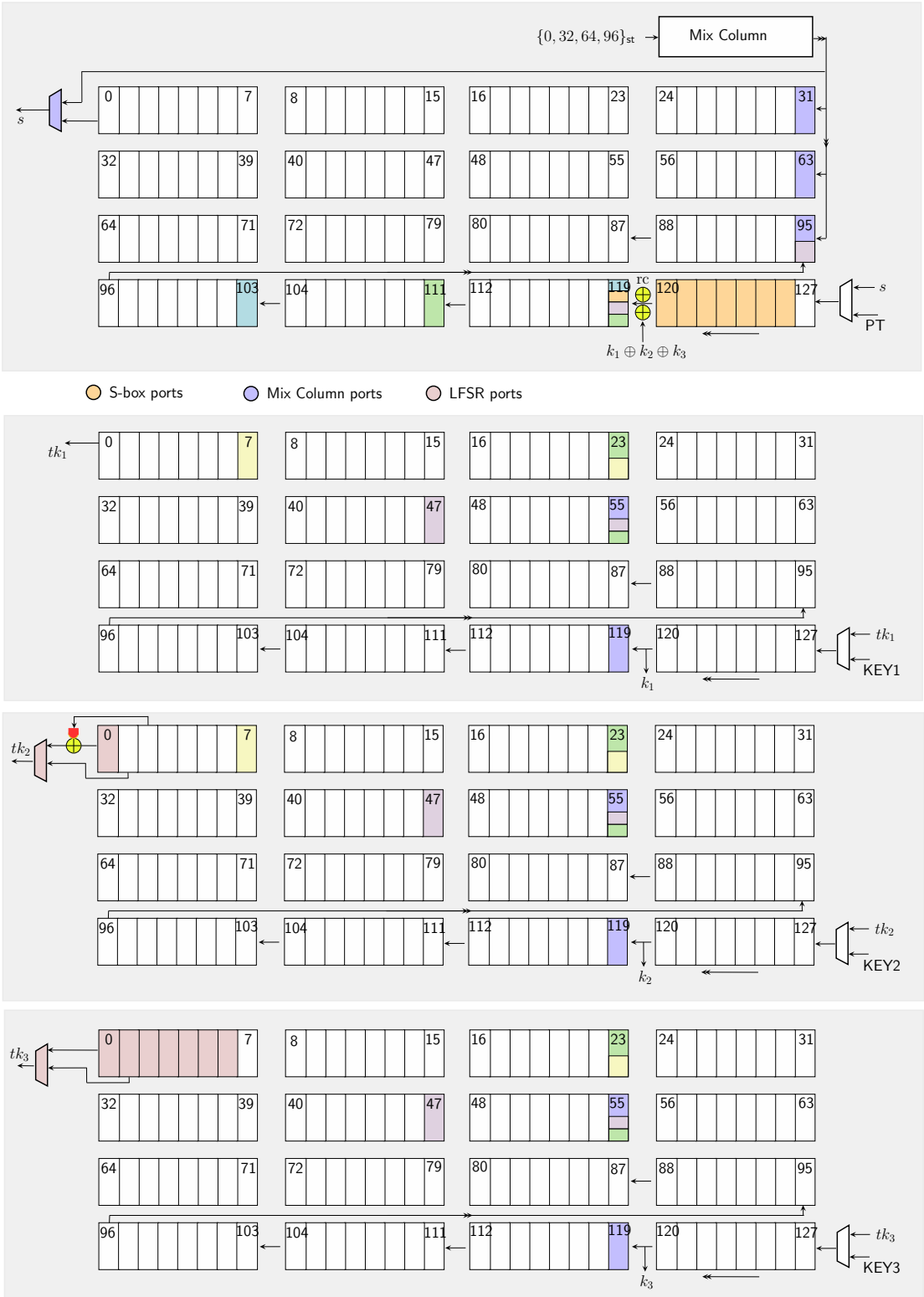

Figure 6.4 – The state (above) and key ($\mathcal{TK}1$, $\mathcal{TK}2$ and $\mathcal{TK}3$ respectively) pipelines of SKINNY encryption.

flip-flop FF_{120}. The order of execution here is $(96, 120)$, $(104, 120)$ and $(112, 120)$, in given order.

6.5.2 Key Pipeline

SKINNY can have up to three blocks of tweakey, referred to as $\mathcal{TK}1$, $\mathcal{TK}2$, $\mathcal{TK}3$ [BJK+16]. The key schedule algorithm is quite similar in all three key blocks. More precisely, suppose that $K_0, K_1, \ldots, K_{15}$ represent the key bytes of a particular tweakey block. Then the next round key sequence $K_{16}, \ldots, K_{31}$ is computed as follows:

$$\begin{bmatrix} K_{16} & K_{17} & K_{18} & K_{19} \\ K_{20} & K_{21} & K_{22} & K_{23} \\ K_{24} & K_{25} & K_{26} & K_{27} \\ K_{28} & K_{29} & K_{30} & K_{31} \end{bmatrix} \leftarrow \begin{bmatrix} \mathcal{L}_i(K_9) & \mathcal{L}_i(K_{15}) & \mathcal{L}_i(K_8) & \mathcal{L}_i(K_{13}) \\ \mathcal{L}_i(K_{10}) & \mathcal{L}_i(K_{14}) & \mathcal{L}_i(K_{12}) & \mathcal{L}_i(K_{11}) \\ K_0 & K_1 & K_2 & K_3 \\ K_4 & K_5 & K_6 & K_7 \end{bmatrix}$$

where the operation $\mathcal{L}_i$ are 8-bit permutations given below:

$$\begin{aligned} \mathcal{L}_1(x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7) &:= x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7 \\ \mathcal{L}_2(x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7) &:= x_1||x_2||x_3||x_4||x_5||x_6||x_7||(x_0 \oplus x_2) \\ \mathcal{L}_3(x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7) &:= (x_1 \oplus x_7)||x_0||x_1||x_2||x_3||x_4||x_5||x_6 \end{aligned}$$

Therefore, our key pipelines do the following operations in sequence. First, we swap the first and the last eight bytes by using the swap $(56, 120)$. Then we perform the local byte permutations on the upper half (i.e. the first 8 bytes) of the key through swaps $(48, 56)$, $(24, 56)$, $(8, 24)$. Finally we apply the 8-bit permutation $\mathcal{L}_2$ through another swap $(0, 1)$ for $\mathcal{TK}2$, and use a dedicated 8-bit LFSR circuit for $\mathcal{L}_3$ in $\mathcal{TK}3$.

6.5.3 8-bit

The 8-bit implementation is in fact simpler than 1-bit, because circuitry such as LFSR, S-box are already compatible with the data path size. We only need to add extra gates for swaps, e.g. extend each single swap into byte swap, and duplicate circuit for MixColumns. The timetable is also updated so that each consecutive activity in 8 clock cycles are squeezed into one clock cycle. Table 6.8 tabulates the synthesis results for the 1/8-bit circuits for 5 different standard cell libraries.

6.6 GIFT*

We will be focusing our efforts on the bit-sliced design of the GIFT block cipher, as used in the NIST LWC candidates GIFT-COFB and SUNDAE-GIFT [BCI+19, BBP+19]. We denote it by GIFT* as it differs from the original construction in the way data bits are organized (the implementation of the regular GIFT circuit is given in Section 6.7). In this

Table 6.8 – Synthesis figures for the SKINNY encryption-only circuits.

Library	Area (μm^2)	Area (GE)	Power (μW) @ 10 MHz	Latency (cycles) round	Latency (cycles) total	Energy (nJ/128-bit)	Throughput (Mbit/s)
SKINNY-128-128 1-bit							
STM 90 nm	4697.7	1070	51.47	128	5248	27.0	12.51
UMC 90 nm	4249.3	1355	52.08	128	5248	27.3	10.72
TSMC 90 nm	4022.6	1425	47.12	128	5248	24.7	14.36
NanGate 15 nm	391.7	1992	15.89	128	5248	8.3	258.45
NanGate 45 nm	1394.9	1748	122.06	128	5248	64.1	38.77
SKINNY-128-128 8-bit							
STM 90 nm	5820.6	1326	62.44	16	656	4.1	71.02
UMC 90 nm	5233.2	1669	61.38	16	656	4.0	58.52
TSMC 90 nm	4812.2	1706	51.67	16	656	3.4	72.69
NanGate 15 nm	453.1	2304	18.62	16	656	1.2	979.38
NanGate 45 nm	1617.8	2022	146.17	16	656	9.6	209.08
SKINNY-128-256 1-bit							
STM 90 nm	6642.7	1513	75.30	128	6272	47.2	11.93
UMC 90 nm	6043.9	1927	75.70	128	6272	47.5	10.52
TSMC 90 nm	5730.9	2030	69.25	128	6272	43.4	14.36
NanGate 15 nm	561.0	2853	22.99	128	6272	14.4	232.60
NanGate 45 nm	1996.9	2502	175.28	128	6272	109.9	38.13
SKINNY-128-256 8-bit							
STM 90 nm	8252.9	1880	90.47	16	784	7.1	59.64
UMC 90 nm	7463.7	2380	88.64	16	784	6.9	43.27
TSMC 90 nm	6864.1	2434	75.21	16	784	5.9	60.42
NanGate 15 nm	658.5	3350	27.19	16	784	2.1	1033.79
NanGate 45 nm	2338.7	2923	211.66	16	784	16.6	166.14
SKINNY-128-384 1-bit							
STM 90 nm	8631.5	1966	99.36	128	7296	72.5	8.25
UMC 90 nm	7895.7	2518	99.73	128	7296	72.8	9.94
TSMC 90 nm	7465.2	2645	91.38	128	7296	66.7	14.82
NanGate 15 nm	733.7	3732	30.22	128	7296	22.0	211.46
NanGate 45 nm	2603.6	3263	229.10	128	7296	167.2	38.77
SKINNY-128-384 8-bit							
STM 90 nm	10674.2	2431	119.60	16	912	10.9	50.84
UMC 90 nm	9670.6	3084	116.40	16	912	10.6	38.61
TSMC 90 nm	8896.9	3155	99.29	16	912	9.1	68.41
NanGate 15 nm	859.5	4372	35.72	16	912	3.3	744.33
NanGate 45 nm	3060.3	3825	277.45	16	912	25.3	143.14

variant, the cipher state is reordered and interpreted as a two-dimensional array, i.e. four 32-bit segments S_0, S_1, S_2, S_3 such that

$$\begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} s_3 & s_7 & \dots & s_{127} \\ s_2 & s_6 & \dots & s_{126} \\ s_1 & s_5 & \dots & s_{125} \\ s_0 & s_4 & \dots & s_{124} \end{bmatrix}$$

where $s_0 s_1 \dots s_{127}$ are the state bits. Further details are given in Section 2.2.5.

6.6.1 1-bit Datapath

In this section, we present our 1-bit swap-and-rotated GIFT* architecture in which each round function computation is performed in exactly 128 cycles.

State Pipeline

The bit-wise nature of both the GIFT* and GIFT permutation complicates matters in a swap-and-rotate setting, since each state bit needs to be moved to its designated position individually. As a consequence, a simple solution with few swaps as devised for the AES ShiftRows procedure, detailed in Section 6.4.2 is not achievable.

Nevertheless, the GIFT* permutation can be partitioned into three layers each can be generated with three separate swaps, thus, in total, we allocate nine swaps.

1. $(\mathsf{FF}_{31}, \mathsf{FF}_{30})$, $(\mathsf{FF}_{31}, \mathsf{FF}_{28})$, $(\mathsf{FF}_{31}, \mathsf{FF}_{29})$.
2. $(\mathsf{FF}_{28}, \mathsf{FF}_{24})$, $(\mathsf{FF}_{28}, \mathsf{FF}_{26})$, $(\mathsf{FF}_{28}, \mathsf{FF}_{26})$.
3. $(\mathsf{FF}_{22}, \mathsf{FF}_{4})$, $(\mathsf{FF}_{22}, \mathsf{FF}_{10})$, $(\mathsf{FF}_{22}, \mathsf{FF}_{16})$.

Due to the column-wise application of the substitution layer in GIFT*, the S-box ports in the state pipeline are FF_{31}, FF_{63}, FF_{95} and FF_{127} which are active during the cycles 96 to 127. A graphical depiction of the GIFT* state pipeline is given in Figure 6.5.

