# Secure cryptographic implementations in hardware: A delicious adventure

ISBN: 9789464193312
Typeset using LaTeX

Cover background and images by: annapustynnikova. Adobe stock.
Cover design: Camila Torrano

Radboud University



Applied and Engineering Sciences

This work is part of the research programme TYPHOON with project number 13499, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

**Secure cryptographic implementations in hardware: A delicious adventure**

Proefschrift
ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op maandag 8 november 2021
om 16:30 uur precies

door
Pedro Maat Costa Massolino
geboren op 04 maart 1988
te São Bernardo do Campo – Brazilië

**Promotor**

Prof. dr. L. Batina

**Manuscriptcommissie**

Prof. dr. E.R. Verheul
Prof. dr. Máire O'Neill (Queen's University Belfast, Verenigd Koninkrijk)
Prof. dr.-ing. Tim Güneysu (Ruhr-Universität Bochum, Duitsland)
Prof. dr. Francisco Rodríguez-Henríquez (CINVESTAV 3-D, Mexico)
Dr. Gilles Van Assche (STMicroelectronics, België)

# Acknowledgements

"Why the Netherlands?" "Why did you choose to go so far from Brazil?" "Isn't the weather better there?". The answer to all those questions is also a question, *Why not?* After finishing my Masters in Brazil, I started looking for my next adventure. During this search, I found one opportunity on the IACR jobs list: pursuing a PhD in the Netherlands, more precisely at Radboud University with prof. Lejla Batina. Until that moment, I have never been to the Netherlands or had any idea what kind of language they spoke or even food they ate. Thus, instead of searching for reasons to go, I tried to find the opposite, reasons not to go. I could not find any strong reason not to try; after all I could always go somewhere else and try again. When we switch from finding motivation to act, to finding motivation not to act, then we are more open to new experiences, new adventurous, to fall, and thus to live. This was my main motivation for pursuing the PhD: I was looking for a new life experience.

During all those 5 years, I have met many people from many different places, and for different amounts of time. Some people I just saw very briefly, maybe in conferences or short visits, while others I had the pleasure to see for a longer time, as they did a full PhD, were post-docs, or longer visitors etc. While I could try to explicitly write some people's name to thank them, most likely I will end up forgetting someone. Therefore, for everyone that I shared some good or bad moments within those 5 years, thank you very much. Thanks to all the experiences and enjoyable moments, I was able to have the energy and motivation to continue pursuing my PhD.

A PhD is not only made by experiences, but also by the hosting institute, so I would like to thank Radboud University, the Faculty of Science and its Digital Security department in particular for having me as a PhD student. I would like to thank my co-workers and office mates in the Digital Security department and in the Mercator 1 building for the very great environment and coffee breaks with very nice sweets. And a special thanks to my co-authors, which without their work this thesis would not have been made possible. Finally, I would like to thank the colleagues at my two internships, at Riscure in Delft and STMicroelectronics in Milan, the time spent was pretty good and insightful.

This work was only possible thanks to donations received in form of a grant, equipment and tools. First, I would like to thank NWO, the European Commission, and Radboud University for their monetary support during my PhD. I would also like to thank Xilinx for the University tools licenses and the FPGA boards that were donated.

But not everything is work related, so I would like to thank to the Factorio game

group, the incredible Friday nights with nice movies, Rick and Morty and rice cooker facts, and the Tuesday (Wednesday) movie nights. I also would like to thank Barış and Rachel for organizing the board game nights. Because of them I had strength to continue my work. I also want thank for the people that organized and were part of other small events like: Goulasch night, beach volleyball, pancake boat, escape room, paintball, bouldering, ice skating and so on. Finally, I want to thanks the researchers and friends I met at workshops and conferences, which not only helped with nice discussions, but also with enjoyable moments.

I would like to show special gratitude to prof. Peter Schwabe for the very nice discussions, opportunities, barbecues and printing out my recommendation letter to Lejla, which made it possible for me to do the PhD. I also want to show special gratitude to prof. Joan Daemen for also the nice discussions, opportunities, barbecues and helping me a lot in my research. I also want to thank prof. Lejla Batina in particularly for almost everything: discussions, opportunities, barbecues, etc. In summary, having me as PhD student and guided me all these 5 years.

I would like also give a special thanks for Joost Renes for sharing the office with me since first day, the trips and lots of nice moments/bars that we spent together. I also would like to give special thanks to Veelasha Moonsamy for the nice trips and funny moments/bars with me and Joost Renes, and also for all the help during my stay in Nijmegen.

I would like to give a very heartfelt thanks to my parents, Pedro Tadeu and Aurea Luzia, which without their support I would probably not even reached anywhere close to a thesis. Thank you very much. I also want to give special thanks to my sister Caroline and her husband Napoleão for receiving me in Rugby and helping me during my moving from Netherlands to the UK.

I also want to give special thanks to PQShield and Ali El Kaafarani, for allowing me to spend time at my thesis, when I should have finished it already.

Thank you very much to everyone who read and reviewed the previous version of this thesis : Ali El Kaafarani, Anna Guinet, Bas Westerbaan, Fabio Campos, Lejla Batina, Jon Moroney, Matthias J. Kannwischer, Thom Wiggers, and Veelasha Moonsamy. Also, a special thanks for Bas Westerbaan for translating to me the summary in Dutch.

Finally, I want to acknowledge the “algorithm”, thanks to it I was able to “understand” Dutch.

Pedro Maat Costa Massolino
Nijmegen, the Netherlands, Earth, 2020
Rugby, England, Earth, 2021.

# Contents

# List of Figures

# List of Tables

# List of Symbols

| | |
|---|---|
| $\mathbb{F}_q$ | Finite field of order $q$ |
| $\mathbb{Z}$ | Set of integers |
| $\mathbb{Z}_q$ | Set of integers modulo $q$ |
| $\lceil x \rceil$ | Ceiling approximation of $x$ to an integer |
| $\lfloor x \rfloor$ | Floor approximation of $x$ to an integer |
| $\|a\|$ | Bitlength of $a$ |
| $\log_n$ | Logarithm on base $n$ |
| $\gcd(a, b)$ | Greatest Common Divisor of $a$ and $b$ |
| $\oplus$ | EXclusive-OR (XOR) operation |
| $\wedge$ | AND operation |
| $\vee$ | OR operation |
| $\lll$ | Rotate Left |
| $\ll$ | Shift Left |
| $a\|\|b$ | Concatenation of the bytearrays or strings $a$ and $b$ |

# Introduction

While cryptography is not delicious or advised to be tasted, the stories around it are full of adventures: wars, romance, revolution, etc. In short, *secrecy* is one of the main requirements motivating the use of cryptography and keeping information private or to be shared with only those we trust. Thanks to the advances in electronics and the democratization of the means of communication, computation and information storage, cryptography became an essential part of our daily lives around the end of the 20th century. Nowadays, cryptography is not only used to keep information secret but also to sign or prove someone's identity, which helps to provide services like internet banking and shopping, public transport cards and online document attestation. During this democratization, cryptography also changed its approach from secret-based algorithms to public algorithms with secret keys, as proposed by Auguste Kerckhoffs [69] which became part of Kerckhoffs' principles. Public algorithms have the advantage of being evaluated by anyone without leaking any secret information, while secret-based algorithms can only be analyzed by those with access to the information. Given the larger amount of people that can review them, public algorithms are less likely to have errors or leakages than the secret-based counterpart.

Cryptography was not the only beneficiary of the electronics advancements in the modern era, but also the field of computer engineering in general and in particularly in hardware design. In hardware design, a machine, be it mechanical, electronic and/or any other technology is built to solve a certain problem. However, engineering and building such machines is usually a process that requires lots of finances and time, that is not available most of the time[1]. With the advent of microelectronics, the cost of building specialized machines could be spread thanks to the easiness and speed at which the machines could be produced. Therefore, hardware design focused more on building generic machines that could be programmed to carry out a wide array of tasks, and subsequently, reducing the non-recurring costs incurred with production volume. However, some tasks still have very tight requirements regarding execution time, data throughput or operating conditions which makes it necessary to engineer and produce specialized hardware for them. One such example is cryptography.

Cryptography is usually executed or accelerated in hardware because of imposed short execution timings, high amounts of data and/or environments with passive and active attackers whose aim is to recover internal secrets. Video streaming is an example for high amounts of data that have to be encrypted, or decrypted, in a short

[1]For instance the first programmer, Ada Lovelace, wrote algorithms [82, 51] for the mechanical computer "Analytical Engine" by Charles Babbage, because of its costs and other difficulties it was never finished [51].

time frame. The amount of data that has to be processed can easily reach the order of megabytes in a hundredth of a millisecond. Therefore, hardware accelerators are usually employed to handle this. Other use cases with very short execution time but low amounts of data, are public transport card authentication and highway toll booths. In both cases, the entire authentication including communication should be executed in less than a second on a cheap device. In order to reduce costs per device and have a very small execution time, usually custom hardware is employed. In the special case of cheap and/or low power devices, hardware accelerated cryptography is more common, not only because of the time and memory requirements, but also to have one device that can be employed in as much applications as possible. In other cases, the information in a device might be of very high value, such as bank cards, which not only have to authorize operations when used by its owner, but also not reveal any sensitive information when being monitored via secondary channels (side-channels) or physically attacked (fault attacks). Examples of side-channels include power, electromagnetic, temperature or sound emissions, while fault attacks can be done through power supply glitches, electromagnetic pulses or laser pulses. In such cases, not only must the cryptographic algorithm be implemented in hardware, but also the countermeasures against these kind of attacks.

Given that hardware implementations of cryptography are widely used and deployed in different environments, the study and evaluation of them is of prime importance. When we study and evaluate cryptographic primitives, we provide results and tools to designers to create faster and cheaper cryptographic primitives in hardware.

One very common hardware evaluation tool and platform is a Field Programmable Gate Array (FPGA). FPGAs can naively be explained as an array of interconnected memories, where each element behaves as look-up tables for logic, memory elements for storage or control mechanism for the internal routing. The most common memory technology employed is Static Random-Access Memory (SRAM), because it has low-latency reads and writes. By employing a fast internal memory, the entire system can operate at high frequencies. One disadvantage of SRAM or other volatile technologies, is that they require an external (or internal) non-volatile memory, like Flash, to store the FPGA state to be loaded during boot up. While the device was originally made for digital circuit simulation, evaluation or prototyping, FPGAs became faster over the years and started being used as a replacement for custom digital circuits, usually in low volumes products or when it is required to update the design in the field. More recently, they are also being deployed in data centers to accelerate various types of computations, such as artificial intelligence or big data. In those cases, since data is processed in the FPGA, thus it is intuitive to add the cryptographic primitives.

"Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication", according to Menezes et al. [92]. *Confidentiality*, or secrecy, is the act of keeping the information enclosed to only those who should have access to it. With *data integrity* one knows if the information has been modified by a non-authorized party. *Entity authentication* is the act of proving one's identity. Finally, *data origin authentication* extends the idea of entity authentication to also prove that a certain piece of data comes from a certain identity.

*Cryptographic primitives*, according to Menezes et al. [92], are tools that can be directly used or combined to provide one or more information security aspects. The

primitives can then be subdivided into three types : *public key*, *symmetric key* and *unkeyed*.

Public-key (or asymmetric-key) cryptography it is the study of constructing cryptographic primitives with two keys, a public and a private one. While the public key can be shared with any other party, the private key is meant to only be accessible by the owner. The security of this type of cryptosystem relies on the assumption that some mathematical problems are hard, such as integer factorization [112], discrete logarithm on finite fields [45]/elliptic curves [94, 71] and finding isogenies between supersingular elliptic curves [50].

Symmetric key (or secret-key) cryptography on the other hand has a single key for all the parties involved and usually is not based on the mathematical problems as the ones in public-key cryptography. Secret-key cryptography is orders of magnitude faster than public-key cryptography in most scenarios and applications. Therefore secret-key cryptography is regularly applied to encryption and authentication of the data, while public-key cryptography is used to exchange or encapsulate the required symmetric keys, or to sign data.

Unkeyed cryptography is the study of primitives that do not have any key in their construction. Examples of such primitives are hash functions and the eXtended Output Function (XOF). Hash functions take as input a bit string of any size and output a fixed size bit string, called the digest where length is fixed and defined by the function. An XOF can be seen as an upgrade of the hash function, as it allows a user to pick the desired length of the digest. A hash and XOF can be used to provide unique identification or summaries of messages. In the case of XOFs, they can also be used directly as a seed expander where a small seed generates more bits. Because of the limited output size of the hash and XOF, it is theoretically possible to find a second input that gives the same output, this is called *second preimage*, and a *collision* is for the case of finding the same output for any input. Even if the output length of a XOF is increased, the problem persists, as a XOF can only internally store a limited amount of bits, which makes it a bottleneck for possible collisions or second preimages. Unkeyed cryptography shares lots of constructions and concepts from symmetric cryptography, and thus it is more common to see unkeyed primitives being studied in symmetric cryptography.