Key Pipeline

The bit-sliced interpretation of GIFT* significantly simplifies how the 64-bit round keys are extracted in each round since they are now mixed into a continuous stretch of the cipher state. For this we can assume, without loss of generality, that the master key K is

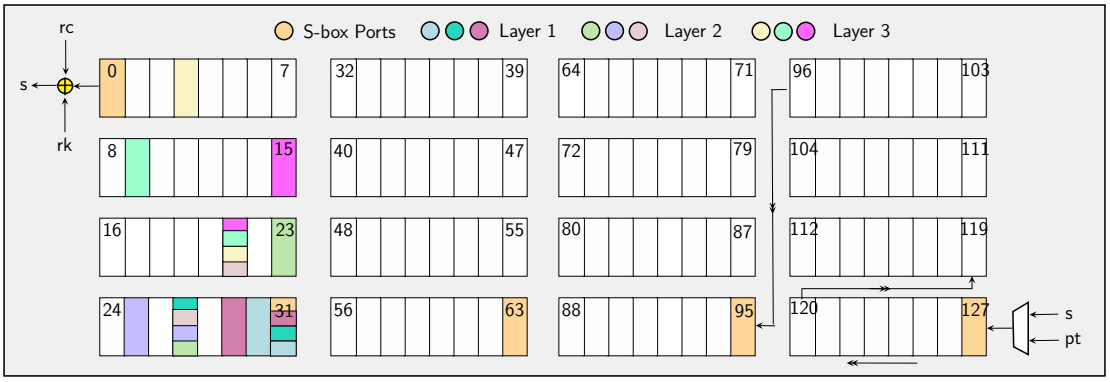

Figure 6.5 – 128-cycle, bit-serial GIFT* round function implementation using nine swaps.

loaded in the following order as to simplify the swapping algorithm.

$$K = \begin{bmatrix} K_0 \,||\, K_1 \\ K_6 \,||\, K_7 \\ K_2 \,||\, K_3 \\ K_4 \,||\, K_5 \end{bmatrix},$$

In this scenario, the 64-bit round keys $K_2||K_3||K_6||K_7$ are added to the block cipher states during the cycles 32 to 96.

The swap sequence for the GIFT* key schedule is partitioned into four phases.

Phase 1 (Rotating the state). We rotate the entire key state by 64 positions to the left. This operation can be achieved with a single swap during 64 active cycles. Preferably, the transformation should occur concurrently with the addition of the round key into the cipher state, i.e. we allocate FF_0 and FF_{64} to perform the rotation during the cycles 32 to 96.

Phase 2 (Swapping the precedence). To achieve a full emulation of the 96-bit rightward rotation of the key schedule, it is further necessary to swap the precedence of the utilized round key halves, i.e. $K_2||K_3$ and $K_6||K_7$. This again only requires a single swap during 32 cycles and can be performed subsequently to the first phase, hence we allocate FF_0 and FF_{96} for this second phase.

Phase 3 (Rotating K_6). This transformation can been seen as a 14-bit leftward rotation that can be achieved by composing three leftward rotations of magnitude 8, 4, and 2. The position and the interval of those three swaps can be chosen relatively freely, as $K_6||K_7$ is not a part of the current round key, as long as they occur after the second phase has terminated. To simplify the matter, we chose to perform them back-to-back during the cycles 32 and 66. More concretely, the 4-bit rotation is done during the cycles 32 to

Table 6.9 – The timetable of operations for bit-serial GIFT* encryption.

pipeline	operation	active cycles
state	swap (31, 30)	$\{8k+7 : k \in [0,15]\}$
	swap (31, 28)	$\{8k+5 : k \in [0,15]\} \cup \{8k+7 : k \in [0,15]\}$
	swap (31, 29)	$\{8k+5 : k \in [0,15]\}$
	swap (28, 24)	$\{0,1,42,43,50,51,58,59,66,67,104,105,112,113,120,121\}$
	swap (28, 26)	$\{6,7,10,11,14,15,18,19,22,23,26,27,30,31\}$ $\cup\{34,35,72,73,80,81,88,89,96,97\}$
	swap (28, 22)	$\{74,75,82,83,90,91,98,99\}$
	swap (22, 4)	$\{2,3,34,35,66,67,98,99\}$
	swap (22, 10)	$\{4,5,26,27,36,37,58,59,68,69,90,91,100,101,122,123\}$
	swap (22, 16)	$\{6,7,18,19,28,29,38,39,50,51,60,61,70,71\}$ $\cup\{82,83,92,93,102,103,114,115,124,125\}$
	key addition	$[32,96)$
	rc addition	(lookup table)
	load S-box	$[96,128)$
key	swap (64, 128)	$[32,96)$
	swap (32, 128)	$[96,128]$
	swap (96, 100)	$[32,44)$
	swap (84, 92)	$[44,52)$
	swap (76, 78)	$[52,66)$
	swap (96, 100)	$[48,60)$

44 using a swap at register FF_{95} and FF_{99}. Subsequently, we perform the 8-bit rotation during cycles 44 to 52 with the registers FF_{83} and FF_{91}, followed by the 2-bit rotation during cycles 52 to 66 using the registers FF_{75} and FF_{77}.

Phase 4 (Rotating K_7). Phase 3 is followed by a 4-bit leftward rotation of K_7 that is congruent to the 12-bit rightward rotation of the specification. This necessitates a single swap of size 4 for which we can reuse the same swap as utilized in phase 3, i.e. FF_{99} and FF_{95} during the cycles 48 to 60.

A summary of both the key schedule and round function swaps is tabulated in Table 6.9.

6.6.2 4-Bit Datapath

Analogous to the bit-serial implementation presented in the previous section, we now describe the 4-bit-serial architecture that completes execution of a round in 32 clock cycles.

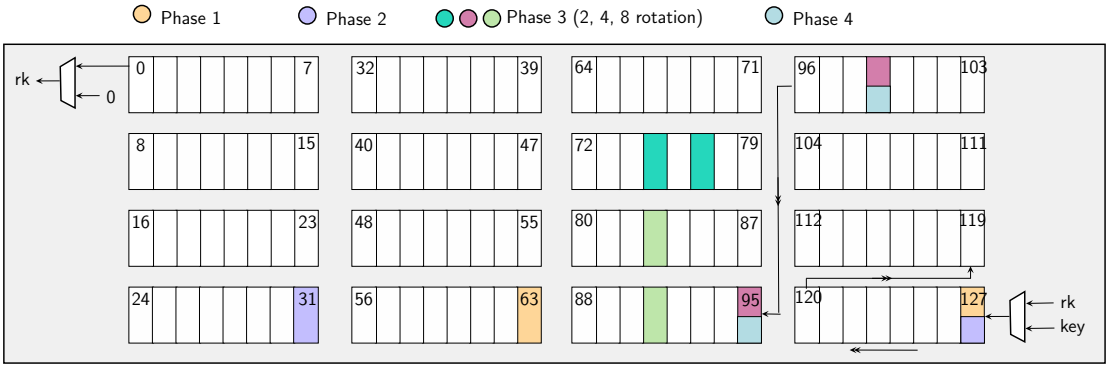

Figure 6.6 – 128-cycle, bit-serial GIFT* key schedule implementation using five swaps.

State Pipeline

The 4-bit state pipeline is unlikely to be achieved by simple swaps and a concurrent rotation as the substitution layer overwrites 4 non-adjacent bits FF_{31}, FF_{63}, FF_{95}, FF_{127} (see Figure 6.5). Then it follows that the swaps performing the permutation must necessarily be placed in the most significant quarter of the state pipeline and each 32-bit row of the state has to be permuted in only 8 cycles. Furthermore since we employed 9 swaps, i.e. 18 scan flip-flops, in the bit-serial construct, we need at the least four times this amount in the 4-bit case . This makes at least 72 MUXed flip-flops which significantly complicates the placement of swaps.

A second difficulty arises due to the fact that the pipeline rotates four positions at a time, thus the S-box taps are not constant but move further down the pipeline with every clock cycle, requiring a significant number of multiplexers to differentiate the different taps.

In order to circumvent those complexities, we chose to equip the entire 128-bit state with scan flip-flops and execute the permutation in the last cycle of the round while using 4 S-boxes in parallel to substitute 16 bits of the state during the cycles 24 to 31. Note that a single GIFT* S-box can be synthesized in fewer than 20 GE, thus the overhead of using four units is marginal and possibly still smaller than using multiplexers for the moving S-box taps.

Key Pipeline

The 4-bit key pipeline can be seamlessly adapted from the 1-bit counterpart by simply turning the single-bit swaps into nibble swaps, following the generic technique from Section 6.3. As we had 5 swaps in the single-bit version, we now have $4 \times 5 = 20$ swaps, i.e. 40 scan flip-flops. In Table 6.10, we list the synthesis results for our 1-bit and 4-bit GIFT* circuits.

In addition to these encryption circuits, the decryption circuits for all these block ciphers

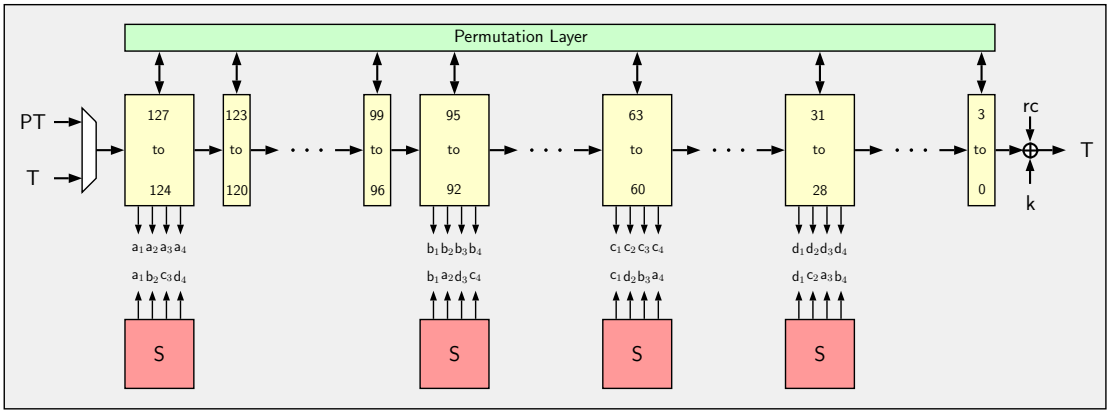

Figure 6.7 – 4-bit GIFT* state pipeline. Registers are marked in yellow.

Table 6.10 – Synthesis figures for 1-bit and 4-bit GIFT* encryption-only circuits.

Library	Area (μm^2)	Area (GE)	Power (μW) @ 10 MHz	Latency (cycles) round	Latency (cycles) total	Energy (nJ/128-bit)	Throughput (Mbit/s)
			1-bit				
STM 90 nm	4863.5	1108	48.7	128	5248	25.5	9.09
UMC 90 nm	4410.8	1332	49.8	128	5248	26.1	9.77
TSMC 90 nm	4176.5	1480	45.1	128	5248	23.7	12.51
NanGate 15 nm	402.3	2047	15.4	128	5248	8.1	178.92
NanGate 45 nm	1432.1	1791	122.3	128	5248	64.2	29.82
			4-bit				
STM 90 nm	6280.5	1430	61.4	32	1312	5.1	35.92
UMC 90 nm	5779.7	1779	60.9	32	1312	4.4	30.91
TSMC 90 nm	5135.6	1819	50.8	32	1312	4.3	38.93
NanGate 15 nm	481.5	2449	17.1	32	1312	1.9	664.60
NanGate 45 nm	1704.5	2130	152.9	32	1312	13.9	114.87

are also easy to construct. Most of the AEAD modes we implement are inverse-free, therefore encryption-only circuits are sufficient for these candidates. However for a complete discussion, we present some ideas on decryption in Section 6.9.

6.7 GIFT

The regular GIFT specification is significantly harder to transform into a low-latency swap-and-rotate circuit due to the fact that the round key bits are not added to cipher state in a continuous stretch. Namely, if $U = K_5||K_4$ and $V = K_1||K_0$ represent the 64-bit round key, then its individual bits are mixed into the state S as follows,

$$s_{4i+2} = s_{4i+2} \oplus u_i,\ s_{4i+1} = s_{4i+1} \oplus v_i,\ \forall i \in \{0, \ldots, 31\}$$

Table 6.11 – Synthesis figures for the bit-serial GIFT encryption circuit.