Examples of symmetric-key primitives are authenticated encryption (AE), authenticated encryption with associated data (AEAD), session authenticated encryption (SAE) and message authentication codes (MAC). An AE is a primitive that is able to encrypt data and provide a *tag* that makes it possible to detect if the original/encrypted data has been changed by someone who does not have access to the key. This type of primitive can require extra inputs, such as a number only used once, *nonce*, or a *distinguisher*. A nonce is a unique public number that might be required for encryption and it should never repeat. If the nonce is repeated with the same keys, some algorithms might end up leaking the original data and/or loosing the authenticity guarantees, but there are other algorithms which are robust to nonce repetitions. A *distinguisher* follows the same idea as a nonce, however as the name implies it can be repeated with different keys. *Associated data* is public data that needs to be authenticated with the encrypted data, for example a header in a communication packet. When performing AE some schemes support having an optional associated data, while some others require the associated data, which are called AEAD. The

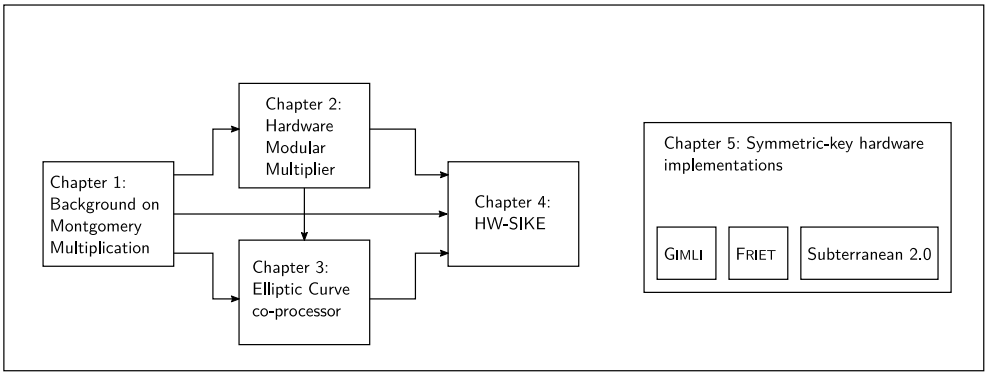
SAE primitive is an AE primitive which adds the concept of a session, where multiple messages can be encrypted and respective tags generated, thus not only protecting a single message, but multiple messages and their order of arrival. Finally, a MAC is used when one desires to only authenticate the data, but not encrypt it.

Examples of public-key cryptosystems types are key exchanges, key encapsulation mechanisms (KEM) and digital signatures. A key exchange is used when two parties try to obtain one common shared secret, which is typically used as a symmetric key. A key exchange is interactive or non-interactive [27]. A non-interactive key exchange requires only one message to be shared between parties in a non-synchronized way, thus both parties can simultaneously start the protocol. One classical example of this is the Diffie-Hellman (DH) protocol [45]. When the key exchange is interactive, one of the two parties initiates the protocol and then waits for the second party in order to finish the computation and reach a shared key. An interactive key exchange, can be created with a KEM. Here one party starts by generating a keypair and then sharing the public key to the other party. These keys can be generated before hand, thus they can be reused for other rounds and parties. The second party can then encapsulate/encrypt a symmetric key through the obtained public key and then send the encapsulated symmetric key to the first party. The party that generated the public and private keys can then decapsulate/decrypt the symmetric key, and thus both parties have a common symmetric key. Finally, a digital signature scheme also has a public and private key, in which the private key is used to sign a message (typically its hash), and the public key is used to verify whether the signature is coming from the original message and public key.

Public-key cryptography can be sub-categorized into the mathematical framework that is used to construct the primitives. A few examples are: finite field cryptography [11], elliptic-curve cryptography (ECC) and isogenies over supersingular elliptic curves. Finite field cryptography operates over prime modular integer exponentiation, with primes of more than 2048 bits [11]. One example of finite field cryptography is the original Diffie-Hellman (DH) protocol [45]. Elliptic curves are a mathematical framework that were proposed for cryptography independently by Victor Miller [94] and Neal Koblitz [71]. ECC-based algorithms can be applied as a direct replacement of many protocols based on finite field cryptography, examples are the elliptic curve DH (ECDH) and elliptic curve Schnorr signatures. Elliptic curves are also smaller and faster than finite field cryptography, since they can operate at similar security levels with 256 bit integers or smaller. The discrete logarithm and the integer factorization problem are assumed to be hard for classical computers, but they can be solved in polynomial time and space by quantum computers through Shor's algorithm [125]. All cryptographic algorithms that are secure against classical and quantum computers are called a *post quantum algorithm* [29]. Because most of the elliptic curves primitives are based on a discrete logarithm, they are not post-quantum, but one other type that still is in the same mathematical framework is called isogenies over supersingular elliptic curves. They can be used to create a key exchange called SIDH [50] and a KEM called SIKE [8]. Chapter 3 and Chapter 4 have a more formal definition on elliptic curves, but Chapter 4 is mostly focused on supersingular elliptic curves.

In this thesis, I describe asymmetric cryptographic implementations with elliptic curves and isogenies between supersingular elliptic curves on FPGAs. I also explain and compare three different proposals for authenticated encryption, that are each

Figure 1: Chapters relationship.



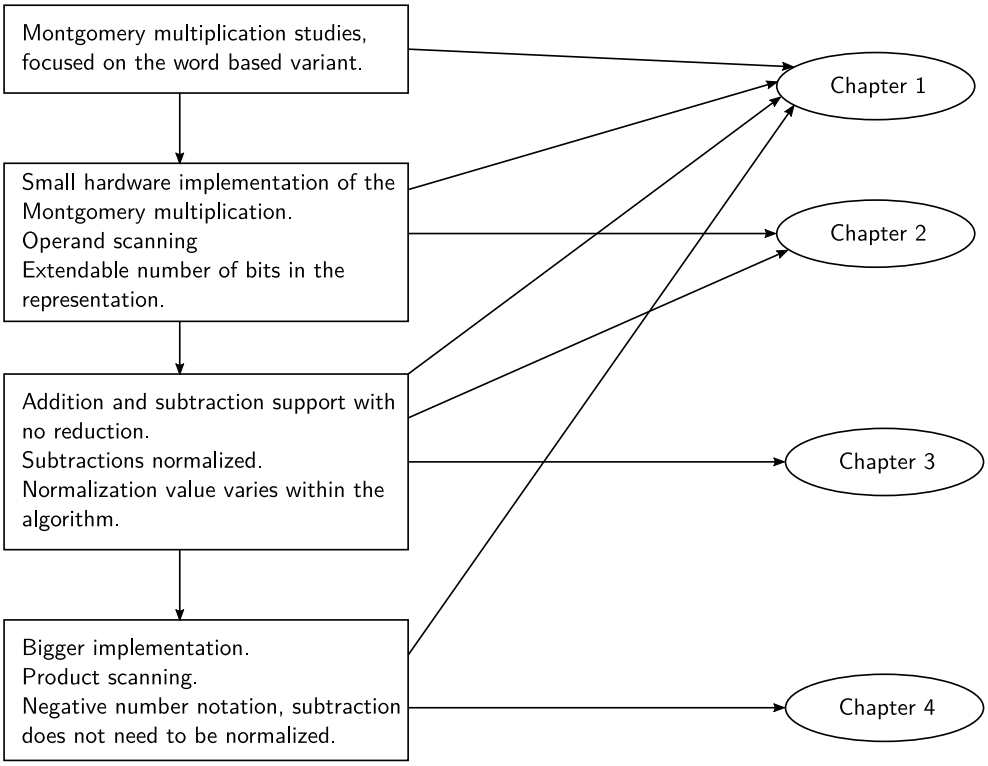

tuned for different environments and constraints.

The organization of this thesis into chapters and their relationship are illustrated in Figure 1. The background on Montgomery multiplication, Chapter 1, is related to Chapters 2, 3 and 4. The work on the Montgomery multiplier was the basis for the elliptic curve scalar multiplication in Chapter 3. Chapter 4 does not directly use the works of Chapters 3 and 2, but it was from their experience and knowledge that Chapter 4 was built. Chapter 5 is isolated from the others chapters, since the other chapters focus mostly on hardware implementations for public-key. Chapter 5 is different from the other chapters, because the research and collaboration for these papers only showed in the middle of the PhD. This chapter illustrates an important aspect of research, which is being able to work with different and well related topics in collaboration with other as they come up.

As mentioned before, Chapters 1, 2, 3 and 4 have a natural flow as they describe the progress and evaluation around elliptic curve cryptography implementation. Figure 2 elaborates on this progress. At the beginning there was an initial study on Montgomery multiplication itself and how to implement a small optimized version of it, which led to the creation of Chapters 1 and 2. As time progressed the idea evolved into making an elliptic curve scalar multiplication co-processor, that culminated on the design of Chapter 3. However for the scalar multiplication, it was required to have addition/subtraction operations and make some small adjustments on the Montgomery multiplier, which are shown in Chapters 1 and 2. The idea to implement SIKE appeared after the publication of Chapters 2 and 3, thus it was possible to revisit the entire design. Therefore, Chapter 4 adopted a signed notation to avoid the normalization step after subtractions, a bigger multiplier to increase performance, and a product scanning algorithm instead of operand scanning. These changes influenced Chapter 1, since it now had to cover those design choices.

Chapter 5 does not connect with the other chapters, but the research path still is hardware implementation of cryptographic primitives. The chapter starts with the Gimli permutation implementation, which has the first ideas of a common interface and framework to implement and evaluate permutations. Those ideas were not new and have been proposed before the Gimli permutation research [61]. After Gimli permutation implementation was done, some of the ideas (such as the interface) were

Figure 2: Progress over time and their impact on Chapters 1, 2, 3 and 4.



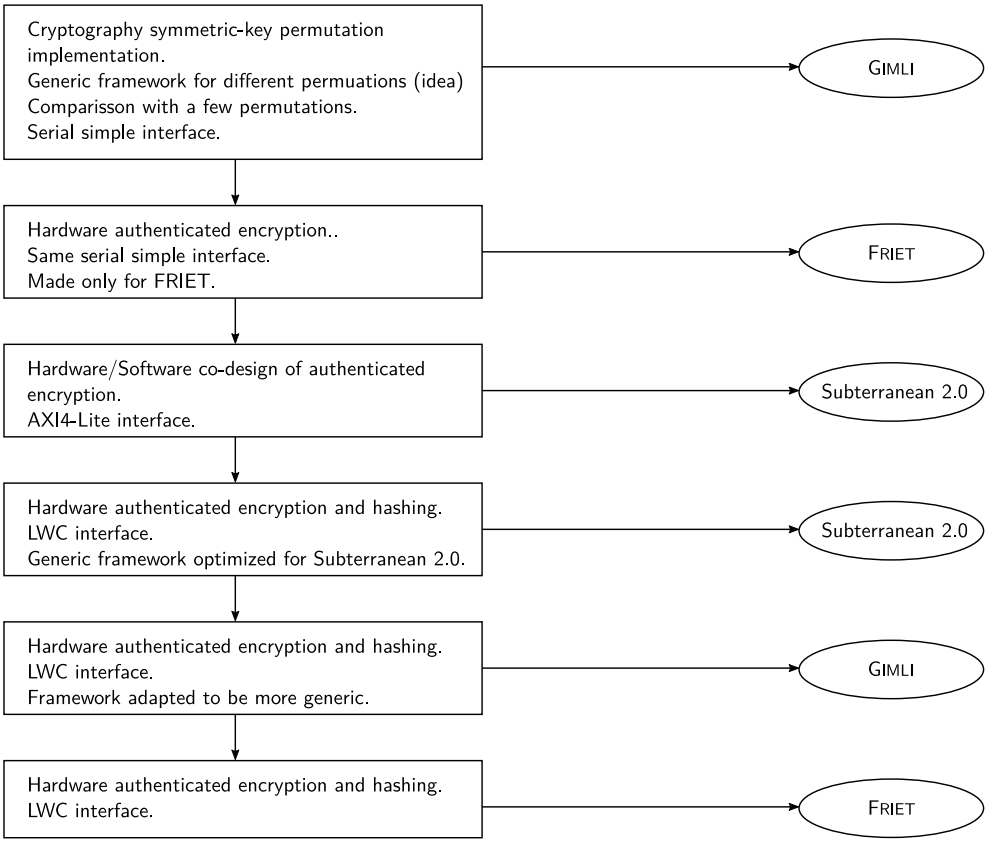

used to make the implementation of the authenticated encryption Friet. Because the authenticated encryption Friet does not only have a permutation its structure is more complex, thus it is not easy to adapt to other authenticated encryption schemes. The first Subterranean 2.0 design adopts a new strategy, a hardware/software co-design and a AXI4-Lite interface to replace the custom interface from previous designs. Those ideas later helped during the creation of a generic framework for the subsequent Subterranean 2.0 hardware implementation of authenticated encryption and hash, with the LWC interface [67]. The first version of the framework was more tuned to Subterranean 2.0, but when adapting to the authenticated encryption and hashing from Gimli it was easier to adapt to other permutations, as was done for Friet.

## Contributions

### Hardware Montgomery (Chapter 2)

Elliptic curve cryptography and public-key cryptography requires some kind of modular arithmetic, with modular multiplication as one of the main operations. Among modular multiplication algorithms for generic primes, Montgomery multiplication is one of the most commonly used. The algorithm is seen on a wide array of archi-

Figure 3: Chapter 5 progress along the time.




tectures, from large systolic arrays to iterative datapaths. Iterative datapaths are commonly used in scenarios where the power and/or the energy are restricted, and when combined with a low power technology it becomes an interesting embedded solution.

In Chapter 2, we aim for a modular multiplier with an iterative datapath, thus we design and evaluate two Montgomery multiplication word-based algorithms that are called CIOS and FIOS by Koç et al. [72]. Each algorithm is implemented with its own architecture, while exploiting the DSPs and memory modules of the IGLOO2 FPGA. As a result, we produced a very compact and competitive architecture for Montgomery multiplication in IGLOO2 FPGA. This was published in the following publication:

Pedro Maat C. Massolino, Lejla Batina, Ricardo Chaves, and Nele Mentens. "Area-optimized Montgomery multiplication on IGLOO 2 FPGAs". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056762. URL: https://dx.doi.org/10.23919/FPL.2017.8056762

**Contribution.** I am the main author of the aforementioned publication. The multipliers were designed by all authors, while the implementation, evaluation and testing was only myself.

### Elliptic curve co-processor (Chapter 3)

After designing a modular multiplier, the next step is to implement a basic operation for a public-key algorithm, such as scalar multiplication for elliptic curves, which is the basis for ECDH and ECsignatures. Therefore it is essential to evaluate scalar multiplication with different hardware architectures and elliptic curves addition formulas.

In Chapter 3, we combine a parallel implementation with the Montgomery multipliers from Chapter 2 and Renes et al.'s [111] Weierstrass complete addition formulas. Even though the cited elliptic curve formulas have been parallellized from two up to six multiplier units, our proof-of-concept hardware only employs three units. This chapter is based on the following paper:

Pedro Maat C. Massolino, Joost Renes, and Lejla Batina. "Implementing Complete Formulas on Weierstrass Curves in Hardware". In: *Security, Privacy, and Applied Cryptography Engineering. SPACE 2016*. Vol. 10076. Cham: Springer International Publishing, 2016, pp. 89–108. DOI: 10.1007/978-3-319-49445-6_5. URL: https://doi.org/10.1007/978-3-319-49445-6_5

Chapter 3 extends the work that has been done in the above paper by explaining how to fix a mistake that was found in the implementation.

**Contribution.** I am the main author of the aforementioned publication. In the publication, I designed, evaluated and tested the hardware implementation of the elliptic curve co-processor. The Montgomery multiplier was adapted from Chapter 2, which was joint work with my co-authors of that paper.

### SIKE (Chapter 4)

While the previously studied elliptic curves protocols can perform key exchange and signatures that are safe against classical computers, they are not safe against very large quantum computers. For this reason and to broaden the number of primitives, the USA National Institute of Standards and Technology (NIST) has launched a competition to choose public-key primitives that are safe against very large quantum computers. Part of the process is to benchmark these primitives in a wide array of platforms.

In Chapter 4, I showcase the largest project of this thesis: the implementation of the Supersingular Isogeny Key Encapsulation mechanism (SIKE) [8]. Because of its size and complexity, the SIKE architecture was implemented as a hardware/software co-design. This also allows for more trade-offs. A traditional hardware/software co-design uses a generic CPU together with one or more co-processors, however the CPU in this project was custom-made to run the SIKE algorithm. It also uses a co-processor that is different from the one in Chapter 2, and is called `Carmela`. It applies a different Montgomery multiplication algorithm than in the ones used in Chapters 2

and 3, and is also optimized for Montgomery-friendly primes. The entire project was published in:

Pedro Maat C. Massolino, Patrick Longa, Joost Renes, and Lejla Batina. "A Compact and Scalable Hardware/Software Co-design of SIKE". in: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.2 (Mar. 2020), pp. 245–271. DOI: 10.13154/tches.v2020.i2.245-271. URL: https://tches.iacr.org/index.php/TCHES/article/view/8551

**Contribution.** I am the main author of the aforementioned publication. In the publication, I designed, evaluated and tested the hardware and software parts of the SIKE core.

## Symmetric implementations (Chapter 5)

Public-key cryptography is not the only type of cryptosystem that is employed in real world, there is also symmetric cryptography, and more recently the so-called lightweight symmetric schemes came into focus. Lightweight symmetric schemes makes some trade-off between security and/or performance in order to make them fit into resource constrained devices/scenarios. In this chapter, I will talk about implementing in hardware three lightweight proposals: GIMLI, FRIET and Subterranean 2.0.

GIMLI is a permutation with state of 384 bits (a lot smaller than in SHA-3) with good performance in software and hardware. I present two different architectures for the GIMLI permutation: a round-based architecture and a serial-based architecture. I also explain the full hash (GIMLI-24-HASH), and the AE (GIMLI-24-CIPHER) architectures that are based on the round architecture, which are not part of the original publication. This work was published in:

Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. "Gimli : A Cross-Platform Permutation". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Cham: Springer International Publishing, 2017, pp. 299–320. DOI: 10.1007/978-3-319-66787-4_15. URL: https://dx.doi.org/10.1007/978-3-319-66787-4_15

**Contribution.** In the publication, I designed, evaluated and tested the hardware architecture for the GIMLI permutation in FPGA. The ASIC evaluations were done by the co-authors. The full hash and authenticated circuit is an unpublished contribution of my own.

Friet-P is a 384-bit permutation with 128-bit parity, thus 512 bits in total that is capable of detecting a single fault. Friet scheme is built on top of the permutation Friet-P by using it inside of the sponge duplex construction. After introducing the permutation and its mode of operation, I discuss the hardware architecture and results for FRIET. This work was published in:

Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat C. Massolino, Kostas Papagiannopoulos, Francesco Regazzoni, and Niels Samwel. "Friet: an Authenticated Encryption Scheme with Built-in Fault Detection". In: *Advances in Cryptology – EUROCRYPT 2020*. Vol. 12105. Cham: Springer International Publishing, 2020, pp. 581–611. DOI: 10.1007/978-3-030-45721-1_21. URL: https://doi.org/10.1007/978-3-030-45721-1_21

**Contribution.** In the publication, I designed and evaluated the hardware implementation for FRIET and FRIET-P in the FPGA. While the ASIC design was also done by me, the evaluation was done by Francesco Regazzoni.

`Subterranean 2.0` is a cipher suite based on the original primitive called Subterranean by Daemen [39] with a new mode operation for hashing and authenticated encryption. The cipher suite has the same primitive function as in Daemen [39], but with new input and output location in the state to construct the permutation. The cipher suite also applies the sponge duplex mode on top of the permutation, in order to offer three cryptographic schemes: an extendable-output function (XOF), a session authenticated encryption (SAE) and a message authentication code (MAC). After presenting Subterranean 2.0, I describe the hardware architecture for the permutation and the design choices that led to a hardware and software co-design architecture. I also explain the Subterranean 2.0 hardware architecture that is capable of hashing and authenticated encryption. Subterranean 2.0 work was published in the following paper:

Joan Daemen, Pedro Maat Costa Massolino, Alireza Mehrdad, and Yann Rotella. "The Subterranean 2.0 Cipher Suite". In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (June 2020), pp. 262–294. DOI: 10.13154/tosc.v2020.iS1.262-294. URL: https://tosc.iacr.org/index.php/ToSC/article/view/8622

**Contribution.** In the publication, I designed and evaluated the hardware implementation for Subterranean 2.0 in FPGA and ASIC. In addition to the published contribution, I added in this thesis the architecture that is capable of performing hashing and authenticated encryption.

# Chapter 1

# Background on Montgomery multiplication

Public and symmetric key cryptography require as mathematical structure a finite group or fields for their internal computation. When implementing the arithmetic for the groups or fields used in cryptography, one of the main components is the modular multiplication, which is a multiplication followed by a modular reduction. The multiplication and modular reduction demand a substantial amount of resources, but for the moment we will focus on the modular reduction. The most employed algorithms for modular reduction for a modulus bigger than 2 and with no exploitable structure are Barret reduction [12], Montgomery reduction [97] or some variant of the previous two. Both methods have advantages and disadvantages, but at least in my perception it is more common to see papers employing Montgomery than the Barret method. Examples of cryptosystems where Montgomery reduction/multiplication is deployed are RSA implementations [13], integer-based Diffie-Hellman, elliptic-curve cryptography [4], lattices-based cryptography [3] or isogeny-based cryptography [77].

**Chapter organization.**

This chapter starts by explaining the notation and Montgomery reduction in Section 1.1. Then, Section 1.2 explains Montgomery multiplication and a few selected variants. An optimized version of the methods from Section 1.2 for primes with a certain structure, called "Montgomery-friendly", are then discussed in Section 1.3. Finally, Section 1.4 gives some final considerations and points to how Montgomery multiplication has been used in the next chapters.

## 1.1 Montgomery reduction

The notation in this chapter is as follows. Values are defined by lowercase letters, such as $x \in \mathbb{Z}$ is an integer element. When values have a subscript they are being interpreted as polynomials over the powers of $2^{ws}$, thus $x = x_0 \cdot 2^{0 \cdot ws} + x_1 \cdot 2^{1 \cdot ws} + ...$, with each $x_i$ being between $[0, 2^{ws} - 1]$, except the last $x_i$ which has values between $[-2^{ws-1}, 2^{ws-1} - 1]$ . When values are always positive, $y \in \mathbb{Z}^+$, then the last $y_i$ is

between $[0, 2^{ws} - 1]$. This double interpretation mimics one of the ways computers store and handle integers bigger than their internal word size ($ws$). The double vertical bar "||" is used as a concatenation of values with the most significant part on the right side and the less significant part on the left. The concatenation is applied when an operation that would overflow a single element is able to fit into two elements, thus the result is split between the two. The variable *null* is used when part of the output is ignored.

The Montgomery reduction algorithm [97] performs modular reduction by changing the target operation $o \leftarrow t \pmod p$ to $o \leftarrow t/r \pmod p$, where $0 < t < r \cdot p$ with $r, p \in \mathbb{Z}, r > p > 1$ and that $r$ is coprime to $p$, thus the greatest common divisor $\gcd(p, r) = 1$. This change, while counter-intuitive due to the extra division by $r$ is in fact cheaper because all operations, multiplication, division and modular reduction are done over $r$, and if $r$ is chosen as a power of 2, thus $r = 2^{re}$ and $re \in \mathbb{Z}^+, re = \lceil log_2(p) \rceil$, then it is easier to compute on binary systems. During the computation, two values are required $p' \in \mathbb{Z}$ between $[1, r - 1]$ and the inverse of $r$ modulus $p$ denoted by $r^{-1}$, both can be computed by the following equation:

$$r \cdot r^{-1} - p \cdot p' = 1$$

Montgomery reduction, as shown in Algorithm 1, starts with a value $t$ between $0 \leq t \leq r \cdot p$. The first step is to multiply "t" by $p'$ modulo $r$, thus giving us $m$. Then, we multiply $m$ by $p$ itself which is added to $t$ and then divided by $r$. Accordingly to Montgomery [97], this works because:

$$t + m \cdot p = t + t \cdot p' \cdot p = t + t \cdot (-p^{-1}) \cdot p = t - t = 0 \pmod r$$

The last step in the Montgomery reduction is a conditional subtraction, that is computed in order to ensure that the output is within $[0, p]$, since:

$$0 \leq t + m \cdot p < r \cdot p + r \cdot p$$
$$0 \leq o < 2p$$

**Algorithm 1** Montgomery reduction algorithm [97].

**Require:** $0 \leq t \leq r \cdot p$, $\gcd(p, r) = 1$, $r = 2^{re}$, $re = \lceil log_2(p) \rceil$, $p \cdot p' = -1 \pmod r$
**Ensure:** $o = t/r \pmod p$
1: $m \leftarrow p' \cdot t \pmod r$
2: $o \leftarrow (t + m \cdot p)/r$
3: **if** $o \geq p$ **then**
4: **return** $o - p$
5: **else**
6: **return** $o$
7: **end if**

If implemented exactly as described in Algorithm 1, Montgomery reduction is known to leak due to side-channels in the conditional subtraction in step 3 [74] [44]. In order to avoid this conditional subtraction, one option is to increase the value of $re$ by 2, so $re = \lceil log_2(p) \rceil + 2$, which makes $r > 4p$, thus $t < 4p^2$. According to

Walter [136] [137], this is enough to guarantee the output of step 2 in Algorithm 1 will be within $[0, p]$.

Because 0 and $p$ are both valid outputs of the Montgomery reduction, one should be careful when using the output $o$. If the output is going to be used for any arithmetic procedure modulo $p$, then values 0 and $p$ are the same modulo $p$, thus it can be used directly. The problem occurs if the output is applied into a cryptographic hash function, such as SHA-3, or any other function that receives the value as a bit string. In those cases, the bit string representation of the integer $p$ will differ from the representation of 0. Therefore, it is necessary to apply a conditional subtraction that is side-channel safe in order to ensure that all values are between $[0, p-1]$. For some cryptosystems, an output 0 or $p$ would never occur because they are not valid, such as in an integer exponentiation of RSA, but it may happen in elliptic curve computations depending on the finite field and coordinate system. If the coordinate system in an elliptic curve has valid points with 0 in one of the dimensions, then the conditional subtraction will be necessary. As a rule of thumb, a 0 in one of the dimensions in an elliptic curve defined over a prime field is associated with the neutral point, which is not a problem since the neutral point most likely will not be used to feed a cryptographic hash function. If the curve is defined over an extension field instead of a prime field, then it may happen that one of the parts of an element in the extension field has the value 0.

It is possible to skip a modular reduction during an addition or a subtraction that happens before the multiplication and reduction by increasing the value of $re$. In this case we take an extra variable $ba \in \mathbb{Z}^+$, $ba > 2$ and $re = \lceil log_2(p) \rceil + ba$, if $ba = 4$ then we know that $r > 16p$ and thus $t < 16p^2$. This value of $t$ could have been generated by multiplying two values between $[0,\ 4p-2]$, which themselves are the output of an addition of elements between $[0,\ 2p-1]$. This has been proved by Gueron [56] and Batina et al. [13], not only for addition, but for subtractions if you add $2p$ in the operation in order to shift the output from $[-2p+1, 2p-1]$ to $[1, 4p-1]$. If more successive additions/subtraction are necessary, then one can increase the value of $ba$ until the intermediate values do not exceed the representation.

If one increases $ba$ to handle several subtractions before a multiplication, it will be also necessary to handle different multiples of the prime to be sure the output of the subtraction is a positive number. This can be avoided by adopting an internal number representation with negative numbers, thus allowing the numbers to increase in the negative side as well. The adoption of negative numbers does not change Algorithm 1, only the value $t$ is now between $[-r \cdot p, r \cdot p]$. The intermediate value $m$ computed in line 1 of Algorithm 1 continues to be unsigned, thus the output of line 2 will always be an addition. The output bounds when using the negative notation are $[-p, 2p-1]$, as already seen $t + m \cdot p \geq -r \cdot p + 0 \cdot p$, thus $-p \leq o < 2p$.

While we have described the Montgomery reduction algorithm, it is common to see in the literature and this thesis the term *Montgomery multiplication*. Montgomery multiplication just performs a multiplication before applying the Montgomery reduction. While at first glance there is no difference in adding a multiplication before reduction, in practice, the multiplication and reduction are merged together by operating in a bit or a word-level manner. By joining the multiplication and reduction steps, it is possible to reduce the required amount of memory to a constant plus the input size, but if separated, one would need at least the double the input size for mul-

tiplication. Another advantage of operating in a word-level manner is the easiness to exploit some structure in the primes to skip some steps.