Library	Area		Power (μW)	Latency (cycles)		Energy	Throughput
	(μm^2)	(GE)	@ 10 MHz	round	total	(nJ)	(Mbit/s)
STM 90 nm	5334.3	1215	51.3	128	5248	26.9	7.35
UMC 90 nm	4801.2	1531	51.8	128	5248	27.2	6.32
TSMC 90 nm	4507.3	1597	45.9	128	5248	24.1	8.61
NanGate 15 nm	430.5	2190	16.1	128	5248	8.4	146.92
NanGate 45 nm	1528.4	1915	131.8	128	5248	69.2	23.34

By reordering the key bits in a manner such that the bits of U and V exit the pipeline during the correct cycles, we can reuse the rotation techniques to obtain a key schedule with 6 different swaps. On the other hand, we can recall the intuition for the state pipeline from Section 6.6.1, in order to generate the swap sequence for the GIFT round function. The summary of all GIFT key schedule and round function swaps are tabulated in Table 6.12.

6.8 AEAD Implementations

As standalone block ciphers are not ready-to-use primitives, they are usually wrapped in a mode of operation. In this section, we investigate four NIST LWC candidates from the second round which are bootstrapped via the improved 1-bit (and 4/8-bit) implementations of AES, SKINNY and GIFT* presented in the previous sections. Namely, these candidates are SUNDAE-GIFT, SAEAES, Romulus and SKINNY-AEAD [BBP+19, NMMaS+19, IKMP19, BJK+19]. For all four schemes, we report the hitherto smallest block-cipher-based authenticated encryption circuits in the literature.

The choice of these four particular candidates in our work is influenced by the observation that the area of a block cipher is determined, to a large extent, by the amount of storage elements, rather than how lightweight the round operations are. This is more evident when one compares SKINNY-128-384, whose round function comprises lightweight operations, to AES, whose S-box and MixColumns circuits are significantly larger. The former is much larger, only because it requires large number of flip-flops to store the key.

Because an authenticated encryption scheme produces a tag besides the ciphertext blocks, it is natural to expect a particular value that is initialized at the beginning and updated repetitively after processing each new block of data. We refer to this value as *the running state*. The running state is eventually used to compute the tag, so that all blocks contribute to its value. From the area perspective, an important question is whether storing the running state requires an extra register or not. For the chosen candidates, the running state is actually not a separate value, but rather it is passed between consecutive encryption calls. In other words, we can use the state register inside the block cipher to

Table 6.12 – The timetable of operations for bit-serial GIFT encryption.

pipeline	operation	active cycles
state	swap (39, 71)	$[0, 8) \cup [8, 16) \cup [16, 24) \cup [121, 128)$
	swap (38, 22)	$[0, 9] \cup [58, 73] \cup [122, 127]$
	swap (98, 110)	$[10, 13] \cup [34, 37] \cup [54, 57] \cup [74, 77] \cup [98, 101] \cup [118, 121]$
	swap (109, 85)	$[7, 10] \cup [51, 54] \cup [71, 74] \cup [115, 118]$
	swap (108, 72)	$[4, 7] \cup [68, 71]$ $\cup\{34, 35, 72, 73, 80, 81, 88, 89, 96, 97\}$
	swap (121, 117)	$\{16k + 5 : k \in [0, 15]\} \cup \{16k + 13 : k \in [0, 15]\} \cup \{16k + 15 : k \in [0, 15]\}$
	swap (122, 114)	$\{16k + 1 : k \in [0, 15]\} \cup \{16k + 3 : k \in [0, 15]\}$
	swap (123, 111)	$\{16k + 1 : k \in [0, 15]\}$
	swap (22, 4)	$\{2, 3, 34, 35, 66, 67, 98, 99\}$
	swap (22, 10)	$\{4, 5, 26, 27, 36, 37, 58, 59, 68, 69, 90, 91, 100, 101, 122, 123\}$
	swap (22, 16)	$\{6, 7, 18, 19, 28, 29, 38, 39, 50, 51, 60, 61, 70, 71\}$ $\cup\{82, 83, 92, 93, 102, 103, 114, 115, 124, 125\}$
	key addition	$\{4k + 1 : k \in [0, 31]\} \cup \{4k + 2 : k \in [0, 31]\}$
	rc addition	(lookup table)
	load S-box	$\{4k + 3 : k \in [0, 31]\}$
key	swap (120, 128)	$\{4k : k \in [3, 16]\}$ if round mod $4 = 0$ $\{4k - 1 : k \in [3, 16]\}$ if round mod $4 = 1$ $\{4k + 1 : k \in [3, 16]\}$ if round mod $4 = 2$ $\{4k - 2 : k \in [3, 16]\}$ if round mod $4 = 3$
	swap (112, 128)	$\{1\} \cup \{4k - 2 : k \in [5, 16] \cup [21, 32]\}$ if round mod $4 = 0$ $\{4k : k \in [5, 16] \cup [21, 31]\}$ if round mod $4 = 1$ $\{1\} \cup \{4k - 1 : k \in [5, 16] \cup [21, 32]\}$ if round mod $4 = 2$ $\{4k + 1 : k \in [5, 16] \cup [21, 31]\}$ if round mod $4 = 3$
	swap (1, 128)	$\{4k : k \in [1, 31]\}$ if round mod $4 = 0$ $\{0\}$ if round mod $4 = 1$ $\{4k + 2 : k \in [0, 31]\}$ if round mod $4 = 3$
	swap (1, 33)	$\{4k + 1 : k \in [0, 7]\}$ if round mod $4 = 0$ $\{4k + 3 : k \in [0, 7]\}$ if round mod $4 = 1$ $\{4k + 2 : k \in [0, 7]\}$ if round mod $4 = 2$ $\{4k : k \in [1, 8]\}$ if round mod $4 = 3$
	swap (4, 5)	$\{4k : k \in [0, 31]\}$ if round mod $4 = 0$
	swap (2, 3)	$\{4k : k \in [0, 31]\}$ if round mod $4 = 3$

keep this value temporarily until the next encryption starts. It is precisely the reduction in the storage area that yields the impressive area results for the four candidates.

In the special case of Romulus, which actually defines six different variants, we decided to implement two members, the primary member N1 and its sibling N3 that is likely to cost the smallest area in ASIC circuit. Romulus-N1 is larger than Romulus-N3, because the latter favors the smaller SKINNY-128-256, while its other nonce-based siblings all use SKINNY-128-384.

Another important detail about our AEAD implementations, which directly concerns the hardware API, is that we assume the padding is done a priori to the AEAD call. In other words, our implementations leave padding task to the caller, and assume that the associated data and message bits are well aligned with the block boundaries. This is in contrast to the CAESAR Hardware API, which assumes the padding as the responsibility of the circuit [HDF+16]. Hence, our reported area figures should be carefully interpreted, if one happens to compare them with other implementations which contain the padding circuit. AEAD mode of operations generally treat the last, empty or partial blocks specially through some allocated bits in the domain separator. Hence, when assuming that the associated data and message are properly chopped into blocks and passed to the circuit, information lost during the padding must also be passed along. In our lightweight API, we use few input signals to indicate if the current data block must be specially processed, e.g. whether the current data block is the last block of associated data, or a padded block. The input and output ports of our hardware API are defined in the following way and can be scaled for both 1/4/8-bit inputs:

- `input_wire` CLK, RST: System clock and active-low reset signals.
- `input_vector` KEY, NONCE: Key and nonce ports through which key/nonce are introduced in the circuit in chunks of 1/4/8 bits.
- `input_vector` DATA: Unified data port from which both associated data and regular plaintext blocks are loaded into the circuit in chunks of 1/4/8 bits.
- `input_wire` EAD, EPT: Single-bit signals that indicate whether there are no associated data blocks (EAD) or no plaintext blocks (EPT). Both signals are supplied with the reset pulse and remain stable throughout the computation.
- `input_wire` LBLK, LPRT: Single-bit signals that indicate whether the currently processed block is the last associated data block or the last plaintext block (LBLK), and also whether it is partially filled (LPRT). Both signals are supplied alongside each data block and remain stable until the next block is fed to the circuit.
- `output_wire` BRDY, ARDY: Single-bit output signals that indicate whether the circuit has finished processing a data block and a new one can be supplied on

the following rising clock edge (BRDY) or the entire AEAD computation has been completed (ARDY).

- `output_wire` CRDY, TRDY: Single-bit output signals that indicate whether the CT and TAG ports will have meaningful ciphertext and tag values starting from the following rising clock edge.
- `output_vector` CT, TAG: Separate ciphertext and tag ports, via which the output is available in chunks of 1/4/8 bits.

6.8.1 SUNDAE-GIFT

The SUNDAE-GIFT AEAD scheme was proposed by Banik et al. and is based on the SUNDAE mode of operation, featuring GIFT* block cipher at its core [BBP+19, BBLT18]. It is a bare-bones construction that does not require any additional registers aside the ones used within the block cipher. After the encryption of the init vector, each data block is mixed into the AEAD state between the encryption calls. A field multiplication over $GF(2^{128})$ is applied after the last associated data has been added to the state. The same multiplication is also performed for the last message block. The multiplication is either $\times 2$ when the last AD or message block has been padded or $\times 4$ whenever the last blocks are complete without any padding. More formally, the multiplication $\times 2$ is encoded as a byte-wise shift and the addition of the most significant byte into other bytes of the state such that if $B_0||B_1||\dots||B_{15}$ represents the 16 bytes of the intermediate AEAD state (with B_0 being the most significant byte), we have that

$$2 \times (B_0||B_1||\dots||B_{15}) = B_1||B_2||\dots B_{10}||B_{11} \oplus B_0||B_{12}||B_{13} \oplus B_0||B_{14}||B_{15} \oplus B_0||B_0,$$

and $4 \times (B_0||B_1||\dots||B_{15}) = 2 \times (2 \times (B_0||B_1||\dots||B_{15}))$. The tag is produced after processing all AD and message blocks and the ciphertext blocks are generated by reprocessing the message blocks afterwards. A schematic of the SUNDAE-GIFT is depicted in Figure 6.8.

The simplicity of SUNDAE-GIFT can be exploited in a bit-serial implementation to attain a circuit with very low overhead in terms of area. In fact, except for the slight increase in the control logic, the sole addition to the GIFT* circuit presented in Section 6.6 is the field multiplication.

The multiplier can be achieved with two swaps (one for $\times 2$, another for $\times 4$) and one XOR gate. More concretely, we allocate 128 rounds for the multiplication $\times 2$ and $\times 4$ during which the block cipher round function and key swaps are disabled. In other words, while the ciphertext bits exit the last round function computation, we swap FF_{120} and FF_0 during the cycles 8 to 127 which rotates the state by 8 positions to the left. Similarly, FF_{112} and FF_0 are swapped during the cycles 16 to 127 in order to execute the 16-bit rotation. Hence, in the worst case, we require $2 \times 128 = 256$ additional cycles for

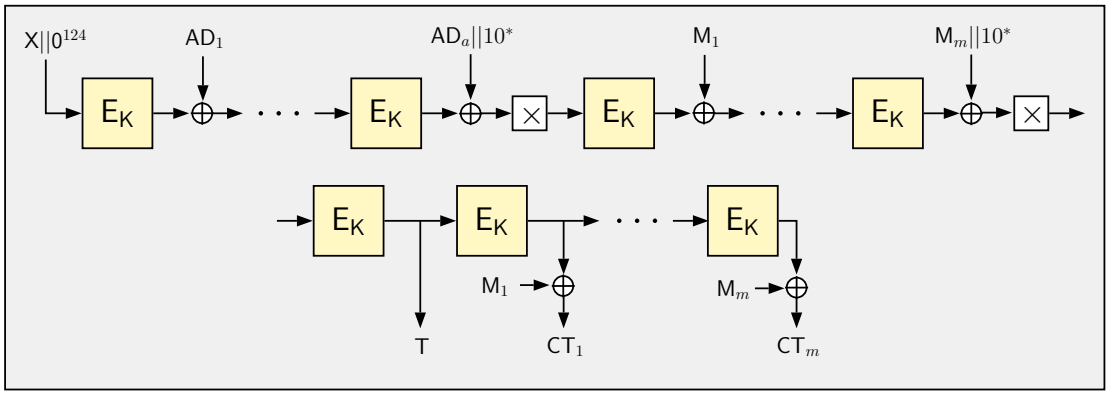

Figure 6.8 – The high-level overview of SUNDAE-GIFT, which depicts the processing of m message and a associated data blocks. X denotes a 4-bit parameter, whose value depends on the length of the nonce and whether there are no AD or message blocks.

multiplications. In terms of latency, each new encryption call is loaded with the new plaintext, while the ciphertext bits of the previous computation exit the pipeline. As a consequence, the very first encryption operates over $41 \times 128 = 5248$ cycles, while the remaining encryption each take $40 \times 128 = 5120$ cycles.