## 1.2 Word-based Montgomery multiplication

In terms of word-based Montgomery multiplications, there exist different proposals in the literature: Koç et al.[72], Großschädl et al.[55] and Gura et al. [59]. Between those cited, I only focus on the one from Koç et al.[72]. In their paper, there are 5 algorithms for Montgomery multiplication called : Separated Operand Scanning (SOS), Coarsely Integrated Operand Scanning (CIOS), Finely Integrated Operand Scanning (FIOS), Finely Integrated Product Scanning (FIPS), Coarsely Integrated Hybrid Scanning (CIHS). The SOS algorithm is not exactly Montgomery multiplication as we defined before, since it performs the multiplication separate from the reduction, thus it will not be analyzed.

Algorithms 2, 3, 4 and 5 operate in a word size level, thus the value of $r$ is defined by the equation $r = 2^{re}$, where $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$. In this case, the number of bits of the exponent $r$ is 2 or more than the prime size and it is a multiple of the $ws$. If it is not a multiple of the word size, then in the last iteration of the algorithm, a different shift size than the previous iterations would be necessary. A shift itself is not expensive, but it is better to avoid operations with different shift sizes.

**Algorithm 2** CIOS Montgomery multiplication algorithm [72].

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod r$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$
**Ensure:** $o = a \cdot b/r \pmod n$

```
1:  o ← 0
2:  tc ← 0
3:  for i ← 0 to l − 1 by 1 do
4:      o_0||c_0 ← a_0 · b_i + o_0
5:      for j ← 1 to l − 1 by 1 do
6:          o_j||c_0 ← a_j · b_i + o_j + c_0
7:      end for
8:      tc_1||tc_0 ← tc_0 + c_0
9:      m_0 ← p'_0 · o_0 (mod w)
10:     null||c_0 ← (o_0 + m_0 · p_0)
11:     for j ← 1 to l − 1 by 1 do
12:         o_{j−1}||c_0 ← (o_j + m_0 · p_j + c_0)
13:     end for
14:     o_{l−1}||tc_0 ← tc_1 + tc_0 + c_0
15: end for
16: return o
```

Algorithm 2 shows how the CIOS algorithm works in pseudo code by scanning the inputs operands to perform the necessary operations (this is where the "IOS" comes from). The main loop in the algorithm starts by taking one word $b_i$ of the input $b$

and multiplying it by the entire operand $a$, giving the value $o$. Then, the value $o$ is reduced by one word by multiplying it with $p'$ modulus $w$. This process repeats for all $b$ elements and then $o$ should have the result of Montgomery multiplication. Because one multiplication iteration is done through the entire word $a$ before the reduction starts, then the integration of the procedures is coarse-grained.

**Algorithm 3** FIOS Montgomery multiplication algorithm [72].

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod r$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$
**Ensure:** $o = a \cdot b/r \pmod n$

```
1: o ← 0
2: tc ← 0
3: for i ← 0 to l − 1 by 1 do
4:     o_0||c_0 ← a_0 · b_i + o_0
5:     m_0 ← p'_0 · o_0 (mod w)
6:     null||c_1 ← o_0 + m_0 · p_0
7:     for j ← 1 to l − 1 by 1 do
8:         o_j||c_0 ← a_j · b_i + o_j + c_0
9:         o_{j−1}||c_1 ← o_j + m_0 · p_j + c_1
10:    end for
11:    o_{l−1}||tc_0 ← tc_0 + c_0 + c_1
12: end for
13: return o
```

FIOS, Algorithm 3, operates in a fine-grained manner by interleaving multiplication and reduction as close as possible. FIOS begins by multiplying $a_0$ by $b_i$, followed by the generation of the reduction value $m_0$. With $m_0$, both multiplication of $a$ and $b_i$ and the reduction of $o$ are performed together, one word at the time. This fine-grained integration can be seen as taking the CIOS algorithm and merging the two loops for product and reduction into a single one.

CIOS and FIOS algorithms scan through the operands $a$ and $b$ from least to most significant. On the other hand, FIPS Algorithm 4 scans the product from least to most significant. FIPS splits the process in two parts: in the first part the product and reduction is performed for all indexes of $a_i$ and $b_j$ if $i + j < l$. This condition is because only one half of the direct product of $a$ and $b$ needs reduction. Since $a$ and $b$ have $l$ words and $r = 2^l$, then the multiplication of $a \cdot b = t$ has $2l$ words, thus $t$ is $t = t_1 \cdot 2^l + t_0$. In this case, we can see that $t_1$ is already a multiple of $r$ and thus can be divided directly without having to add a multiple of the prime. In the first loop $i$ of the FIPS process, $t_0$ is computed and reduced, and in the second loop, $t_1$ is computed and added to $t_0$ and remaining parts of $t_0$ are reduced. Because of how the scanning of FIPS works, the index computations are done relatively to each other, which makes the control logic more complex than the one used in FIOS or CIOS. One way to avoid this more complex control mechanism is to unroll the loops for a given size.

CIHS, given in Algorithm 5, is a modification to the FIPS algorithm by shifting the reduction from the first loop into the second loop. In FIPS, the first loop computes $t_0$

**Algorithm 4** FIPS Montgomery multiplication algorithm [72].

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod r$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$
**Ensure:** $o = a \cdot b/r \pmod n$

$o \leftarrow 0$
$acc \leftarrow 0$
**for** $i \leftarrow 0$ **to** $l - 1$ **by** 1 **do**
  **for** $j \leftarrow 0$ **to** $i - 1$ **by** 1 **do**
    $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + a_j \cdot b_{i-j}$
    $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + o_j \cdot p_{i-j}$
  **end for**
  $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + a_i \cdot b_0$
  $o_i \leftarrow acc_0 \cdot {p'}_0 \pmod w$
  $acc_1||acc_2||null \leftarrow (acc_0||acc_1||acc_2) + o_i \cdot p_0$
**end for**
**for** $i \leftarrow l$ **to** $2l - 1$ **by** 1 **do**
  **for** $j \leftarrow i - l + 1$ **to** $l - 1$ **by** 1 **do**
    $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + a_j \cdot b_{i-j}$
    $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + o_j \cdot p_{i-j}$
  **end for**
  $o_{i-l} \leftarrow (acc_0||acc_1||acc_2)$
  $acc_1||acc_2||null \leftarrow (acc_0||acc_1||acc_2)$
**end for**
**return** $o$

and performs the relative reduction, while in CIHS, the first loop only computes the product of $a$ and $b$ up to position $l$. In the second loop, the reduction of the computed product in the previous loop is done by one word, followed by the computation of the next product of $a$ and $b$ by one word. When comparing FIPS and CIHS, FIPS is a simpler algorithm, since it has fewer loop steps and more concise behavior. CIHS has the advantage of not requiring an accumulator to operate, but directly operating over the array of words.

## 1.3 Montgomery-friendly primes

One common step in all presented algorithms is $m_0 \leftarrow o_0 \cdot {p'}_0 \pmod w$. This is a very crucial step, but it is also one of the most intensive steps in all algorithms, which might lead to special circuitry in hardware implementations. It is possible to avoid this step if ${p'}_0 = 1 \pmod w$, which makes $m_0 \leftarrow o_0$, which leads to other optimizations as adopting $\hat{p} = p+1$, since $\hat{p}_0 = 0 \pmod w$. Primes with the structure $k \cdot 2^x \pm 1$, where $k \in \mathbb{Z}^+$ and $x \geq ws$, are called Montgomery-friendly primes [37]. For those primes or composites for which ${p'}_0 = 1 \pmod w$, the previous 4 word-based algorithms can be rewritten as shown in Algorithms 6, 7, 8 and 9.

The difference between the regular Algorithms 2, 3, 4 and 5 and Algorithms 6, 7, 8 and 9 is very subtle. The difference on the optimized version is the removal of the

**Algorithm 5** CIHS Montgomery multiplication algorithm [72].

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod r$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$
**Ensure:** $o = a \cdot b/r \pmod n$

```
o ← 0
tc ← 0
for i ← 0 to l − 1 by 1 do
    o_i||c_0 ← o_i + a_0 · b_i
    for j ← 1 to l − 1 − i by 1 do
        o_{i+j}||c_0 ← o_{i+j} + a_j · b_i + c_0
    end for
    tc_0||c_0 ← tc_0 + c_0
    tc_1||null ← tc_1 + c_0
end for
for i ← 0 to l − 1 by 1 do
    m_0 ← o_0 · p'_0 (mod w)
    null||c_0 ← o_0 + m_0 · p_0
    for j ← 1 to l − 1 by 1 do
        o_{j−1}||c_0 ← o_j + m_0 · p_j + c_0
    end for
    o_{l−1}||c_0 ← t_0 + c_0
    tc_0||tc_1 ← t_1 + c_0
    for j ← i + 1 to l − 1 by 1 do
        o_{l−1}||c_0 ← o_{l−1} + b_j · a_{l−j+i} + c_0
        tc_0||c_0 ← tc_0 + c_0
        tc_1||null ← tc_1 + c_0
    end for
end for
return o
```

steps regarding computation of $m_0$. In some of them, this might not have big effects on the main loop, such as CIHS Algorithm 9, while others like FIOS, Algorithm 7 shows more regularity that can be exploited.

In algorithms with the $p'_0 = 1 \pmod w$, there is another optimization trick that exploits the fact that more than one position of $\hat{p}_i$ is 0. When $p'_0 = 1 \pmod w$, then $p_0 = -1 \pmod w$ and consequently, $\hat{p}_0 = 0 \pmod w$, thus it is possible to skip all multiplications by $\hat{p}_0 = 0 \pmod w$. The number of zeroes in the least significant part of $\hat{p}$ is $\lfloor x/ws \rfloor$, where $x$ is from the prime written as $p = k \cdot 2^x \pm 1$. This idea to use the extra zeroes in $\hat{p}$ has been seen before in the FIPS optimization by Costello et al. [37].

So far, Algorithms 6, 7, 8 and 9 only work for primes with the structure $p = k \cdot 2^x - 1$ and to make it work for primes $p = k \cdot 2^x + 1$, some changes are necessary. For primes $p = k \cdot 2^x + 1$, $p'$ has to be computed as $p' = p^{-1} \pmod r$, step 2 in Algorithm 1 should become a subtraction instead of addition, and $\hat{p}$ should be $\hat{p} = p - 1$. While these changes might seem costless as they require only a sign

change in the algorithm, a hardware unit designed to work with both cases might require an extra unit to invert sign and/or a special multiplier that works in signed and unsigned cases. A multiplier that can operate both in unsigned and signed cases is usually made from a signed multiplier that is 1 bit longer in both operands, which might have some significant cost in the final design. The change in the prime format also affects the output bounds as it will become $[-p, p]$ in the case of unsigned Montgomery multiplication, and $[-2p, p]$ in the case of the signed version.

**Algorithm 6** CIOS Montgomery multiplication algorithm for ${p'}_0 = 1 \pmod{w}$.

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod{r}$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$, $\hat{p} = p + 1$
**Ensure:** $o = a \cdot b/r \pmod{n}$

```
1:  o ← 0
2:  tc ← 0
3:  for i ← 0 to l − 1 by 1 do
4:      m_0||c_0 ← a_0 · b_i + o_0
5:      for j ← 1 to l − 1 by 1 do
6:          o_j||c_0 ← a_j · b_i + o_j + c_0
7:      end for
8:      tc_0||tc_1 ← tc_0 + c_0
9:      o_0||c_0 ← (o_1 + m_0 · p̂_1)
10:     for j ← 2 to l − 1 by 1 do
11:         o_{j−1}||c_0 ← (o_j + m_0 · p̂_j + c_0)
12:     end for
13:     o_{l−1}||c_0 ← tc_0 + c_0
14:     tc_0||null ← tc_1 + c_0
15: end for
16: return o
```

Orup [106] not only studied how to use this optimization in case of the FIOS variant and other Montgomery multiplication algorithms, but also how to use the optimized version for any prime. In case of primes where ${p'}_0 \neq 1 \pmod{w}$, then computations are done through a new composite $q = p \cdot {p'}_0$, thus $q = -1 \pmod{w}$ and $q' = 1 \pmod{w}$. Therefore, it is possible to use the optimized Algorithms 6, 7, 8 and 9 for all primes. However, all computations are done with the composite $q$ that is $ws$ bits longer than $p$. If removing from the Montgomery domain by performing a Montgomery multiplication by 1 is done through $q$ and the optimized algorithm, the output will be by $[0, 2q]$, which is not between $[0, 2p]$ as desired. To avoid an extra step to reduce the value in $[0, 2q]$ to $[0, p-1]$, one can apply a last Montgomery multiplication by 1 using the value $p$ with the non-optimized version of the algorithm. But to be able to use this trick, the value of $r$ has to be the same in the $q$ and $p$ variants, thus the value of $r$ has to be chosen as $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws + 1)\rceil \cdot ws)$.

**Algorithm 7** FIOS Montgomery multiplication algorithm for ${p'}_0 = 1 \pmod{w}$ seen in [106].

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod{r}$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$, $\hat{p} = p + 1$
**Ensure:** $o = a \cdot b/r \pmod{n}$

$o \leftarrow 0$
$tc \leftarrow 0$
**for** $i \leftarrow 0$ **to** $l - 1$ **by** 1 **do**
    $m_0||c_0 \leftarrow a_0 \cdot b_i$
    $c_1 \leftarrow 0$
    **for** $j \leftarrow 1$ **to** $l - 1$ **by** 1 **do**
        $o_j||c_0 \leftarrow a_j \cdot b_i + o_j + c_0$
        $o_{j-1}||c_1 \leftarrow m_0 \cdot \hat{p}_j + o_j + c_1$
    **end for**
    $o_{l-1}||tc_0 \leftarrow tc_0 + c_0 + c_1$
**end for**
**return** $o$

**Algorithm 8** FIPS Montgomery multiplication algorithm for ${p'}_0 = 1 \pmod{w}$ seen in [37].

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod{r}$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$, $\hat{p} = p + 1$
**Ensure:** $o = a \cdot b/r \pmod{n}$

$o \leftarrow 0$
$acc \leftarrow 0$
**for** $i \leftarrow 0$ **to** $l - 1$ **by** 1 **do**
    **for** $j \leftarrow 0$ **to** $i - 1$ **by** 1 **do**
        $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + a_j \cdot b_{i-j}$
        $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + o_j \cdot \hat{p}_{i-j}$
    **end for**
    $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + a_i \cdot b_0$
    $o_i \leftarrow acc_0$
    $acc_1||acc_2||null \leftarrow (acc_0||acc_1||acc_2) + o_i \cdot p_0$
**end for**
**for** $i \leftarrow l$ **to** $2l - 1$ **by** 1 **do**
    **for** $j \leftarrow i - l + 1$ **to** $l - 1$ **by** 1 **do**
        $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + a_j \cdot b_{i-j}$
        $acc_0||acc_1||acc_2 \leftarrow (acc_0||acc_1||acc_2) + o_j \cdot \hat{p}_{i-j}$
    **end for**
    $o_{i-l} \leftarrow acc_0$
    $acc_1||acc_2||null \leftarrow (acc_0||acc_1||acc_2)$
**end for**
**return** $o$

**Algorithm 9** CIHS Montgomery multiplication algorithm for ${p'}_0 = 1 \pmod{w}$.

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod{r}$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$, $\hat{p} = p + 1$
**Ensure:** $o = a \cdot b/r \pmod{n}$

```
1:  o ← 0
2:  tc ← 0
3:  for i ← 0 to l − 1 by 1 do
4:      o_i||c_0 ← o_i + a_0 · b_i
5:      for j ← 1 to l − 1 − i by 1 do
6:          o_{i+j}||c_0 ← o_{i+j} + a_j · b_i + c_0
7:      end for
8:      tc_0||c_0 ← tc_0 + c_0
9:      tc_1||null ← tc_1 + c_0
10: end for
11: for i ← 0 to l − 1 by 1 do
12:     m_0 ← o_0
13:     o_0||c_0 ← o_1 + m_0 · p̂_j
14:     for j ← 2 to l − 1 by 1 do
15:         o_{j−1}||c_0 ← o_j + m_0 · p̂_j + c_0
16:     end for
17:     o_{l−1}||c_0 ← tc_0 + c_0
18:     tc_0||tc_1 ← tc_1 + c_0
19:     for j ← i + 1 to l − 1 by 1 do
20:         o_{l−1}||c_0 ← o_{l−1} + b_j · a_{l−j+i}
21:         tc_0||c_0 ← tc_0 + c_0
22:         tc_1||null ← tc_1 + c_0
23:     end for
24: end for
25: return o
```

## 1.4 Final considerations of the chapter

Chapter 2 gives the first of my studies on Montgomery multiplication by implementing a Montgomery multiplier in hardware aimed at the IGLOO2 FPGA. The algorithms used for the implementation are both CIOS and FIOS, since they are more regular and can make an easier control unit. Additionally, Chapter 2 presents a more in-depth study on how to implement CIOS and FIOS in hardware.

Chapter 3 expands the design from Chapter 2 by implementing an elliptic curve co-processor that applies the idea of increasing the value of $r$ in order to not perform reductions during additions and subtractions, but only after multiplications. Later, a paper by Scott [124] showed the bounds of the intermediate operations and the expected size of $r$ in some elliptic curve formulas. Thanks to his paper, I discovered a problem in the original design of Chapter 3, because every subtraction was applying the same factor to make the value not negative. The correct way is to employ a different normalization factor on every step, according to the expected bound of the inputs, and thus the bounds have to be checked beforehand. In the case of the design

of Chapter 3, the original work is extended by showing the subtraction normalization factors that needed to be implemented. Also, it is shown that the complete formulas by Renes et al. [111] only requires 8 extra bits, which is less than the bounds provided by Scott's paper [124].

As discussed above, it is possible to avoid making every subtraction with a positive output by assuming the inputs in the modular reduction method can be signed integers. In the case of Chapter 4, the hardware unit employs a signed/unsigned representation, and thus the output of a subtraction does not need to be a positive number and the unit can handle both signed and unsigned values during the Montgomery reduction.

# Chapter 2

# Hardware modular multiplier

The first step to implement public-key primitives in hardware is to construct modules which can compute modular addition and multiplication. Between the addition and multiplication, addition is less expensive and it is often implemented along side multiplication, or even implemented directly with the schoolbook algorithm. Modular multiplication on the other hand, is often a main research topic and it can be implemented with different algorithms, one of them being Montgomery multiplication.

In the literature, there are a great number of co-processors for modular multiplication using Montgomery's algorithm, such as McIvor et al. [90], Vliegen et al. [135], Alrimeih and Rakhmatov [4], Ghosh et al. [54]. Some of these papers show results for modular multiplication, while others only for the complete implementation of the public-key cryptosystem. These related works can also be divided based on their strategy goals, where some aim for compact structures and others take full advantage of the existing resources for very high speed/throughput implementations. This chapter proposes two hardware co-processors performing Montgomery multiplication and integer addition/subtraction. Both structures strive for a low-area footprint, with the second one using more resources in order to provide better performance.

The presented implementations target the IGLOO 2 FPGA from Microsemi [93]. This FPGA is based on flash technology, as opposed to SRAM technology commonly seen in Xilinx and Altera FPGAs. Since it is a flash-based technology, it is labeled as more energy-efficient [93].

Note that Microsemi offers elliptic curve cryptography embedded cores on higher-end IGLOO 2 and SmartFusion 2 devices, starting from M2GL060 [93]. However, these are only available in a limited set of devices labeled as $TS$ and under restricted exportation rules.

**Chapter organization.**

This chapter is organized in three sections. Section 2.1 presents background information on the optimized Montgomery multiplication algorithms implemented, describes the FPGA main features, and details the proposed structures and respective implementations. Section 2.2 shows the obtained results, presents a comparison with the proposed solution and previous works. Section 2.3 concludes this chapter with some final considerations and future work directions.

## 2.1 CIOS and FIOS details and implementation

As explained in Chapter 1, Koç et al. [72] proposed Coarsely Integrated Operate Operand Scanning (CIOS) and Finely Integrated Operate Operand Scanning (FIOS) algorithms as an optimization for Montgomery multiplication targeting word size ($ws$) processors or multiplication units. The CIOS algorithm, as shown in Algorithm 10 computes one entire partial product (lines 4–8) before performing a reduction (lines 9–14). FIOS on the other hand, computes the multiplication of a word of each input (lines 4 and 8) and then proceeds with the reduction (lines 6, 9 and 11). While the CIOS algorithm operates directly on partial products, the iterative word-based structure of the FIOS algorithm allows for a design with two parallel units, one doing the multiplication (line 8) and the other one the reduction (line 9).

**Algorithm 10** CIOS Montgomery multiplication algorithm [72].

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/\text{ws})\rceil \cdot \text{ws})$, $p \cdot p' = -1 \pmod r$, $w = 2^{\text{ws}}$, $l = \lceil log_2(r/w)\rceil$
**Ensure:** $o = a \cdot b/r \pmod n$

1: $o \leftarrow 0$
2: $tc \leftarrow 0$
3: **for** $i \leftarrow 0$ **to** $l - 1$ **by** 1 **do**
4: $\quad o_0||c_0 \leftarrow a_0 \cdot b_i + o_0$
5: $\quad$ **for** $j \leftarrow 1$ **to** $l - 1$ **by** 1 **do**
6: $\quad\quad o_j||c_0 \leftarrow a_j \cdot b_i + o_j + c_0$
7: $\quad$ **end for**
8: $\quad tc_0||tc_1 \leftarrow tc_0 + c_0$
9: $\quad m_0 \leftarrow {p'}_0 \cdot o_0 \pmod w$
10: $\quad null||c_0 \leftarrow (o_0 + m_0 \cdot p_0)$
11: $\quad$ **for** $j \leftarrow 1$ **to** $l - 1$ **by** 1 **do**
12: $\quad\quad o_{j-1}||c_0 \leftarrow (o_j + m_0 \cdot p_j + c_0)$
13: $\quad$ **end for**
14: $\quad o_{l-1}||tc_0 \leftarrow tc_0 + tc_1 + c_0$
15: **end for**
16: **return** $o$

In the proposed implementations, the IGLOO 2 FPGA family from Microsemi is targeted. These FPGAs include dual-port memory with two inputs and two outputs and a three-port memory, which has two outputs and one input. The dual port memory can do 2 reads, 1 read and 1 write, or 2 writes at the same time. The three-port memory can do 2 reads and 1 write at the same time as long as the writing address is different from the reading addresses. Therefore, loads and stores on the three-port memory need to be carefully scheduled.

Another FPGA component is the Math Block, which is able to perform $z \leftarrow t \pm x \cdot y + (z >> 17)$, where $x$ and $y$ are unsigned 17-bits words and $t$ and $z$ are unsigned 43-bits words. This operation can be summed into a multiplication plus an addition and an optional carry of the previous computed value. By analyzing the main Math Block operation and the lines in CIOS Algorithm 10 it is very easy to map them. Lines 4 and 10 can be done by setting the previous $z$ to 0 and in lines 8 and 14,

**Algorithm 11** FIOS Montgomery multiplication algorithm [72]. Same inputs as Algorithm 10.

**Require:** $a, b \leq 2p$, $r = 2^{re}$, $re = (\lceil((\lceil log_2(p)\rceil + 2)/ws)\rceil \cdot ws)$, $p \cdot p' = -1 \pmod r$, $w = 2^{ws}$, $l = \lceil log_2(r/w)\rceil$
**Ensure:** $o = a \cdot b/r \pmod n$

```
1:  o ← 0
2:  tc ← 0
3:  for i ← 0 to l − 1 by 1 do
4:      o_0||c_0 ← a_0 · b_i + o_0
5:      m_0 ← p'_0 · o_0 (mod w)
6:      null||c_1 ← o_0 + m_0 · p_0
7:      for j ← 1 to l − 1 by 1 do
8:          o_j||c_0 ← a_j · b_i + o_j + c_0
9:          o_{j−1}||c_1 ← o_j + m_0 · p_j + c_1
10:     end for
11:     o_{l−1}||tc_0 ← tc_0 + c_0 + c_1
12: end for
13: return o
```

either $x$ or $y$ has to be set to 0. Lines 6 and 12 fit exactly into $z \leftarrow t+x\cdot y+(z >> 17)$ since in our case the words are 17 bits wide and the $c$ is the previously computed value shifted by the word size. Finally, line 9 can be done by setting $t$ to 0.

Just like CIOS, FIOS Algorithm 11 can also be done in one Math Block, however it is more interesting to pipeline it in two stages since the result of line 8 does not depend on line 9 computation. Therefore, the first pipeline stage computes line 8 and outputs to the second stage that computes line 9. Line 11 can be split into $t_0 \leftarrow o_l+c_0$ and $tc_0, o_{l-1} \leftarrow t_0 + c_1$, then each part happens on a different stage. Line 4 is the same as line 8, but lines 5 and 6 happen in the second stage. Therefore, the pipeline cannot operate non-stop, because there is a bubble to wait for line 5 to be resolved.

The previous mapping is depicted in Figures 2.1 and 2.2: structure for CIOS and for FIOS, respectively. Because CIOS does only one operation per cycle one memory of three ports is enough for all operations. However, FIOS requires at least two memory of three ports, since up to three values are read at once and one value is written. Because FIOS has two memories, it supports primes up to 539 bits, on the other hand the CIOS structure supports only up to 267 bit primes because of the memory size. The memories itself are not fully registered, so values are directly fed into the circuit.

For the addition and subtraction operations the applied algorithm is the same as described by Batina et al. [13]. In this method additions are done without applying reduction, because the reduction can be postponed for the next multiplication. In the Math Block with $z \leftarrow t + x \cdot y + (z >> 17)$ this is done by setting $x$ as 1 while the addition operands are in $t$ and $y$. Subtractions also do not require reduction, but the output needs to be added with a multiple of the prime to guarantee the output is non-negative. The subtraction is then $o \leftarrow c - b + 2n$, which in the CIOS circuit is done by computing $t \leftarrow c - 1b$, and then $o \leftarrow t + 2n$. The FIOS circuit can be used in

Figure 2.1: Montgomery multiplier structure for CIOS with 1 Math Block and 1 memory.