The 1-bit version of SUNDAE-GIFT can seamlessly be amended to a 4-bit datapath design by changing the bit swaps to nibble swaps. After synthesis, the resulting SUNDAE-GIFT architecture is the smallest authenticated encryption circuit at around 1200 GE for the STM 90 nm process, which is only a 8 percent larger compared to the bit-serial GIFT* implementation presented in Section 6.6.

6.8.2 SAEAES

The SAEAES AEAD scheme was proposed by Naito et al. [NMMaS+19] and uses the AES block cipher as the underlying encryption core. The SAEAES document offers a number of parameters according to which the mode can be operated, but the primary candidate among them is SAEAES128-64-128, which implies a key size of 128 bits, message/AD blocks of 64 bits and a tag size of 128 bits. This effectively makes the primary mode of rate 1/2, since 2 block cipher calls are required per 128 bits of message/AD. However, the mode requires no additional state other than those required in the calculation of the block cipher encryption and so a very compact implementation is possible.

We only summarize the details regarding the 1-bit implementation, as transforming it to 8-bit follows the generic technique outline in Section 6.3. A high-level description of the mode of operation is presented in Figure 6.9. It is easy to see that this mode of operation does not require additional storage other than the ones required in the block cipher. From a circuit designer's point of view, it is not difficult to implement the mode, as the only

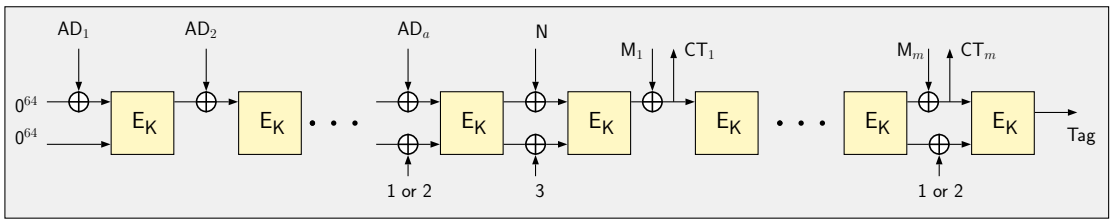

Figure 6.9 – The high-level overview of SAEAES, which depicts the processing of m message and a associated data blocks.

real challenge is to ensure that at the beginning of a particular encryption operation the circuit feeds the correct input vectors to the block cipher circuit, which are as follows:

- $\mathsf{Inp}_i = \mathsf{AD}_i || 0^{64} \oplus \mathsf{E_K}(\mathsf{Inp}_{i-1})$ or AD_1 (if $i = 1$) during the associated data processing stage, where Inp_i is the i-th input to the block cipher.
- $\mathsf{Inp}_a = \mathsf{AD}_a || \mathsf{const}_{64} \oplus \mathsf{E_K}(\mathsf{Inp}_{a-1})$ for the last AD block, where const_{64} denotes a 64-bit constant.
- $\mathsf{IV} = \mathsf{N} \oplus 0^{126}11 \oplus \mathsf{E_K}(\mathsf{Inp}_a)$ before the processing of the plaintext begins, where $0^{126}11$ corresponds to the number 3 encoded as 128-bit string and N denotes the nonce.
- $\mathsf{Inp}'_i = \mathsf{M}_i \oplus \mathsf{E_K}(\mathsf{Inp}'_{i-1})$ during the plaintext processing stage, where Inp'_i is the i-th input to the block cipher during plaintext processing. It can also be seen that Inp'_i is also incidentally the i-th ciphertext block, and the tag is simply the outcome of the final encryption call that the mode performs.

A bit-wise AES encryption core produces output 1 bit per clock cycle during the last 128 cycles of the encryption operation. Since we are using no additional storage blocks, the output bits, once produced, need to be XORed with the appropriate input signal and concurrently fed back to the block cipher as the input of the following encryption call. Essentially, cycles 1281 to 1408 not only produce the output of the i-th encryption but also serve as the input period for the $(i + 1)$-th encryption. Thus one needs to exercise some more fine-grained control over the circuit, to ensure that the block cipher circuit is able to perform the dual role during cycles 1281 to 1408. This effectively means that all encryption calls except the first requires 1280 cycles. Hence, in order to process a AD and m plaintext chunks of 64 bits each, the circuit requires $a + m + 1$ encryption calls which leads to $1408 + 1280 \times (a + m)$ cycles.

6.8.3 Romulus

Romulus is an AEAD scheme designed by Iwata et al. [IKMP19], and uses the SKINNY family of block ciphers. In this work, we provide implementations for two members

Romulus-N1 (both 1-bit and 8-bit) and Romulus-N3 (1-bit only). The former is the primary candidate of the family that employs SKINNY-128-384, whereas the latter is the lightest among them because it employs SKINNY-128-256.

In order to reduce the number of block cipher calls, and make use of the large tweakey space, that is 384 bits for the primary member, Romulus makes 1/2 block cipher call per associated data block, and 1 block cipher call per message block. Romulus-N1 member admits 128-bit key, 128-bit nonce, variable-length message chopped into 128-bit blocks, and produces 128-bit tag. In terms of input parameter sizes, the difference in Romulus-N3 is that it uses 96-bit nonce. An interesting design choice regarding Romulus is that associated data blocks can have alternating size based on which member is chosen. For example, with Romulus-N3, for some integer i, AD_{2i-1} blocks are 128-bit, and AD_{2i} blocks are 96-bit. In order to ease notation and the description, one can actually treat $\mathsf{AD}_{2i-1}||\mathsf{AD}_{2i}$ as a single 224-bit block, assuming that the original padding is preserved during this conversion. In the case of Romulus-N1, things are much simpler, because all associated data blocks are fixed to 128 bits. Figure 6.10 describes the three phases a full

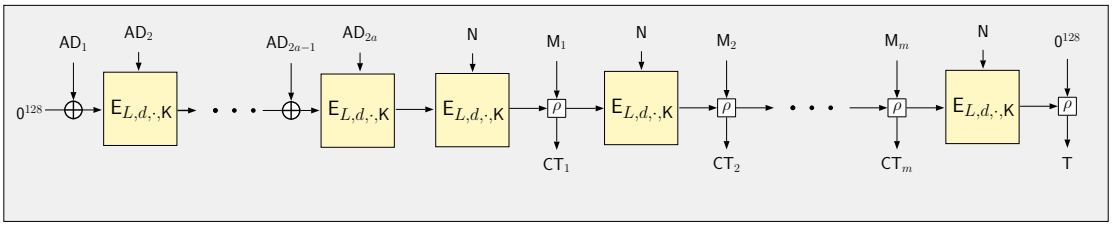

Figure 6.10 – The high-level view of Romulus-N1, which depicts the processing of $2a$ associated data and m message blocks. L denotes the 56-bit LFSR that counts the number of processed blocks, and d denotes a single byte domain separator followed by 0^{64}.

AEAD operation passes through, namely processing of (1) associated data, (2) nonce and (3) message blocks. Below, we first explain Romulus-N3 and the crucial details regarding its 1-bit implementation, and give the differences for Romulus-N1 later.

During associated data phase, each combined 224-bit $\mathsf{AD}_{2i-1}||\mathsf{AD}_{2i}$ block is processed with a single block cipher call E_K. For each of these SKINNY-128-256 calls, the plaintext is AD_{2i-1}, and the tweakey is a concatenation of a 24-bit counter[1], an 8-bit domain separator, a 96-bit AD_{2i} block and the 128-bit key K. The output from the block cipher is treated as the running state, and XORed with each new AD_{2i-1} block. Once all $\mathsf{AD}_{2i-1}||\mathsf{AD}_{2i}$ combined blocks are processed, the running state is encrypted by using the nonce N itself as a part of the tweakey. We refer to this as processing of the nonce. During the message phase, for each of the 128-bit message blocks, the running state and the message block M_i are passed through ρ function defined below. Essentially ρ acts as

[1]The 24-bit counter is defined with regards to a LFSR (see [IKMP19]), and counts the number of block cipher calls during a phase.

XOR in the lateral direction, hence the running state is XORed with the message blocks as before. Once all message blocks are processed, the final block cipher output is passed through ρ with 0^{128} to produce the tag. $\rho(S, M) = (S', C)$ is defined as $S' \leftarrow S \oplus M$ and $C \leftarrow G(S) \oplus M$. For each byte, G performs the following operation:

$$G(x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0) := (x_0 \oplus x_7)||x_7||x_6||x_5||x_4||x_3||x_2||x_1$$

It is then clear how we can use 1-bit-serial SKINNY-128-256 to realize Romulus-N3. Except for the computation of the ciphertext blocks through ρ, we can simply reuse the state pipeline of SKINNY-128-256 to store the running state. In order to compute G, we use two external 7-bit buffer pipelines, which keeps the copy of the last 7 bits that exit the state pipeline and the last 7-bit of message block which is being fed to the circuit. This leads to 7 clock cycle of delay in between the time a message block is fed and the time the ciphertext bits become available. This similarly applies to the tag as well, hence the delay of 7 clock cycles must be considered during latency calculation. As a concrete example, the circuit would process 2×224 bits of associated data and 1×128 bits of message as follows:

- During the first 128 cycles, the key K, and the first associated data block AD_1 are loaded simultaneously. Starting from the clock cycle 32, 96-bit AD_2 is also being loaded[2]. After loading is complete, the circuit becomes busy for 47 rounds (for SKINNY-128-256 encryption), i.e. this takes 47×128 clock cycles. At the last clock cycle, the circuit signals that it is ready for receiving the next data block, which can be either AD_3 or M_1, depending on whether there are more associated data blocks to process. For the sake of this example, we assume there are 224 more bits of associated data to process.

- For the following 128 cycles, the state pipeline XORs its content with AD_3, and initiates the first round of encryption simultaneously. Again, the key is reloaded starting from cycle 0 and AD_4 is also loaded starting from clock cycle 32. The circuit becomes busy for 47 rounds to compute the encryption. At the last cycle, the circuit signals that the key and the nonce must be reloaded during the following round.

- The running state is encrypted, i.e. the state pipeline reloads its own content and starts encryption. No data block needs to be loaded, but the key and the nonce must be loaded simultaneously. Since nonce and 96-bit AD blocks are using the exact same positions in the tweakey, the nonce is loaded starting from clock cycles 32. After 47 rounds, the circuit signals that the next data (i.e. message) block can be loaded.

[2]According to Romulus-N3 specification, the last 96 bits of $TK1$ should receive nonce/associated data blocks. The leftmost 32 bits of $TK1$ are reserved for the counter and the domain separator.

- The message block is loaded, which happens simultaneously with reloading of the key. The nonce also follows the key with 32 clock cycles delay, as before. The ciphertext bits become available with 7 clock cycles of delay. The circuit again takes 47 rounds to perform the final encryption.
- A final ρ operation is performed with the running state and the 0^{128} vector. The tag becomes available with 7 clock cycles delay.

As for 1-bit-serial implementation of Romulus-N1, the steps taken by the state machine is precisely the same. As for differences, however, (1) the invoked block cipher is SKINNY-128-384, (2) all associated data blocks are 128-bit, hence loading for even and odd-numbered associated data blocks (as well as nonce) starts and ends at the same clock cycles, (3) and the block counter is defined as 56-bit LFSR (instead of 24-bit). Moving towards 8-bit implementation is also quite straightforward, with the only difference being the removal of the 7 clock latency caused by ρ function. As it operates on the byte level, it is realized as a fully combinatorial circuit.

According to the 1-bit implementation of Romulus-N1, processing 1 AD blocks and 8 message blocks takes $(1 + 8) \times 56 \times 128 + 128 + 7$ clock cycles. The additional 128 clock cycles are incurred due to the delay of loading/flushing the pipelines, and the 7 clock cycle is due to the execution delay of ρ. As for 8-bit implementation, the clock cycles are amended as $(1 + 8) \times 56 \times 16 + 16$.

6.8.4 SKINNY-AEAD

SKINNY-AEAD relies on the ΘCB3 mode of operation [KR11] and uses the heaviest SKINNY variant, i.e. SKINNY-128-384, as the core block cipher. ΘCB3 requires the addition of three auxiliary registers that store intermediate values during the computation; a 128-bit register denoted by `Auth` that accumulates the encrypted AD block, a second 128-bit register Σ that holds the summation of all message blocks and finally a 64-bit LFSR block counter L. Both the 1-bit and 8-bit version of SKINNY-AEAD can be instantiated without any further modifications to the serial SKINNY-128-384 cores.

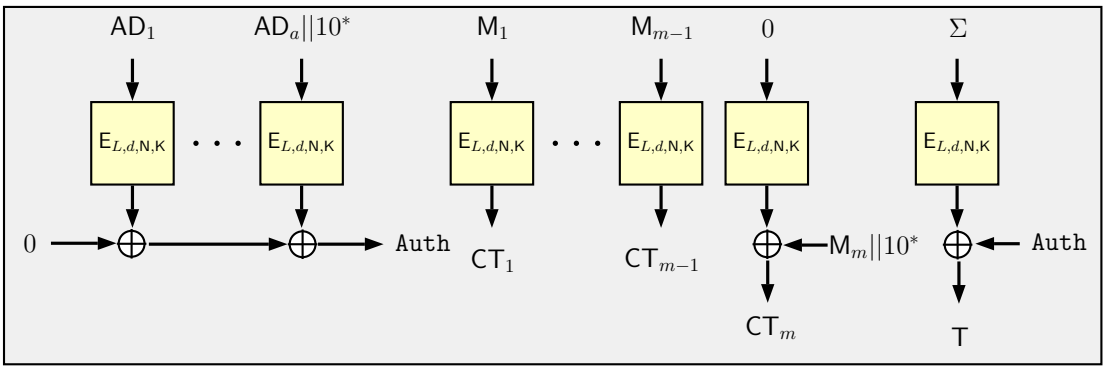

Figure 6.11 – The high-level view of SKINNY-AEAD. The block counter L, a domain separator d, the nonce N and the encryption key K together make up the 384-bit tweakey. The encryption of the zero string is only performed when the last message block is incomplete.

6.8.5 Synthesis Results

Table 6.13 – Synthesis figures for selected AEAD Schemes in STM 90 nm and UMC 90 nm libraries. Energy and throughput are based on 1024 bits of plaintext and 128 bits of AD.

Candidate	Datapath	Area (μm^2)	Area (GE)	Power(μW) @ 10 MHz	Latency (cycles)	Energy (nJ/1152-bit)	Throughput (Mbit/s)
			STM 90 nm				
SUNDAE-GIFT	1-bit	5273.9	1201	50.1	92544	463.6	4.48
SUNDAE-GIFT	4-bit	6969.8	1587	63.9	23136	147.8	13.76
SAEAES	1-bit	5938.0	1350	77.2	24448	188.7	6.13
SAEAES	8-bit	8534.9	1940	108.0	3056	33.0	55.14
Romulus-N1	1-bit	10534.8	2399	98.1	64647	634.2	4.91
Romulus-N1	8-bit	12783.8	2912	114.6	8080	92.6	33.24
Romulus-N3	1-bit	7812.7	1780	79.1	55431	438.5	5.92
SKINNY-AEAD	1-bit	15756.1	3589	134.3	72960	979.9	5.04
SKINNY-AEAD	8-bit	16606.7	3783	149.0	9856	146.9	37.16
Grain-128AEAD[3]	1-bit	9576.6	2181	102.0	1664	17.0	331.78
Grain-128AEAD	4-bit	11378.8	2592	104.0	416	4.3	1320.47
Grain-128AEAD	8-bit	14324.4	3263	106.0	208	2.2	2614.78
			UMC 90 nm				
SUNDAE-GIFT	1-bit	4729.9	1508	51.1	92544	472.9	4.67
SUNDAE-GIFT	4-bit	6109.7	1948	63.5	23136	146.9	13.01
SAEAES	1-bit	5329.6	1700	95.0	24448	232.3	9.52
SAEAES	8-bit	8094.0	2581	103.9	3056	31.7	55.56
Romulus-N1	1-bit	9683.2	3088	103.7	64647	670.4	4.80
Romulus-N1	8-bit	11696.5	3730	118.5	8080	95.7	30.02
Romulus-N3	1-bit	7155.6	2282	81.6	55431	452.3	6.31
SKINNY-AEAD	1-bit	14567.5	4645	143.1	72960	1044.1	3.68

[3]The 1/4/8-bit Grain-128AEAD implementations were taken from [SHSK19] and re-synthesized.

SKINNY-AEAD	8-bit	15161	4834	155.0	9856	152.8	23.42
Grain-128AEAD	1-bit	7354.7	2345	91.4	1664	15.2	354.96
Grain-128AEAD	4-bit	9006.6	2872	94.4	416	3.9	1239.87
Grain-128AEAD	8-bit	11255.1	3589	100.0	208	2.1	2456.68

Table 6.14 – Low-latency synthesis figures for selected AEAD Schemes in TSMC 90 nm, NanGate 15nm and 45nm libraries. Energy and throughput are calculated for processing 1024 bits of plaintext and 128 bits of AD.

Candidate	Datapath	Area (μm^2)	Area (GE)	Power(μW) @ 10 MHz	Latency (cycles)	Energy (nJ/1152-bit)	Throughput (Mbit/s)
TSMC 90 nm							
SUNDAE-GIFT	1-bit	4444.6	1576	45.9	92544	424.8	5.37
SUNDAE-GIFT	4-bit	5640.6	2000	52.1	23136	120.5	12.73
SAEAES	1-bit	4942.7	1751	56.9	24448	139.1	7.11
SAEAES	8-bit	6895.1	2452	70.2	3056	21.5	61.88
Romulus-N1	1-bit	9019.0	3198	95.3	64647	616.1	6.99
Romulus-N1	8-bit	10552.3	3742	100.8	8080	81.4	39.30
Romulus-N3	1-bit	6658.8	2361	74.0	55431	410.2	9.31
SKINNY-AEAD	1-bit	13554.6	4807	122.5	72960	893.8	6.84
SKINNY-AEAD	8-bit	13943.4	4944	137.0	9856	135.0	35.96
Grain-128AEAD	1-bit	7509.0	2663	87.4	1664	14.5	452.21
Grain-128AEAD	4-bit	8763.6	3108	93.1	416	3.9	1375.49
Grain-128AEAD	8-bit	13943.4	4944	95.9	208	2.0	2627.79
NanGate 15 nm							
SUNDAE-GIFT	1-bit	426.6	2170	15.9	92544	147.1	84.80
SUNDAE-GIFT	4-bit	541.4	2754	19.5	23136	45.1	279.33
SAEAES	1-bit	464.3	2362	18.8	24448	46.0	142.21
SAEAES	8-bit	606.7	3086	24.9	3056	7.6	1119.93
Romulus-N1	1-bit	882.3	4488	32.1	64647	207.5	73.89
Romulus-N1	8-bit	1012.9	5152	36.8	8080	29.7	566.54
Romulus-N3	1-bit	650.8	3310	25.0	55431	138.6	152.46
SKINNY-AEAD	1-bit	1323.5	6732	46.0	72960	335.6	75.29
SKINNY-AEAD	8-bit	1381.2	7025	32.0	9856	31.5	530.80
Grain-128AEAD	1-bit	631.4	3211	20.4	1664	3.4	3143.97
Grain-128AEAD	4-bit	732.5	3726	20.6	416	0.9	11003.88
Grain-128AEAD	8-bit	914.7	4652	21.3	208	0.4	21127.46
NanGate 45 nm							
SUNDAE-GIFT	1-bit	1527.9	1910	130.3	92544	1205.8	14.13
SUNDAE-GIFT	4-bit	1871.3	2339	168.2	23136	389.1	49.98
SAEAES	1-bit	1653.5	2067	148.8	24448	363.8	21.20
SAEAES	8-bit	2190.2	2745	205.5	3056	62.8	186.27
Romulus-N1	1-bit	3103.4	3879	265.5	64647	1716.4	17.17
Romulus-N1	8-bit	3566.0	4458	311.9	8080	252.0	99.25
Romulus-N3	1-bit	2304.1	2880	199.0	55431	1103.1	19.82
SKINNY-AEAD	1-bit	4784.0	5980	408.4	72960	2979.7	15.21
SKINNY-AEAD	8-bit	4793.3	5992	410.8	9856	404.9	94.46
Grain-128AEAD	1-bit	2584.4	3231	214.0	1664	35.6	1082.35
Grain-128AEAD	4-bit	2958.5	3698	237.1	416	9.9	4001.41
Grain-128AEAD	8-bit	3609.1	4511	288.1	208	6.0	7545.52

6.8.6 Interpretation of Power and Throughput Results

So far in this chapter, we have extensively focused on the trade-off between the area versus the latency of an evaluated circuit on a rather abstract manner, where the latency is expressed in terms of number of clock cycles. From this point of view, the latency is an attribute of the proposed architecture, and therefore it is independent of the implementation technology. On the other hand, the throughput of the circuit heavily depends on the implementation technology. For a single invocation of the encryption operation, let us use the latency parameter ℓ to denote the total number of clock cycles used by the circuit during this operation, N to denote the number of plaintext bits and τ to denote the largest timing delay that comes from the longest path of the circuit. Then, we compute the maximum achievable throughput T_{max} by

$$T_{\mathsf{max}} = \frac{N}{\tau \times \ell}$$

Therefore, once we fix an AEAD scheme, conversion from latency to throughput only depends on the parameter τ. It is evident that depending on the timing characteristics of the library cells, as well as the compilation options used to synthesize the circuit control the value of τ. In this section, we first elaborate on the technology library aspect of the throughput and energy. Later, we shall give brief explanation as to why there are large observed differences among schemes such as SUNDAE-GIFT, SKINNY-AEAD and Grain-128AEAD.

Energy comparison. It is worth noting that STM 90 nm and TSMC 90 nm are two libraries for which the available variants from our design kits were tailored for low-power consumption. Naturally, this implies that the cells are designed in a way that the leakage power is minimal, and hence large amount of consumed energy is proportional to the total switching activity of the circuit. On the other hand, the variant we used with UMC 90 nm library is not low-power, but instead a *standard* version. In the case of NanGate 15 nm and NanGate 45 nm libraries, the *fast* variants are used, which are much more leaky in comparison. This overall comparison can be much better summarized in Figure 6.12. Here, the power consumption is extracted from 1-bit serial 6AES circuit, similar to the one described in Chapter 3. The same results were also reported in a follow-up work by Lombardía, Balli and Banik [LBB21].