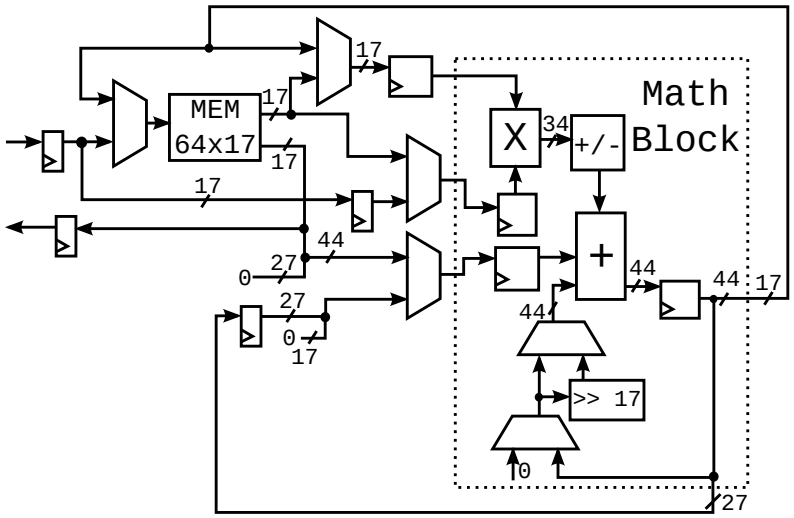
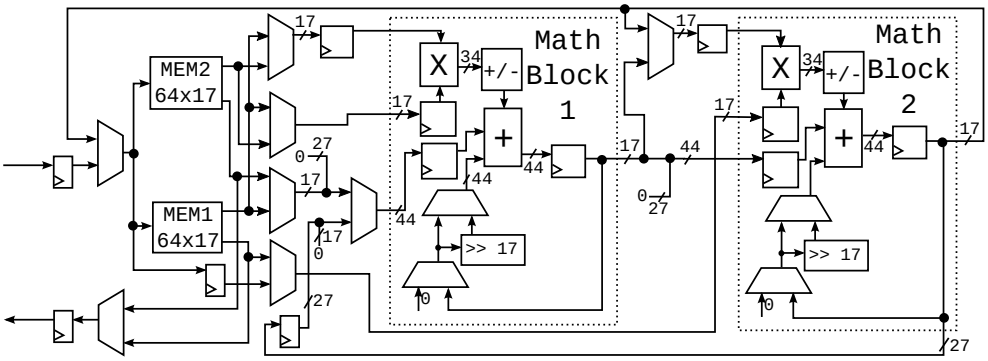



Figure 2.2: Montgomery multiplier structure for FIOS with 2 Math Blocks and 2 memories.




pipeline mode where the first stage computes $t \leftarrow c - 1b$, and the second $o \leftarrow t + 2n$. This delayed reduction has a small price to pay in the Montgomery multiplication where the number of bits that are added in $r$ is not only 2, but 4 instead [13]. To support even more additions and subtractions, we implement our solution with a minimum of 5 extra bits instead of only 4. Because one of the multiplications inputs has to be 1 in additions and subtractions it was opted to have a register outside of the Math Block that can be directly set to 1.

## 2.2 Hardware results and comparison

In order to properly evaluate the proposed structures, experimental results were compared with the state-of-the-art in Tables 2.1 and 2.2. However, only previous work

Table 2.1: Comparison of our results with the relevant literature on hardware implementations for ECC in terms of resources.

| Work | Field | FPGA | LUT4 | FF | Emb. Mult. | BRAM 64x18 | BRAM 1kx18 |
|---|---|---|---|---|---|---|---|
| Only Finite Field Multiplication | | | | | | | |
| [90] | 256 | Virtex II Pro | 2866$^\star$ | 2866$^\star$ | 4 | 0 | 0 |
| [105] | 256 | Virtex E | 3096$^\star$ | 3096$^\star$ | 0 | 0 | 0 |
| [119] | 512 | Virtex 5 | 40$^\star$ | 40$^\star$ | 4 | 0 | 4 |
| [99] | 512 | Artix 7 | 1789 | 2164 | 57 | 0 | 0 |
| [5] | 256 | Virtex 7 | 1917 | – | 9 | 0 | 0 |
| Finite Field Addition, Subtraction and Multiplication | | | | | | | |
| This 1 | 251-267 | IGLOO 2 | 505 | 257 | 1 | 1 | 0 |
| This 2 | 251-267 | IGLOO 2 | 680 | 341 | 2 | 2 | 0 |
| This 2 | 506-522 | IGLOO 2 | 680 | 341 | 2 | 2 | 0 |
| Finite Field with Inversion | | | | | | | |
| [43] | 256 | Virtex II | 6218$^\star$ | 6218$^\star$ | 0 | 0 | 0 |
| Full ECC Processors | | | | | | | |
| [134] | 256 | SmartFusion | 3690 | 3690 | 0 | 0 | 12 |
| [134] | 256 | Virtex II Pro | 1546$^\star$ | 1546$^\star$ | 1 | 0 | 3 |
| [134] | 256 | Virtex II Pro | 2316$^\star$ | 2316$^\star$ | 4 | 0 | 3 |
| [135] | 256 | Virtex II Pro | 3664$^\star$ | 3664$^\star$ | 2 | 0 | 9 |

$^\star$ Maximum possible value assumed from the number of slices.
$^\dagger$ Values estimated by multiplying time with frequency.

with multiplication timing results are depicted. Other work considering low latency and/or high throughput are not discussed, such as [104, 65], since they require significantly more FPGA resources, thus not resulting in a meaningful comparison.

Given the amount of resources of the two proposed structures, it is possible to compute the percentage of the total resources taken by each one. In the IGLOO2 M2GL005 with a total of 6060 LUT4, 11 Math Blocks, 11 BRAMs 64x18 and 10 BRAMs 1kx18 [93], the first structure occupies 7.7% and the second 13.3% of the FPGA total. In terms of efficiency ($latency \cdot area$) the first has 17.94 and the second 16.63, thus the second is 8% more efficient and supports fields up to 522 bits in comparison with the first structure. It should be noted that both structures are able to operate at a frequency of 250MHz, the maximum frequency imposed by the three port Memory Block [93].

In the literature, McIvor et al. [90] considered the SOS, CIOS and FIOS algorithms proposed in [72] targeting a Virtex II Pro® FPGA. The proposed design is identical for the 3 algorithms using an ALU to perform the multiplication or addition. This approach is similar to our more compact one, implementing CIOS, in the aspect that it only uses one ALU iterating over the intermediate values. However, this structure requires multiple cycles to perform the operation $t+x\cdot y+carry$, while ours does it in a single cycle. Also, the implementation of McIvor et al. requires significantly more area, 4 DSPs to implement the ALU, and only operates at 100MHz. The multiple cycle and lower frequency are related to an older technology, and not a design choice.

Table 2.2: Comparison of our results with the relevant literature on hardware implementations for ECC in terms of cycles and latency. The speed results are for one modular multiplication.

| Work | Field | FPGA | Frequency (MHz) | Add. Cycles | Sub. Cycles | Mult. Cycles | Time (µs) |
|---|---|---|---|---|---|---|---|
| Only Finite Field Multiplication | | | | | | | |
| [90] | 256 | Virtex II Pro | 101.87 | – | – | 582 | 5.7 |
| [105] | 256 | Virtex E | 100.44 | – | – | 772† | 7.69 |
| [119] | 512 | Virtex 5 | – | – | – | – | 8.67 |
| [99] | 512 | Artix 7 | 106 | – | – | 66 | 0.62 |
| [5] | 256 | Virtex 7 | 225 | – | – | 114 | 0.51 |
| Finite Field Addition, Subtraction and Multiplication | | | | | | | |
| This 1 | 251-267 | IGLOO 2 | 250 | 21 | 37 | 583 | 2.33 |
| This 2 | 251-267 | IGLOO 2 | 250 | 23 | 23 | 312 | 1.25 |
| This 2 | 506-522 | IGLOO 2 | 250 | 38 | 38 | 1062 | 4.25 |
| Finite Field with Inversion | | | | | | | |
| [43] | 256 | Virtex II | 31.92 | 1 | 1 | 257 | 8.05 |
| Full ECC Processors | | | | | | | |
| [134] | 256 | SmartFusion | 109 | 46 | 46 | 401 | 3.7 |
| [134] | 256 | Virtex II Pro | 210 | 46 | 46 | 401 | 1.9 |
| [134] | 256 | Virtex II Pro | 210 | 28 | 28 | 157 | 0.75 |
| [135] | 256 | Virtex II Pro | 108.2 | 44 | 44 | 637 | 5.89 |

⋆ Maximum possible value assumed from the number of slices.
† Values estimated by multiplying time with frequency.

Örs et al. [105] propose a systolic array architecture for Montgomery modular multiplication. The systolic array approach has an array of cells where each cell computes one bit of the modular multiplication. This strategy is more suitable when embedded multipliers are not available, such as in ASICs. Nevertheless, it still requires more cycles to perform the computation.

Also not using the embedded primitives of the FPGA to store or to compute the data is the work of Daly et al. [43]. They propose a structure with an adder of the size of the input and perform the multiplication iteratively, requiring as many cycles as the prime length. This way, a multiplication can be computed in each iteration by multiplying one bit of one of the inputs by the other entire input. This approach allows to reduce the number of computational cycles, but results in a significantly larger data path and consequently lower frequencies. Moreover, the amount of occupied LUTs is also very high, imposing a very high area cost.

Varchola et al. [134] proposed a small ECC co-processor only supporting NIST primes curves. This work also has the main goal of achieving an ECC design as small as possible, therefore using only a small number of DSPs. Their structure follows a similar approach as ours, using a single ALU that keeps receiving values to be processed. However, their ALU is composed of one multiplier, using the built-in DSP in the FPGA, followed by a final adder implemented using FPGA LUTs. The separation between multiplier and adder allows for simplification of the operation

scheduling, and thus to use fewer cycles to compute the result at the cost of more LUTs and routing. The authors present three implementations, one supported by SmartFusion® FPGA which does not include embedded multipliers, and two others supported by a Xilinx® Virtex II Pro®. In Virtex II Pro® based structures, one employs a single embedded multiplier, achieving a computation delay of 1.9 µs, and the other employs 4 multipliers, achieving a computation delay of 0.75 µs. Since the presented area is for the entire ECC co-processor with scalar point multiplication, a fair analysis cannot be done. However, the less compact structure suggests a higher area cost potentially allowing a faster computation.

Another approach with a minimization strategy was proposed by Vliegen et al. [135] which also targeted a Virtex II Pro® FPGA. The authors mapped the CIOS algorithm in a more straightforward manner not worrying so much about the resource mapping and operation scheduling. As a consequence, the resulting structure requires 2 embedded multipliers and significantly more LUTs and memory blocks. Despite this, the resulted computational delay is twice as long as ours, mostly due to the much lower attainable frequency, resulting in a less efficient design.

From this, it can be concluded that with a careful mapping of the resources available in the FPGA devices, along with a careful scheduling and reuse of the needed operations, significant improvements and area savings can be achieved.

## 2.3 Final considerations of the chapter

The work presented in this chapter proposes two area-optimized FPGA structures for the computation of the Montgomery modular multiplication algorithm for generic primes targeting low–power IGLOO 2 FPGAs from Microsemi. The proposed structures impose a very low usage of the available FPGA resources while still achieving good performance. To achieve this performance with a low area the Math Blocks and embedded memories were used together with a careful scheduling to assure a full pipeline usage and a low number of computation cycles. While the first structure achieves the lowest area, the second one allows to approximately half the computation time at the cost of twice the amount of embedded memories and Math Blocks and only 35% more LUTs and registers.

# Chapter 3

# Elliptic curve co-processor

Hardware implementations of elliptic curve cryptography are deployed and used in various protocols and devices. Even though it can be considered a solved problem to deploy in hardware, there is still lots of new research into finding faster, secure, cheaper and energy efficient solutions. Therefore it is essential to explore different paths, options and architectures that later can become better solutions for tomorrow.

There are numerous hardware implementations of elliptic curve-based cryptography (ECC) in various FPGA platforms and different goals, a few selected works aiming at high speed or low resources are shown. In terms of high speed design, there are some works from Güneysu and Paar [58], Guillermin [57], Yao et al. [138] and Sakiyama et al. [118]. Güneysu and Paar [58] proposed a new speed-optimized architecture that makes intensive use of the DSP blocks in an FPGA platform. Guillermin [57] introduced a prime field ECC hardware architecture and implemented it on several Altera FPGA boards. The design is based on Residue Number System (RNS), facilitating carry-free arithmetic and parallelism. Yao et al. [138] followed the idea of using RNS to design a high-speed ECC co-processor for pairings. Sakiyama et al. [118] proposed a superscalar coprocessor that could deal with three different curve-based cryptosystems, all in characteristic 2 fields.

In terms of less resources some other works that are interesting are from Varchola et al. [134], Pöpper et al. [108], Roy et al. [114], Fan et al. [48] and Vliegen et al. [135]. Varchola et al. [134] designed a processor-like architecture, with instruction set and decoder, on top of which they implemented ECC. This approach has the benefit of having a portion written in software, which can be easily maintained and updated, while having special optimized instructions for the elliptic curve operations. The downside of this approach is that the resource costs are higher than a fully optimized processor. As was the case for Güneysu and Paar [58], their targets were standardized NIST prime curves P–224 and P–256. Consequently, each of their synthesized circuit would only work for one of the two primes. Pöpper et al. [108] follow the same approach as Varchola et al. [134], with some side-channel related improvements. The paper focuses on an analysis of each countermeasure and its effective cost. Roy et al. [114] followed the same path, but with more optimizations with respect to resources and only for curve NIST P–256. However, the number of Block RAMs necessary for the architecture is much larger than of Pöpper et al. [108] or Varchola

et al. [134]. Fan et al. [48] created an architecture for special primes and curves, namely the standardized NIST P–192. The approach was to parallelize Montgomery multiplication and formulas for point addition and doubling on the curve. Vliegen et al. [135] attempted to reduce the resources with a small core aimed at 256-bit primes.

While the proposed architecture in this chapter is aiming for a small elliptic curve cryptography architecture, it has some novel parts. The architecture here exploits the parallelism in Renes et al.[111] elliptic curve point addition formulas, which has not been done before. The architecture also builds on Chapter 2, which also helps to benchmark and understand if the modular multiplier used is a good solution.

**Chapter organization.**

This chapter starts with preliminaries in Section 3.1, and briefly discusses parallelism for the complete formulas in Section 3.2. Later, Section 3.3 shows the proposed elliptic curve co-processor using three Montgomery multipliers. Later, Section 3.4 shows some mistakes assumed in the number of bits in the previous implementation and tells how to compute them properly. Finally, Section 3.5 has the final considerations for this chapter.

## 3.1 Preliminaries for elliptic curve cryptography

Let us define $\mathbb{F}_q$ as the finite field of order $q$, given by $q = p^n$, where $n \in \mathbb{Z}^+$ and $p$ is a prime not equal to 2 or 3. An elliptic curve in the short Weierstrass form is given with the following equation:

$$E : Y^2Z = X^3 + aXZ^2 + bZ^3 \subset \mathbb{P}^2$$

In this equation, the points are defined by $(X : Y : Z) \in \mathbb{P}^2$, where $\mathbb{P}^2$ is the projective space defined over the field $\mathbb{F}_q$, and $a, b \in \mathbb{F}_q$ are curve constants. The *infinity point* is defined as $\mathcal{O} = (0 : 1 : 0)$ and all the other points are defined as *affine points*. Because the affine points have $Z \neq 0$, then an affine point $(X : Y : Z)$ can be rewritten as $(x : y)$, where $x = X/Z$ and $y = Y/Z$. For the case of $(x : y)$ notation, the equation can be rewritten as:

$$y^2 = x^3 + ax + b$$

The points from curve $E$ form a group where $\mathcal{O}$ is the identity point. The subgroup of $\mathbb{F}_q$-rational points is denoted by $E(\mathbb{F}_q)$, and the amount of points in the elliptic curve $E$ together with $\mathcal{O}$ is called *order* and is given by $\#E(\mathbb{F}_q)$. In the remainder of this chapter it will be assumed $n = 1$, thus $q = p$, and the points from curve $E$ form a group whose order is prime.

In this work it was chosen to use the group law presented in Renes et al. [111] with homogenous coordinates. In Renes et al. [111] work, the sum of two points $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$ is computed as $P + Q = (X_3 : Y_3 : Z_3)$, where

$$
\begin{aligned}
X_3 &= (X_1Y_2 + X_2Y_1)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2) - \\
&\quad (Y_1Z_2 + Y_2Z_1)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2), \\
Y_3 &= (3X_1X_2 + aZ_1Z_2)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2) + \\
&(Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2), \\
Z_3 &= (Y_1Z_2 + Y_2Z_1)(Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2) + \\
&\quad (X_1Y_2 + X_2Y_1)(3X_1X_2 + aZ_1Z_2). \qquad (3.1)
\end{aligned}
$$

Elliptic curve cryptography [71, 94] commonly relies on the hard problem called the "Elliptic Curve Discrete Logarithm Problem (ECDLP)". This means that given two points $P, Q$ on an elliptic curve, it is hard to find a scalar $k \in \mathbb{Z}$ such that $Q = kP$, if it exists. Therefore the main component of curve-based cryptosystems is the scalar multiplication operation $(k, P) \mapsto kP$. Since in many cases $k$ is a secret, this operation is very sensitive to attacks. In particular many side-channel attacks [73, 14] and countermeasures [33] have been proposed. To ensure protection against simple power analysis (SPA) attacks, it is important to use regular scalar multiplication algorithms, e. g. Montgomery ladder [66] or Double-And-Add-Always [33], executing both an addition and a doubling operation per scalar bit.

Examples of cryptosystems based on elliptic curve cryptography with the security related to the ECDLP are Elliptic Curve Diffie–Hellman (ECDH), Elliptic Curve Integrated Encryption Scheme (ECIES) and Schnorr signature scheme. The ECDH is the original Diffie-Hellman [45] key exchange protocol, but instead of relying on modular integer exponentiation it is based on scalar multiplication of elliptic curves. The ECDH is only used to exchange random key material between parties and in the case of sending messages with semantics it is required to employ ECIES [126] or similar schemes. ECIES adds an extra layer to the ECDH protocol in order to encrypt messages and exchange them with another party. In terms of signatures another example is the original Schnorr protocol [122] which also can be used with modular integer exponentiation and elliptic curves. Elliptic curves then offer the same security guarantees as modular integer exponentiation, with the added value of being an order of magnitude smaller, roughly 10 times, for key material and shared secrets and requiring less computing power. When compared to other cryptosystems, like the ones based on RSA [112], elliptic curves are still smaller and faster for key exchange, encryption and signature generation, except for signature verification where RSA is faster than elliptic curves.

## 3.2 Parallelism in Elliptic Curve Formulas

An important way to increase the efficiency of the implementation is to use multiple Montgomery multipliers in parallel. In this section, I give a brief explanation of our choice of three multipliers.

The addition formulas on which our scalar multiplication is built are shown in Algorithm 12, that was obtained from Renes et al. [111]. We choose to ignore additions and subtractions since we assume to be relying on a Montgomery multiplier for which the cost of field multiplications is far higher than that of field additions. The total

**Algorithm 12** Complete addition formulas for a prime order elliptic curve in Weierstrass form from Renes et al. [111]

| | | |
|---|---|---|
| 1. $t_0 \leftarrow X_1 \cdot X_2$; | 2. $t_1 \leftarrow Y_1 \cdot Y_2$; | 3. $t_2 \leftarrow Z_1 \cdot Z_2$; |
| 4. $t_3 \leftarrow X_1 + Y_1$; | 5. $t_4 \leftarrow X_2 + Y_2$; | 6. $t_3 \leftarrow t_3 \cdot t_4$; |
| 7. $t_4 \leftarrow t_0 + t_1$; | 8. $t_3 \leftarrow t_3 - t_4$; | 9. $t_4 \leftarrow X_1 + Z_1$; |
| 10. $t_5 \leftarrow X_2 + Z_2$; | 11. $t_4 \leftarrow t_4 \cdot t_5$; | 12. $t_5 \leftarrow t_0 + t_2$; |
| 13. $t_4 \leftarrow t_4 - t_5$; | 14. $t_5 \leftarrow Y_1 + Z_1$; | 15. $X_3 \leftarrow Y_2 + Z_2$; |
| 16. $t_5 \leftarrow t_5 \cdot X_3$; | 17. $X_3 \leftarrow t_1 + t_2$; | 18. $t_5 \leftarrow t_5 - X_3$; |
| 19. $Z_3 \leftarrow a \cdot t_4$; | 20. $X_3 \leftarrow b_3 \cdot t_2$; | 21. $Z_3 \leftarrow X_3 + Z_3$; |
| 22. $X_3 \leftarrow t_1 - Z_3$; | 23. $Z_3 \leftarrow t_1 + Z_3$; | 24. $Y_3 \leftarrow X_3 \cdot Z_3$; |
| 25. $t_1 \leftarrow t_0 + t_0$; | 26. $t_1 \leftarrow t_1 + t_0$; | 27. $t_2 \leftarrow a \cdot t_2$; |
| 28. $t_4 \leftarrow b_3 \cdot t_4$; | 29. $t_1 \leftarrow t_1 + t_2$; | 30. $t_2 \leftarrow t_0 - t_2$; |
| 31. $t_2 \leftarrow a \cdot t_2$; | 32. $t_4 \leftarrow t_4 + t_2$; | 33. $t_0 \leftarrow t_1 \cdot t_4$; |
| 34. $Y_3 \leftarrow Y_3 + t_0$; | 35. $t_0 \leftarrow t_5 \cdot t_4$; | 36. $X_3 \leftarrow t_3 \cdot X_3$; |
| 37. $X_3 \leftarrow X_3 - t_0$; | 38. $t_0 \leftarrow t_3 \cdot t_1$; | 39. $Z_3 \leftarrow t_5 \cdot Z_3$; |
| 40. $Z_3 \leftarrow Z_3 - t_0$; | | |

cost of multiplications in the most general case is $12\mathbf{M} + 2\mathbf{m}_a + 3\mathbf{m}_{3b}$[1]. Because our processors do not distinguish between full multiplications and multiplications by constants, we consider this cost simply as $17\mathbf{M}$. The authors of [111] introduce optimizations for mixed addition and doubling, but in our case this only saves a single multiplication (and some additions). Since this does not make up for the price we would have to pay for the implementation of a second algorithm, we only examine the most general case. In Table 3.1, the interdependencies of the multiplications are given.

This allows us to write down algorithms for implementations running $n$ processors in parallel. Denote by $\mathbf{M_n}$ and $\mathbf{a_n}$ the cost of doing $n$ multiplications and additions (or subtractions) in parallel, respectively. In Table 3.2, there are the costs for $1 \leq n \leq 6$. When computing the costs, it is assumed that $\mathbf{M_n} = \mathbf{M}$ and $\mathbf{a_n} = \mathbf{a}$. This assumption ignores some practical aspects, for example a larger number of Montgomery multipliers can result in a scheduling overhead. All algorithms and their respective Magma [26] verification code can be found in the original paper [88] or Rene's thesis [110]. For our implementation $n = 3$ was chosen, i. e. three Montgomery multipliers, which are described in Algorithm 13. This number of multipliers achieves a great area-time trade-off, while obtaining a good speed-up compared to $n = 1$. Moreover, the aforementioned practical issues (e. g. scheduling) are not as complicated to deal with for larger $n$.

[1] In this chapter, $\mathbf{M}$, $\mathbf{m}_a$, $\mathbf{m}_{3b}$, $\mathbf{a}$ are the cost: general multiplication, a multiplication by curve constant $a$, a multiplication by curve constant $3b$, and an addition respectively.

Table 3.1: Dependencies of multiplications inside the complete addition formulas.

| Stage | Result | Multiplication | Dependent on |
|---|---|---|---|
| 0 | $\ell_0$ | $X_1 \cdot X_2$ | - |
| 0 | $\ell_1$ | $Y_1 \cdot Y_2$ | - |
| 0 | $\ell_2$ | $Z_1 \cdot Z_2$ | - |
| 0 | $\ell_3$ | $(X_1 + Y_1) \cdot (X_2 + Y_2)$ | - |
| 0 | $\ell_4$ | $(X_1 + Z_1) \cdot (X_2 + Z_2)$ | - |
| 0 | $\ell_5$ | $(Y_1 + Z_1) \cdot (Y_2 + Z_2)$ | - |
| 1 | $\ell_6$ | $b_3 \cdot \ell_2$ | $\ell_2$ |
| 1 | $\ell_7$ | $a \cdot \ell_2$ | $\ell_2$ |
| 1 | $\ell_8$ | $a \cdot (\ell_4 - \ell_0 - \ell_2)$ | $\ell_0, \ell_2, \ell_4$ |
| 1 | $\ell_9$ | $b_3 \cdot (\ell_4 - \ell_0 - \ell_2)$ | $\ell_0, \ell_2, \ell_4$ |
| 2 | $\ell_{10}$ | $a \cdot (\ell_0 - \ell_7)$ | $\ell_0, \ell_7$ |
| 2 | $\ell_{11}$ | $(\ell_3 - \ell_0 - \ell_1) \cdot (\ell_1 - \ell_8 - \ell_6)$ | $\ell_0, \ell_1, \ell_3, \ell_6, \ell_8$ |
| 2 | $\ell_{13}$ | $(\ell_1 + \ell_8 + \ell_6) \cdot (\ell_1 - \ell_8 - \ell_6)$ | $\ell_1, \ell_6, \ell_8$ |
| 2 | $\ell_{15}$ | $(\ell_5 - \ell_1 - \ell_2) \cdot (\ell_1 + \ell_8 + \ell_6)$ | $\ell_1, \ell_2, \ell_5, \ell_6, \ell_8$ |
| 2 | $\ell_{16}$ | $(\ell_3 - \ell_0 - \ell_1) \cdot (3\ell_0 + \ell_7)$ | $\ell_0, \ell_1, \ell_3, \ell_7$ |
| 3 | $\ell_{12}$ | $(\ell_5 - \ell_1 - \ell_2) \cdot (\ell_{10} + \ell_9)$ | $\ell_1, \ell_2, \ell_5, \ell_9, \ell_{10}$ |
| 3 | $\ell_{14}$ | $(3\ell_0 + \ell_7) \cdot (\ell_{10} + \ell_9)$ | $\ell_0, \ell_7, \ell_9, \ell_{10}$ |

Table 3.2: Efficiency approximation of the number of Montgomery multipliers against the area used.

| $n$ | *Cost* | *Area* $\times$ *Time* | *Algorithm* |
|---|---|---|---|
| 1 | $17\mathbf{M} + 23\mathbf{a}$ | $17\mathbf{M} + 23\mathbf{a}$ | Algorithm 12 |
| 2 | $9\mathbf{M}_2 + 12\mathbf{a}_2$ | $18\mathbf{M} + 24\mathbf{a}$ | [88] |
| 3 | $6\mathbf{M}_3 + 8\mathbf{a}_3$ | $18\mathbf{M} + 24\mathbf{a}$ | Algorithm 13 |
| 4 | $5\mathbf{M}_4 + 7\mathbf{a}_4$ | $20\mathbf{M} + 28\mathbf{a}$ | [88] |
| 5 | $4\mathbf{M}_5 + 6\mathbf{a}_5$ | $20\mathbf{M} + 30\mathbf{a}$ | [88] |
| 6 | $3\mathbf{M}_6 + 6\mathbf{a}_6$ | $18\mathbf{M} + 36\mathbf{a}$ | [88] |

## 3.3 Implementation of the formulas with 3 processors

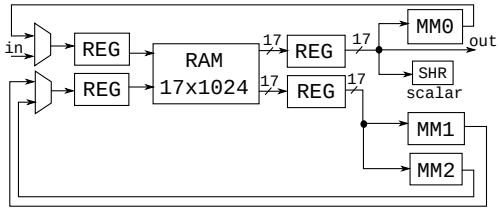
In this section, we introduce a novel hardware implementation, parallelizing the new formulas using three Montgomery processors. I make use of the Montgomery processor based on FIOS, which has been proposed in Chapter 2 for Microsemi IGLOO2 FPGAs. As a consequence of building on top of this processor, we target the same FPGA. However, it is straightforward to port to other FPGAs or even ASICs which have a Montgomery multiplier with the same interface.