We typically use 10 MHz to report the average power consumption. Hence at this frequency, according to Figure 6.12, it is evident that large portion of the power consumption is due to leakage for both NanGate 15 nm and NanGate 45 nm. This immediately tells us that we should cautiously interpret the power measurements in Table 6.14 and Table 6.13. Namely, the reported figures do not purely reflect the switching activity, and there is a significant share of power that scales with the area of the circuit. The same phenomenon happens with UMC 90 nm also, but to more limited extent. As for STM 90 nm and TSMC 90 nm, the reported power measurements are close to pure switching activity, and

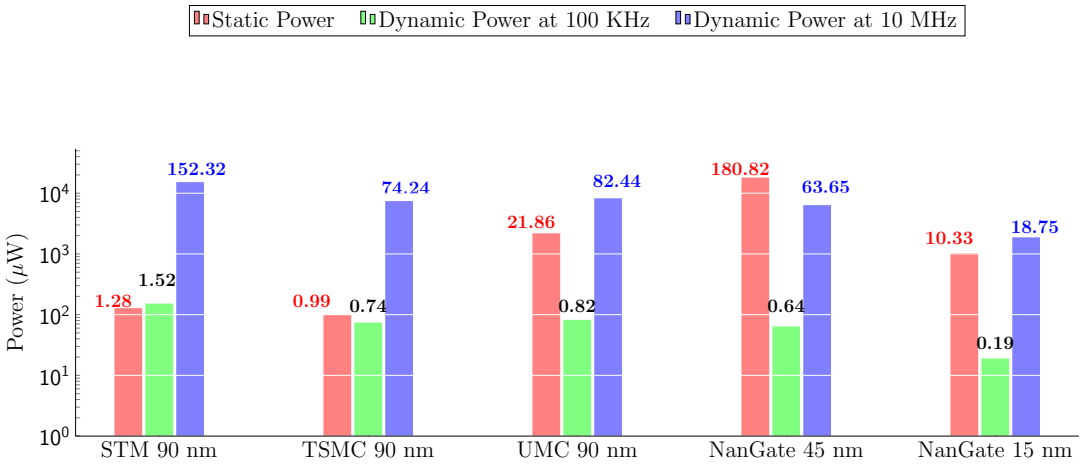

Figure 6.12 – Breakdown of the total power consumption into its static and dynamic components for 1-bit serial 6AES circuit reported by Lombardía et al. [LBB21]. Note that y-axis is scaled logarithmically.

hence they are somewhat independent of the circuit size.

Throughput comparison. In our synthesis, we used the `compile_ultra` setting. This means that the throughput, or reduction of the largest timing path, is not the priority during compilation. In that sense, the reported throughput values do not necessarily reflect the absolute maximum throughput that can be obtained from the given circuits. Instructing the compiler to take further timing constraints into account would lead to smaller timing values, and hence it would increase the throughput. This would come at the cost of a slight increase in the area and power consumption, because in order to shorten the largest path, the compiler typically picks larger cells with better timing characteristics from the technology library. The amount of feasible throughput reduction also depends on how rich the technology library is in terms of a number of different silicon implementations for the same gate. For instance, TSMC 90 nm provides five different cells for the simple 2-input NAND gate, whereas NanGate 15 nm provides only two implementations.

Scheme differences. Despite the large variance among multiple libraries, it stands out that SKINNY-AEAD > SUNDAE-GIFT > Grain-128AEAD in terms of energy consumption. There are surely valid intuitions as to why such large differences in energy appear, once these schemes are put into circuit form.

Grain-128AEAD follows a stream-cipher-based AEAD design paradigm. In other words, it spares the design from the use of block ciphers, which are much heavier in design. While a block cipher typically needs to operate many rounds of computationally demanding layers, a stream-cipher constructions follow a LFSR-based paradigm with much lighter and

faster operations. This explains not only why Grain-128AEAD outperforms the all other candidates in terms of minimizing energy consumption, but also in terms of throughput as well.

On the other side, among block-cipher-based candidates, the throughput and energy cost of the AE scheme typically depends on how heavy the core block cipher is. Here, the block ciphers of interest are mainly AES, GIFT* and SKINNY-128-384. Both in terms of minimum energy consumption and maximum throughput, these ciphers can be given as AES > GIFT* > SKINNY-128-384. In other words, AES is the most lightweight choice with respect to energy and throughput metrics. This is a natural result of the number of rounds required by each block ciphers, that are respectively 10, 40 and 56 rounds.

Here, we can see the extension of this cost all the way into AE schemes. Mainly, for Romulus-N1 and SKINNY-AEAD, the underlying block cipher is SKINNY-128-384, which makes them much more costly compared to SUNDAE-GIFT and SAEAES, in terms of energy consumption and throughput.

6.9 Cost of Decryption

Some of the AEAD schemes in the NIST LWC [NISa] do require the inverse, i.e. the decryption functionality, of block ciphers as well. Therefore, in this section, we assess the cost of implementing the combined encryption and decryption circuit for the three block ciphers.

For AES, the challenge really comes in arranging the order of operations, these are namely inverse MixColumns, inverse ShiftRows and inverse SubBytes. Note that the inverse ShiftRows is also a special permutation of type 4, and a swap sequence can be constructed in the same way as described in Section 6.3. Furthermore, in order to avoid adding more MUXes to the circuit, we can reuse the same swap locations as much as possible from the encryption. Hence, the cost of implementing inverse ShiftRows is small, other than the control logic required to generate the sequence of selection bits that controls when to swap and when not to do so. The inverse MixColumns operation is perhaps the most difficult operation to implement in this setting. It is well known that MixColumns matrix M used in AES has the property that $M^3 = M^{-1}$. Hence, if we want to implement multiplication by M^{-1} without any extra gates, then it would be necessary to make three full rotations in the state pipeline until the MixColumns operation is completed (this is the approach tried out in [BBR16b, JMPS17]). This invariably comes with a latency penalty. If we do not want to impose a latency penalty, we must pay with extra gate area, by accommodating 2 additional MixColumns circuits one after the other. This comes with an additional area penalty of 100–120 GE, but makes it possible to complete the decryption round in 128 clock cycles. Implementing a combined circuit for the forward and inverse S-box also requires at most 50 GE [ME19]. The inverse key schedule can be

implemented without much additional logic as already explained in [BBR16a].

For SKINNY, which also has the similar structure with AES, the costs are considerably smaller. Inverse ShiftRows is again easily implemented with the techniques described in Section 6.3. For this cipher, the S-box and MixColumns circuit are extremely lightweight, and so the forward and inverse function can be implemented without any significant area costs. The inverse tweakey update functions at all the 3 levels are simple byte-based LFSR updates and permutations, and can be instantiated with only a few multiplexers.

For GIFT, both the swaps in the round function and the key schedule are partitioned in layers that are executed one after another (see Section 6.6.1). This means the decryption can be performed by simply inverting the order of the swap layers. The main overhead comes therefore in the form of the additional inverse S-box which can be synthesized with fewer than 20 GE.

6.10 Conclusion

To conclude the chapter, let us take a look back at the main results of this chapter. The synthesis results of AES, SKINNY and GIFT* are summarized in Tables 6.6, 6.8, 6.10 respectively for 5 standard cell libraries. For each of the 3 block ciphers, we can deduce that in the bit-serial mode, the area occupied by the circuits is very close to the total area required by the storage elements for the state and key registers. For AES, the area is only slightly larger, since it has an 8-bit S-box and a reasonably heavyweight MixColumns circuit. But for the other ciphers that have relatively lightweight S-box and linear layer, the purely combinatorial circuit elements occupy only around 10% of the total silicon area. Additionally, we are able to reduce the round latency to match precisely the block size of the underlying block cipher. Note that it is not possible to have an implementation that has lower round latency in clock cycles than the block size of the cipher, because for a bit-serial circuit of SPN-based ciphers, all the state bits must be rotated across the pipeline. Therefore, this represents the sweet spot in the area-latency curve, as far as SPN-based block ciphers are concerned. For implementations with higher-bit data paths, the area only marginally grows, mainly because the number of MUXes and XOR gates required in the circuit needs to be multiplied by the length of the datapath the circuit aims to achieve.

An interesting research direction is to extend our results to Feistel-based block ciphers. Note that the SIMON/SPECK family of block ciphers were mainly designed to achieve this optimal trade-off point, as the state update of these ciphers can simply be described using rotate and a few bit-wise AND/XOR operations [BSS+13]. Nevertheless, not all Feistel ciphers in the literature are designed with this specific goal in mind. It would be interesting to see how a block cipher like PICCOLO can be re-engineered at the circuit level to achieve the best possible area and latency figures [SIH+11b].

Tables 6.13 and 6.14 tabulate synthesis results we obtained for all the individual modes of operation that we investigated in this paper. SUNDAE-GIFT and SAEAES are essentially rate 1/2 modes that need 2 block cipher calls for every 128-bit message block. Note that for these two, the underlying block ciphers admit 128-bit key, and they require exactly 256 flip-flops to store the key and the state. Thus in a sense, minimalism of the core block cipher comes at the cost of having to execute 2 block cipher calls per 128-bit message block. On the other hand, the rate 1 modes, which require only 1 block cipher call per block of message, such as Romulus and SKINNY-AEAD employ SKINNY-128-384. They take advantage of the large (384-bit) tweakey space to accommodate nonce, domain separator, and counter for each block cipher invocation. However, for SKINNY-128-384, this comes at the cost 512 flip-flops for both the state and the tweakey. This leads to an interesting latency and area trade-off, and our work gives further insights on the nature of these design decisions.

7 Conclusion and Future Work

In this thesis, we studied commonly used lightweight block ciphers, e.g. AES, GIFT, PRESENT, SKINNY, and few of the block-cipher-based AEAD schemes that employ these block ciphers, from the perspective of hardware lightweight metrics. Our study looked at their 1-bit serial implementation, for the sake of pursuing even further reduction in the silicon area. We proposed the swap-and-rotate technique to handle the execution of the fine-grained permutation layers, which can be seen as an economical solution to a relocation problem among storage elements. We further show how streamlined execution can be adopted in the serial implementation of block ciphers, instead of the conventional strictly-ordered execution among inner layers of the round functions.

In Chapter 3, our effort was on the architectural-level optimization of AES. The idea of combining multiple functionality into the same circuit can come in handy for those constrained applications (such as java cards) that are expected to support a large portfolio of protocols. Instead of implementing each of these primitives as stand-alone circuits, considerable silicon area can be saved by reusing the same register blocks. Although we only looked at the family of AES, for the future work, the similar approach can be taken for other family of block ciphers such as GIFT and SKINNY. This approach would work very well particularly for SKINNY family, as the key scheduling operations among the members SKINNY-128-128, SKINNY-128-256 and SKINNY-128-384 mostly contain the same operations.

Moreover, it is not far-fetched to envision implementation of a single circuit that supports all encryption and decryption functionalities of multiple block ciphers at once. For instance, SKINNY-128-128 and AES-128 could be implemented on top of the same 128-bit state and key pipelines. This approach is promising, as roughly 80% of the silicon area is used by the storage elements in serial block cipher implementations. The modular approach we presented simplifies this seemingly complicated engineering task. Along these lines, later a follow-up work was presented by Lombardía, Balli and Banik [LBB21]. Lombardía's implementation starts with the same design ideas from Chapter 3, and

combines this approach with the swap-and-rotate technique introduced in Chapter 5. The final circuit is 1-bit serial all-in-one 6AES circuit that only occupies 2268 GE of area, with further 38% reduction in silicon area.

In Chapter 4, our effort was on testing the design intuition behind the fork cipher, from the perspective of lightweight hardware metrics. In particular, we evaluated ForkAES, as this was the first proposed fork cipher construction by the designers [ARVV18]. Later on, in the full-fledged AEAD candidate ForkAE, instead of ForkAES, the designers opted for ForkSKINNY, constructed by forking SKINNY block cipher [ALP+19]. Therefore the future work could look into comparison between SKINNY and ForkSKINNY, and even extend the results to full AEAD scheme ForkAE. The techniques presented in this chapter can also be extended to produce the smallest and most energy-efficient ForkAE circuits. On the other hand, it is unclear whether the forking paradigm will be used in practice, as it was eliminated during the final selection process of NIST LWC.

In Chapter 5, we looked at 1-bit permutation layers of block ciphers PRESENT and GIFT, when they are implemented on top of 1-bit serial pipeline. A trivial solution to implement any permutation layer over the cipher state is to add 1 MUX for each flip-flop, yet this approach is expensive. Therefore, we studied how we can express these permutations in terms of simple swap operations, in order to make do with as few extra MUX gates as possible. Our techniques are not limited to particular permutations that operate at 1-bit level, as we have extended these results into larger-width block ciphers such as AES and SKINNY in Chapter 6. In this thesis, for further application of our technique, we only looked at few block ciphers that are common among NIST LWC candidates, with varying block sizes: GIFT, GIFT*, PRESENT, AES, SKINNY. However, our technique is not necessarily limited to this small set of block ciphers. In particular, other SPN-based ciphers such as Pyjamask, Saturnin, TweAES, TweGIFT, ForkSKINNY could also be implemented via our technique. The research effort here goes in two directions: first, the permutation layers need to be concisely expressed in terms of minimum number of swaps, and secondly, the deduced swap operations must be seamlessly integrated with other operations of the round function, i.e. round key addition and S-box. In the case of TweAES, TweGIFT and ForkSKINNY, the tweaked part of the block cipher does not modify the permutation layer, hence the effort to produce these primitives would be relatively small.

Our findings on the simpler and cheaper executions of permutation layers are followed up by block cipher and block-cipher-based AEAD schemes in Chapter 6. We proposed 1-bit and 4/8-bit serial implementations for AES, SKINNY and GIFT, with improved latency characteristics while preserving the small area requirement. We also produced few of the smallest AEAD scheme implementations in the literature, with SUNDAE-GIFT, Romulus, SKINNY-AEAD and SAEAES. Given that there are other number of block-cipher-based candidates in the NIST LWC, the future work could focus on the implementations of those with the application of our technique, which achieves the smallest circuits with good

latency properties. Again, the research effort for these other schemes would start with the streamlined serialization of the core block cipher following the footsteps of Chapter 5 and Chapter 6, and later the mode of operation would also be realized through the same level of serialization.

Most of our implementations are realized with the assumption that the circuit is physically protected from the adversary, hence we have not considered protection against any form of side-channel attacks. For devices that are exposed to these class of attacks, producing threshold implementations as the extension of our presented techniques, while keeping the size of the circuit small and the latency minimum remains an open question for future research. In particular there is an open research question regarding the integration of our streamlined serial implementation approach (that targets n clock cycles for n-bit block cipher) with the classical threshold implementation technique [ISW03]. For example, in the 8-bit serial implementation of SKINNY, the architecture by Beierle et al. spends four clock cycles to execute S-box [BJK+16]. In comparison, our unprotected architectures from Chapter 6 spends exactly one clock cycle. Therefore, the challenge here is to ensure that we can squeeze this extra three clock cycles into our dense 128 clock cycles of timetable. With current designs of AES, GIFT and SKINNY, the timetable of operations are already quite dense that there are not spare clock cycles to allocate for extra operations between layers. Hence it remains to be answered whether it is feasible to complete 128-bit round in exactly 128 clock cycles with threshold implementation in place.

A Appendix

A.1 S-boxes

A.1.1 AES S-box

The AES S-box is given in Table A.1. Gate-count wise, the smallest implementation is given by Maximov and Ekdahl [ME19]. The forward-only implementation consists of 58 XOR, 6 XNOR, 27 NAND, 5 NOR and 6 MUX gates, which corresponds to 195.10 GE (see Table 4, [ME19]). The combined implementation consists of 70 XOR, 9 XNOR, 27 NAND, 5 NOR and 16 MUX gates, which corresponds to 253.35 GE (see Table 5, [ME19]). The GE metrics are obtained from the GlobalFoundires 22 nm CMOS technology library.

A.1.2 GIFT S-box

The GIFT S-box is given in Table A.2. The fact that it is 4-bit input and 4-bit output naturally implies that it can be realized with small number of gates. It roughly takes 16.5 GE, as reported by the designers [BPP+17].

A.1.3 SKINNY S-box

The SKINNY S-box is already designed with lightweight principles in mind by Beierle et al., whose description is given in Table A.3. The forward-only S-box can be realized with 8 XNOR and 8 NOR gates (see Figure 3, [BJK+16]).

Table A.1 – AES S-box

	_0	**_1**	**_2**	**_3**	**_4**	**_5**	**_6**	**_7**	**_8**	**_9**	**_A**	**_B**	**_C**	**_D**	**_E**	**_F**
0_	63	7C	77	7B	F2	6B	6F	C5	30	1	67	2B	FE	D7	AB	76
1_	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2_	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3_	4	C7	23	C3	18	96	5	9A	7	12	80	E2	EB	27	B2	75
4_	9	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5_	53	D1	0	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6_	D0	EF	AA	FB	43	4D	33	85	45	F9	2	7F	50	3C	9F	A8
7_	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8_	CD	C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9_	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	B	DB
A_	E0	32	3A	A	49	6	24	5C	C2	D3	AC	62	91	95	E4	79
B_	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	8
C_	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D_	70	3E	B5	66	48	3	F6	E	61	35	57	B9	86	C1	1D	9E
E_	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F_	8C	A1	89	D	BF	E6	42	68	41	99	2D	F	B0	54	BB	16

Table A.2 – GIFT S-box

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
1	A	4	C	6	F	3	9	2	D	B	7	5	0	8	E

Table A.3 – SKINNY S-box (8-bit variant)

	_0	**_1**	**_2**	**_3**	**_4**	**_5**	**_6**	**_7**	**_8**	**_9**	**_A**	**_B**	**_C**	**_D**	**_E**	**_F**
0_	65	4C	6A	42	4B	63	43	6B	55	75	5A	7A	53	73	5B	7B
1_	35	8C	3A	81	89	33	80	3B	95	25	98	2A	90	23	99	2B
2_	E5	CC	E8	C1	C9	E0	C0	E9	D5	F5	D8	F8	D0	F0	D9	F9
3_	A5	1C	A8	12	1B	A0	13	A9	5	B5	A	B8	3	B0	B	B9
4_	32	88	3C	85	8D	34	84	3D	91	22	9C	2C	94	24	9D	2D
5_	62	4A	6C	45	4D	64	44	6D	52	72	5C	7C	54	74	5D	7D
6_	A1	1A	AC	15	1D	A4	14	AD	2	B1	C	BC	4	B4	D	BD
7_	E1	C8	EC	C5	CD	E4	C4	ED	D1	F1	DC	FC	D4	F4	DD	FD
8_	36	8E	38	82	8B	30	83	39	96	26	9A	28	93	20	9B	29
9_	66	4E	68	41	49	60	40	69	56	76	58	78	50	70	59	79
A_	A6	1E	AA	11	19	A3	10	AB	6	B6	8	BA	0	B3	9	BB
B_	E6	CE	EA	C2	CB	E3	C3	EB	D6	F6	DA	FA	D3	F3	DB	FB
C_	31	8A	3E	86	8F	37	87	3F	92	21	9E	2E	97	27	9F	2F
D_	61	48	6E	46	4F	67	47	6F	51	71	5E	7E	57	77	5F	7F
E_	A2	18	AE	16	1F	A7	17	AF	1	B2	E	BE	7	B7	F	BF
F_	E2	CA	EE	C6	CF	E7	C7	EF	D2	F2	DE	FE	D7	F7	DF	FF

Bibliography

[6AE] 6AES Implementation Archive. https://lasec.epfl.ch/people/ballif/codes/6aes.zip.

[AGH+19] Riham AlTawy, Guang Gong, Morgan He, Ashwin Jha, Kalikinkar Mandal, Mridul Nandi, and Raghvendra Rohit. SpoC. *NIST Lightweight Cryptography Project*, 2019.

[ÅHJM11] Martin Ågren, Martin Hell, Thomas Johansson, and Willi Meier. Grain-128a: a New Version of Grain-128 with Optional Authentication. *Int. J. Wirel. Mob. Comput.*, 5(1):48–59, 2011.

[ALP+19] Elena Andreeva, Virginie Lallemand, Antoon Purnal, Reza Reyhanitabar, Arnab Roy, and Damian Vizár. ForkAE. *NIST Lightweight Cryptography Project*, 2019.

[ALS] The-Area-Latency-Symbiosis. https://github.com/qantik/The-Area-Latency-Symbiosis.

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like Ciphers: New Bitsliced AES Speed Records on ARM-Cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):402–425, 2021.

[ARH+18] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. SLISCP-light: Towards Hardware Optimized Sponge-specific Cryptographic Permutations. *ACM Trans. Embed. Comput. Syst.*, 17(4):81:1–81:26, 2018.

[ARVV18] Elena Andreeva, Reza Reyhanitabar, Kerem Varici, and Damian Vizár. Forking a Blockcipher for Authenticated Encryption of Very Short Messages. *IACR Cryptology ePrint Archive*, 2018:916, 2018.

[BB19a] Fatih Balli and Subhadeep Banik. Exploring Lightweight Efficiency of ForkAES. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019,*

Proceedings, volume 11898 of *Lecture Notes in Computer Science*, pages 514–534. Springer, 2019.

[BB19b] Fatih Balli and Subhadeep Banik. Six Shades of AES. In *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, volume 11627 of *Lecture Notes in Computer Science*, pages 311–329. Springer, 2019.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A Block Cipher for Low Energy. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, pages 411–436, 2015.

[BBJ+19] Subhadeep Banik, Jannis Bossert, Amit Jana, Eik List, Stefan Lucks, Willi Meier, Mostafizar Rahman, Dhiman Saha, and Yu Sasaki. Cryptanalysis of ForkAES. In *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, pages 43–63, 2019.

[BBLT18] Subhadeep Banik, Andrey Bogdanov, Atul Luykx, and Elmar Tischhauser. SUNDAE: Small Universal Deterministic Authenticated Encryption for the Internet of Things. *IACR Trans. Symmetric Cryptol.*, 2018(3):1–35, 2018.

[BBP+19] Subhadeep Banik, Andrey Bogdanov, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, Elmar Tischhauser, and Yosuke Todo. SUNDAE-GIFT v1.0. *NIST Lightweight Cryptography Project*, 2019.

[BBR15] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Exploring Energy Efficiency of Lightweight Block Ciphers. In *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, pages 178–194, 2015.

[BBR16a] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Atomic-AES: A Compact Implementation of the AES Encryption/Decryption Core. In *Progress in Cryptology - INDOCRYPT 2016 - 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, pages 173–190, 2016.

[BBR16b] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Atomic-AES v 2.0. *IACR Cryptol. ePrint Arch.*, 2016:1005, 2016.

[BBR17] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Efficient configurations for block ciphers with unified ENC/DEC paths. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017*, pages 41–46, 2017.

[BBRV20] Subhadeep Banik, Fatih Balli, Francesco Regazzoni, and Serge Vaudenay. Swap and Rotate: Lightweight Linear Layers for SPN-based Blockciphers. *IACR Transactions on Symmetric Cryptology*, 2020(1):185–232, 2020.

[BCB20] Fatih Balli, Andrea Caforio, and Subhadeep Banik. The Area-Latency Symbiosis: Towards Improved Serial Encryption Circuits. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):239–278, December 2020. Artifact available at https://artifacts.iacr.org/tches/2021/a5.

[BCB21] Fatih Balli, Andrea Caforio, and Subhadeep Banik. The Area-Latency Symbiosis: Towards Improved Serial Encryption Circuits. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):239–278, 2021.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[BCI+19] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT-COFB v1.0. *NIST Lightweight Cryptography Project*, 2019.

[BDE+13] Lejla Batina, Amitabh Das, Baris Ege, Elif Bilge Kavun, Nele Mentens, Christof Paar, Ingrid Verbauwhede, and Tolga Yalçin. Dietary Recommendations for Lightweight Block Ciphers: Power, Energy and Area Analysis of Recently Developed Architectures. In Michael Hutter and Jörn-Marc Schmidt, editors, *Radio Frequency Identification - Security and Privacy Issues 9th International Workshop, RFIDsec 2013, Graz, Austria, July 9-11, 2013, Revised Selected Papers*, volume 8262 of *Lecture Notes in Computer Science*, pages 103–112. Springer, 2013.

[BDV19] Fatih Balli, F. Betül Durak, and Serge Vaudenay. BioID: A Privacy-Friendly Identity Document. In Sjouke Mauw and Mauro Conti, editors, *Security and Trust Management - 15th International Workshop, STM 2019,*

Luxembourg City, Luxembourg, September 26-27, 2019, Proceedings, volume 11738 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2019.

[BFI19] Subhadeep Banik, Yuki Funabiki, and Takanori Isobe. More Results on Shortest Linear Programs. In Nuttapong Attrapadung and Takeshi Yagi, editors, *Advances in Information and Computer Security - 14th International Workshop on Security, IWSEC 2019, Tokyo, Japan, August 28-30, 2019, Proceedings*, volume 11689 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2019.

[BGI+18] Roderick Bloem, Hannes Gross, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal Verification of Masked Hardware Implementations in the Presence of Glitches. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 321–353, Cham, 2018. Springer International Publishing.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 123–153, 2016.

[BJK+19] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. SKINNY-AEAD. *NIST Lightweight Cryptography Project*, 2019.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BLP+08] Andrey Bogdanov, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, and Yannick Seurin. Hash Functions and RFID Tags: Mind the Gap. In *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, pages 283–299, 2008.

[BMA+18] Subhadeep Banik, Vasily Mikhalev, Frederik Armknecht, Takanori Isobe, Willi Meier, Andrey Bogdanov, Yuhei Watanabe, and Francesco Regazzoni. Towards Low Energy Stream Ciphers. *IACR Trans. Symmetric Cryptol.*, 2018(2):1–19, 2018.

[BP12] Joan Boyar and René Peralta. A Small Depth-16 Circuit for the AES S-Box. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors,

Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012.

[BPP+17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 321–345, 2017.

[BRV20] Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the Core Primitive for Optimally Secure Ratcheting. In *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 621–650. Springer, 2020.

[BSS+] Ray Beaulieu, Douglas Shors, Jason Smith, Treatman-Clark Stefan, Bryan Weeks, and Louis Wingers. Simon and Speck: Block Ciphers for the Internet of Things. Available at https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/papers/session1-shors-paper.pdf.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptol. ePrint Arch.*, 2013:404, 2013.

[c4s] The-Area-Latency-Symbiosis. https://c4science.ch/diffusion/10848.

[Can05] David Canright. A Very Compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

[CBB20a] Andrea Caforio, Fatih Balli, and Subhadeep Banik. Energy Analysis of Lightweight AEAD Circuits. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings*, volume 12579 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2020.

[CBB20b] Andrea Caforio, Fatih Balli, and Subhadeep Banik. Melting SNOW-V: Improved Lightweight Architectures. *Journal of Cryptographic Engineering*, 2020.

[CDJ+19a] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas Lopez, Mridul Nandi, and Yu Sasaki. ESTATE. *NIST Lightweight Cryptography Project*, 2019.

[CDJ+19b] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas Lopez, Mridul Nandi, and Yu Sasaki. LOTUS-AEAD and LOCUS-AEAD. *NIST Lightweight Cryptography Project*, 2019.

[CDJN19] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, and Mridul Nandi. HYENA. *NIST Lightweight Cryptography Project*, 2019.

[CDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 272–288, 2009.

[CDL+19] Anne Canteaut, Sébastien Duval, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Thomas Pornin, and André Schrottenloher. Saturnin. *NIST Lightweight Cryptography Project*, 2019.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[CLM16] Victor Cauchois, Pierre Loidreau, and Nabil Merkiche. Direct Construction of Quasi-involutory Recursive-like MDS Matrices from 2-cyclic Codes. *IACR Trans. Symmetric Cryptol.*, 2016(2):80–98, 2016.

[CN19] Bishwajit Chakraborty and Mridul Nandi. mixFeed. *NIST Lightweight Cryptography Project*, 2019.

[Con13] Keith Conrad. Generating Sets, 2013. available at http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/genset.pdf.

[CP08] Christophe De Cannière and Bart Preneel. Trivium. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 244–266. Springer, 2008.

[DL18] Sébastien Duval and Gaëtan Leurent. MDS Matrices with Lightweight Circuits. *IACR Trans. Symmetric Cryptol.*, 2018(2):48–78, 2018.

[EBB15] John M. Kelsey Elaine B. Barker. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, 2015.

[EJMY18] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Cryptol. ePrint Arch.*, 2018:1143, 2018.

[FAE] Lightweight ForkAES. https://c4science.ch/source/lightforkaes.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[FWR05] Martin Feldhofer, J. Wolkerstorfer, and Vincent Rijmen. AES Implementation on a Grain of Sand. *Information Security, IEE Proceedings*, 152:13–20, 11 2005.

[GJK+19] Dahmun Goudarzi, Jérémy Jean, Stefan Kölbl, Thomas Peyrin, Matthieu Rivain, Yu Sasaki, and Siang Meng Sim. Pyjamask. *NIST Lightweight Cryptography Project*, 2019.

[GJN19] Shay Gueron, Ashwin Jha, and Mridul Nandi. COMET. *NIST Lightweight Cryptography Project*, 2019.

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[HDF+16] Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri, Farnoud Farahmand, Panasayya Yalla, Jens-Peter Kaps, and Kris Gaj. CAESAR Hardware API. Cryptology ePrint Archive, Report 2016/626, 2016. https://eprint.iacr.org/2016/626.

[HJM+19] Martin Hell, Thomas Johansson, Willi Meier, Jonathan Sonnerup, and Hirotaka Yoshida. Grain-128AEAD. *NIST Lightweight Cryptography Project*, 2019.

[IKMP19] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Romulus v1.2. *NIST Lightweight Cryptography Project*, 2019.

[ISO12] Information technology — Security techniques — Lightweight cryptography — Part 2: Block ciphers. Standard, International Organization for Standardization, March 2012.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[JMPS17] Jérémy Jean, Amir Moradi, Thomas Peyrin, and Pascal Sasdrich. Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives - Applications to AES, PRESENT and SKINNY. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 687–707, 2017.

[JNP14] Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. Tweaks and Keys for Block Ciphers: The TWEAKEY Framework. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 274–288, 2014.

[KDH+12] Stéphanie Kerckhof, François Durvaux, Cédric Hocquet, David Bol, and François-Xavier Standaert. Towards Green Cryptography: A Comparison of Lightweight Ciphers from the Energy Viewpoint. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 390–407. Springer, 2012.

[KLSW17] Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. Shorter Linear Straight-Line Programs for MDS Matrices. *IACR Trans. Symmetric Cryptol.*, 2017(4):188–211, 2017.

[KR11] Ted Krovetz and Phillip Rogaway. The Software Performance of Authenticated-Encryption Modes. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.

[LBB21] Roldán S. Lombardía, Fatih Balli, and Subhadeep Banik. Six Shades Lighter: a Bit-serial Implementation of the AES Family. *Journal of Cryptographic Engineering*, 2021.

[LSL+19] Shun Li, Siwei Sun, Chaoyun Li, Zihao Wei, and Lei Hu. Constructing Low-latency Involutory MDS Matrices with Lightweight Circuits. *IACR Trans. Symmetric Cryptol.*, 2019(1):84–117, 2019.

[LW17] Chaoyun Li and Qingju Wang. Design of lightweight linear diffusion layers from near-mds matrices. *IACR Trans. Symmetric Cryptol.*, 2017(1):129–155, 2017.

[Max19] Alexander Maximov. AES MixColumn with 92 XOR gates. *IACR Cryptol. ePrint Arch.*, 2019:833, 2019.

[ME19] Alexander Maximov and Patrik Ekdahl. New Circuit Minimization Techniques for Smaller and Faster AES SBoxes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):91–125, 2019.

[MPL+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 69–88, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[NISa] NIST Lightweight Cryptography Project. https://csrc.nist.gov/projects/lightweight-cryptography.

[NISb] NIST Post-Quantum Cryptography Standardization. https://csrc.nist.gov/Projects/post-quantum-cryptography/Post-Quantum-Cryptography-Standardization.

[NIS01] Advanced Encryption Standard (AES). 2001.

[NIS19] Status Report on the First Round of the NIST Lightweight Cryptography Standardization Process. 2019.

[NMMaS+19] Yusuke Naito, Yasuyuki Sakai Mitsuru Matsui and, Daisuke Suzuki, Kazuo Sakiyama, and Takeshi Sugawara. SAEAES. *NIST Lightweight Cryptography Project*, 2019.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[RPLP08] Carsten Rolfes, Axel Poschmann, Gregor Leander, and Christof Paar. Ultra-Lightweight Implementations for Smart Devices - Security for 1000 Gate Equivalents. In *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, pages 89–103, 2008.

[RTA18] Arash Reyhani-Masoleh, Mostafa M. I. Taha, and Doaa Ashmawy. Smashing the Implementation Records of AES S-box. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):298–336, 2018.

[SHSK19] Jonathan Sönnerup, Martin Hell, Mattias Sönnerup, and Ripudaman Khattar. Efficient Hardware Implementations of Grain-128AEAD. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*, volume 11898 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2019.

[SIH+11a] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An Ultra-Lightweight Blockcipher. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 342–357, 2011.

[SIH+11b] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An Ultra-Lightweight Blockcipher. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 342–357, 2011.

[SMTM01] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

[SS16] Sumanta Sarkar and Habeeb Syed. Lightweight Diffusion Layer: Importance of Toeplitz Matrices. *IACR Trans. Symmetric Cryptol.*, 2016(1):95–113, 2016.

[SSA+07] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and T Iwata. The 128-bit Blockcipher CLEFIA (extended abstract). volume 4593, pages 181–195, 08 2007.

[WMM20] Felix Wegener, Lauren De Meyer, and Amir Moradi. Spin Me Right Round Rotational Symmetry for FPGA-Specific AES: Extended Version. *J. Cryptol.*, 33(3):1114–1155, 2020.

[ZBL+15] Wentao Zhang, Zhenzhen Bao, Dongdai Lin, Vincent Rijmen, Bohan Yang, and Ingrid Verbauwhede. RECTANGLE: A Bit-slice Lightweight Block

Cipher Suitable for Multiple Platforms. *Sci. China Inf. Sci.*, 58(12):1–15, 2015.

Curriculum Vitae

Fatih Balli

E-mail	ballifatih@gmail.com
Nationality	Turkish

Education

2016-2021	**PhD, Computer and Communication Sciences** **Area:** Cryptography (supervised by Prof. Serge Vaudenay) Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
2013-2016	**BSc, Computer Engineering** TOBB University of Economics and Technology (EPFL), Turkey
2010-2015	**BSc, Electrical and Electronics Engineering** TOBB University of Economics and Technology (EPFL), Turkey

Publications

1. *Six Shades Lighter: a Bit-serial Implementation of the AES Family.* Sergio Roldán Lombardía, Fatih Balli, Subhadeep Banik, JCEN 2021
2. *Melting SNOW-V: Improved Lightweight Architectures.* Andrea Caforio, Fatih Balli, Subhadeep Banik, JCEN 2020
3. *The Area-Latency Symbiosis: Towards Improved Serial Encryption Circuits.* Fatih

Balli, Andrea Caforio, Subhadeep Banik, TCHES 2021

4. *Energy Analysis of Lightweight AEAD Circuits.* Andrea Caforio, Fatih Balli, Subhadeep Banik, CANS 2020

5. *Determining the Core Primitive for Optimally Secure Ratcheting.* Fatih Balli, Paul Rösler, Serge Vaudenay, ASIACRYPT 2020

6. *Swap and Rotate: Lightweight Linear Layers for SPN-based Blockciphers.* Subhadeep Banik, Fatih Balli, Francesco Regazzoni, Serge Vaudenay, ToSC 2020 Vol. 1

7. *Exploring Lightweight Efficiency of ForkAES.* Fatih Balli, Subhadeep Banik, INDOCRYPT 2019

8. *BioID: A Privacy-Friendly Identity Document.* Fatih Balli, F. Betul Durak, Serge Vaudenay, Security and Trust Management 2019

9. *Six Shades of AES.* Fatih Balli, S. Banik, AFRICACRYPT 2019

10. *Distributed Multi-Unit Privacy Assured Bidding (PAB) for Smart Grid Demand Response Programs.* Fatih Balli, Suleyman Uludag, Ali Aydin Selcuk, Bulent Tavli, IEEE Transactions on Smart Grid 2017

11. *Privacy-Guaranteeing Bidding in Smart Grid Demand Response Programs.* Suleyman Uludag, M. Fatih Balli, Ali Aydin Selcuk, Bulent Tavli, IEEE GC 2015 Workshop in SG Resilience

12. *Enhanced Duplication: a Technique to Correct Soft Errors in Narrow Values.* I. Burak Karsli, Pedro Reviriego, M. Fatih Balli, Oguz Ergin and J. A. Maestro, IEEE Computer Architecture Letters, 26 April 2012

Programming/Typesetting Languages

C, Python, Java, SageMath, VHDL/Verilog, x86, RISC-V, Matlab/Octave, HTML, PHP, Latex

Languages

Turkish (native), English (advanced), French (A2-B1)

Experience as Teaching Assistant

2019–2020 Fall 2018–2019 Fall	Cryptography and Security
2019–2020 Spring 2017–2018 Spring	Advanced Cryptography
2020–2021 Fall 2017–2018 Fall	Computer Architecture
2016–2017 Spring	Remedial Review Course
2016 Summer	Internet and Data Security
2015–2016 Spring	Introduction to Cyber Security