The elliptic curve scalar multiplication routine is constructed on top of the Montgomery processors. As mentioned before, to protect against SPA attacks, a regular scalar multiplication algorithm (i. e. Double-And-Add-Always [33]) was implemented. The algorithm relies on three registers $\mathtt{R}_0$, $\mathtt{R}_1$ and $\mathtt{R}_2$, which are implemented as mem-

**Algorithm 13** Parallelized complete addition formulas for a prime order elliptic curve in Weierstrass form, using *three* processors

| | | | | | |
|---|---|---|---|---|---|
| | $t_0 \leftarrow X_1 \cdot X_2$; | | $t_3 \leftarrow X_1 + Y_1$; | | $t_6 \leftarrow Y_2 + Z_2$; |
| 1. | $t_1 \leftarrow Y_1 \cdot Y_2$; | 2. | $t_4 \leftarrow X_2 + Y_2$; | 3. | $t_7 \leftarrow X_1 + Z_1$; |
| | $t_2 \leftarrow Z_1 \cdot Z_2$; | | $t_5 \leftarrow Y_1 + Z_1$; | | $t_8 \leftarrow X_2 + Z_2$; |
| | $t_9 \leftarrow t_3 \cdot t_4$; | | $t_3 \leftarrow t_0 + t_1$; | | |
| 4. | $t_{10} \leftarrow t_5 \cdot t_6$; | 5. | $t_4 \leftarrow t_1 + t_2$; | 6. | $t_6 \leftarrow b_3 \cdot t_2$; |
| | $t_{11} \leftarrow t_7 \cdot t_8$; | | $t_5 \leftarrow t_0 + t_2$; | | $t_8 \leftarrow a \cdot t_2$; |
| | $t_2 \leftarrow t_9 - t_3 + 4p$; | | $t_{10} \leftarrow t_9 + t_0$; | | $t_0 \leftarrow a \cdot t_4$; |
| 7. | $t_9 \leftarrow t_0 + t_0$; | 8. | $t_4 \leftarrow t_{11} - t_5 + 4p$; | 9. | $t_5 \leftarrow b_3 \cdot t_4$; |
| | $t_3 \leftarrow t_{10} - t_4 + 4p$; | | $t_7 \leftarrow t_0 - t_8 + 4p$; | | $t_9 \leftarrow a \cdot t_7$; |
| | $t_4 \leftarrow t_0 + t_6$; | | | | $t_1 \leftarrow t_5 \cdot t_6$; |
| 10. | $t_7 \leftarrow t_5 + t_9$; | 11. | $t_5 \leftarrow t_1 - t_4 + 4p$; | 12. | $t_4 \leftarrow t_0 \cdot t_7$; |
| | $t_0 \leftarrow t_8 + t_{10}$; | | $t_6 \leftarrow t_1 + t_4$; | | $t_8 \leftarrow t_3 \cdot t_7$; |
| | $t_9 \leftarrow t_2 \cdot t_5$; | | $X_3 \leftarrow t_9 - t_8 + 4p$; | | |
| 13. | $t_{10} \leftarrow t_3 \cdot t_6$; | 14. | $Y_3 \leftarrow t_1 + t_4$; | | |
| | $t_{11} \leftarrow t_0 \cdot t_2$; | | $Z_3 \leftarrow t_{10} + t_{11}$; | | |

Figure 3.1: Entire architecture with three Montgomery processors as described in Chapter 2, where MM = Montgomery processor, SHR = Shift register, REG = Register.




ory position in the Figure 3.1 RAM. The register $R_0$ contains the operand, which is always doubled. The registers $R_1$ and $R_2$ contain the result of the addition when the exponent bit is zero and one, respectively. This algorithm should be applied carefully since it is prone to fault attacks [25]. From a very high level point of view the architecture consists of the three Montgomery multipliers and a single BRAM block, as shown in Figure 3.1. This BRAM block is more than large enough to store the necessary temporary variables. Although Renes et al. [111] Algorithm 12 tries to minimize the number of memory locations and as consequence Algorithm 13 also minimizes, this is not necessary for our implementation. In the remainder of this section there are more details of the implementation.

### Memory

The main RAM memory in Figure 3.1 operates as a true dual port memory of 1024 words of 17 bits, which is subdivided into a *big word* of 32 words (i. e. 544 bits). Therefore, the main memory is $32 \times 32$ big words. A big word can accommodate any temporary variable, input or output of our architecture. A possible exception could be the scalar of the point scalar multiplication. Although a single word would be large enough to contain 523-bit scalars (in the largest case of a 523-bit field), the scalar blinding technique can double the size of the scalar. Therefore, we use two words to store the scalar. By doing this, it will be possible to execute scalar multiplication with a blinded scalar [31]. Lastly, there is a 17-bit shift register into which the scalar is loaded word by word.

### Control logic

The formulas and control system are done through two state machines: a main one which controls everything, and one related to memory transfer.

The memory-transfer state machine was created with the purpose to reduce the number of states in the main machine. This was done by providing the operation of transfer between the main memory and the Montgomery processors memory. Therefore, the main machine can transfer values with just one state, and can reuse most of the transfer logic. This memory-transfer machine becomes responsible for various parts of the bus between main memories, processors and other counters. However, the main state machine still has to be able to control everything. Hence, the main state machine shares some components with the memory transfer machine, increasing control circuit costs.

The main state machine controls all the circuits that compose the entire cryptographic core. Given it controls the entire circuit, the machine also has the entire Table 3.2 scheduling implemented as states. The advantage of doing this through states is the possible optimization of the design and the entire control. However, the cost of maintenance is a lot higher than a small instruction set or microcode that can also implement the addition formulas or scalar multiplication. Because the addition formulas are complete, it is possible to reduce the costs of performing both addition and doubling through only the addition formulas. This decreases the amount of states and therefore makes the final implementation a lot more compact. Hence, the implementation only iterates over the addition formulas, until the end of the computations.

### Scheduling

The architecture presented in Figure 3.1 has one dual port memory, whereas it has three processors. This means that we can only load values to two processors at the same time. As a consequence the three processors do not run completely in parallel, but one of the three is unsynchronized. Table 3.3 showcases how operations are split into different processors. They are distributed with the goal of minimizing the number of loads and stores for each processor and to minimize `MM2` being idle. The process begins by loading the necessary values into `MM0` and `MM1` and executing their

Table 3.3: Scheduling for point addition $P \leftarrow P + Q$, where $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$. For doubling, simply put $P = Q$.

| Line # Alg. 13 | MM0 | MM1 | MM2 |
|---|---|---|---|
| 1 | $t_0 \leftarrow X_1 \cdot X_2$ | $t_1 \leftarrow Y_1 \cdot Y_2$ | |
| | | | $t_2 \leftarrow Z_1 \cdot Z_2$ |
| 2 | $t_3 \leftarrow X_1 + Y_1$ | $t_4 \leftarrow X_2 + Y_2$ | |
| | | | $t_5 \leftarrow Y_1 + Z_1$ |
| 3 | $t_7 \leftarrow X_1 + Z_1$ | $t_8 \leftarrow X_2 + Z_2$ | |
| | | | $t_6 \leftarrow Y_2 + Z_2$ |
| 4 | $t_9 \leftarrow t_3 \cdot t_4$ | $t_{11} \leftarrow t_7 \cdot t_8$ | |
| | | | $t_{10} \leftarrow t_5 \cdot t_6$ |
| 5 | $t_4 \leftarrow t_1 + t_2$ | $t_5 \leftarrow t_0 + t_2$ | |
| | | | $t_3 \leftarrow t_0 + t_1$ |
| 6,7,8 | $t_6 \leftarrow b_3 \cdot t_2$ | $t_8 \leftarrow a \cdot t_2$ | |
| | | | $t_2 \leftarrow t_9 - t_3$ |
| | | | $t_3 \leftarrow t_{10} - t_4$ |
| | | | $t_4 \leftarrow t_{11} - t_5$ |
| | | | $t_9 \leftarrow t_0 + t_0$ |
| | | | $t_{10} \leftarrow t_9 + t_0$ |
| 9 | $t_5 \leftarrow b_3 \cdot t_4$ | $t_{11} \leftarrow a \cdot t_4$ | |
| | | | $t_7 \leftarrow t_0 - t_8$ |
| | | | $t_9 \leftarrow a \cdot t_7$ |
| 10 | $t_0 \leftarrow t_8 + t_{10}$ | $t_4 \leftarrow t_{11} + t_6$ | |
| | | | $t_7 \leftarrow t_5 + t_9$ |
| 11 | $t_5 \leftarrow t_1 - t_4$ | $t_6 \leftarrow t_1 + t_4$ | |
| 12 | $t_4 \leftarrow t_0 \cdot t_7$ | $t_1 \leftarrow t_5 \cdot t_6$ | |
| | | | $t_8 \leftarrow t_3 \cdot t_7$ |
| 13 | $t_{11} \leftarrow t_0 \cdot t_2$ | $t_9 \leftarrow t_2 \cdot t_5$ | |
| | | | $t_{10} \leftarrow t_3 \cdot t_6$ |
| 14 | $Y_1 \leftarrow t_1 + t_4$ | $X_1 \leftarrow t_9 - t_8$ | |
| | | | $Z_1 \leftarrow t_{10} + t_{11}$ |

respective operations. As soon as the operations in MM0 and MM1 are initialized, it loads the corresponding value into MM2 and executes the operation. As soon as MM0 and MM1 finish their operations, this process restarts. Since the operations executed in MM2 are not synchronized with those in MM0 and MM1, both of the operations in MM0 and MM1 should be independent of the output of MM2, and vice versa. Furthermore, since multiplications are at least ten times slower than additions for our processor choice [85], the additions and subtractions from lines seven and eight in Algorithm 13 can be done by the otherwise idle processor MM2 in stage six. This makes them basically free of cost.

### Comparison

The architecture shown supports primes from 116 to 522 bits, therefore it is possible to benchmark it and compare for multiple bit sizes. The results for different common prime sizes are shown in Tables 3.4 and 3.5. Integer addition, subtraction and Montgomery modular multiplication results are the same as in Chapter 2. The new result is the scalar multiplication routine with the new complete formulas for elliptic curves in short Weierstrass form [111], and it takes about 14.21 ms for a 256-bit prime.

It is not straightforward to do a well-founded comparison with the relevant related work in the literature. Tables 3.4 and 3.5 contain different implementations of elliptic curve scalar multiplication, but they have different optimization goals and sizes, and we focus here on 256 bits primes. For example, this chapter work tops [134, 135] in terms of milliseconds per scalar multiplication, but they use less multipliers or run at a lower frequency. On the other hand the works presented in [4, 58, 121, 57, 91, 83] outperform this architecture in terms of speed, but use a much larger number of embedded multipliers. Also, implementations only focusing on NIST curves are able to use the special prime shape, yielding a significant speed-up. Depending on the needs of a specific hardware designer, this specialization of curves might not always be desirable. As mentioned before, many parties in industry might prefer generic cores. Despite these remarks, we argue that the implementation is competitive with the literature, making a similar trade-off between size and speed. Thus the new formulas can be implemented with little to no penalties, while having the benefit of not having to deal with exceptions.

## 3.4 Number of bits for the Montgomery representation

In our implementation we opted to not perform modular reduction after addition and subtractions, only after multiplications. In order to make this work, it is necessary to verify if the representation can handle the number of bits that increases in each non-reduced operation. The limits of the representation are the bounds, and in case of Montgomery multiplication, they need to fit in $r$-bits at the end of reduction, and during multiplication less than $r^2$-bits. In the original version of this publication [88] the values were not properly checked, but here we give a proper analysis and explicitly state the number of bits required .

When implementing the elliptic curve formulas, the bounds of the points coordinates need to be stable during the entire scalar multiplication. Because we opted to use the same function for point addition and point doubling, it is only needed to guarantee that input points $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$ have the same bounds as $P + Q = (X_3 : Y_3 : Z_3)$. After the scalar multiplication finishes, then we can use the bounds from point addition to verify if it is possible to convert it into affine representation, and later remove from the Montgomery domain.

Algorithm 14 shows the bounds for the implemented version of the complete formulas point addition. The input and output points are assumed to be $X_i, Y_i, Z_i \in [0, 6p], i \in \{1, 2, 3\}$ and the Montgomery constant $r > 64p$ (Chapter 1 and 2), thus $r$ needs at least 6 more bits than the prime size. While 6 more bits is enough to keep it stable, it is less than the biggest multiplication, which is $144p^2$. A more conservative choice would be for $r$ to be bigger than the biggest multiplication, thus $r > 256p$,

Table 3.4: Comparison of our results with the literature on hardware implementations for ECC in terms of resources.

| Work | Field | FPGA | Slice/ ALM | LUT | FF | Emb. Mult. | BRAM 64×18 | BRAM 1k×18 |
|---|---|---|---|---|---|---|---|---|
| For all prime order short Weierstrass curves | | | | | | | | |
| Our | 192 | IGLOO $2^4$ | – | 2828 | 1048 | 6 | 6 | 1 |
| Our | 224 | IGLOO $2^4$ | – | 2828 | 1048 | 6 | 6 | 1 |
| Our | 256 | IGLOO $2^4$ | – | 2828 | 1048 | 6 | 6 | 1 |
| Our | 320 | IGLOO $2^4$ | – | 2828 | 1048 | 6 | 6 | 1 |
| Our | 384 | IGLOO $2^4$ | – | 2828 | 1048 | 6 | 6 | 1 |
| Our | 512 | IGLOO $2^4$ | – | 2828 | 1048 | 6 | 6 | 1 |
| Our | 521 | IGLOO $2^4$ | – | 2828 | 1048 | 6 | 6 | 1 |
| For NIST curves [100] only | | | | | | | | |
| [134] | 224 | SmartFusion$^4$ | – | 3690 | 3690 | 0 | 0 | 12 |
| [134] | 256 | SmartFusion$^4$ | – | 3690 | 3690 | 0 | 0 | 12 |
| [134] | 224 | Virtex II Pro$^4$ | 773 | 1546$^\star$ | 1546$^\star$ | 1 | 0 | 3 |
| [134] | 256 | Virtex II Pro$^4$ | 773 | 1546$^\star$ | 1546$^\star$ | 1 | 0 | 3 |
| [134] | 224 | Virtex II Pro$^4$ | 1158 | 2316$^\star$ | 2316$^\star$ | 4 | 0 | 3 |
| [134] | 256 | Virtex II Pro$^4$ | 1158 | 2316$^\star$ | 2316$^\star$ | 4 | 0 | 3 |
| [108] | 256 | Virtex $5^6$♣ | 1914 | 7656$^\star$ | 7656$^\star$ | 4 | 0 | 12 |
| [48] | 192 | Virtex II Pro$^4$ | 3173 | 6346$^\star$ | 6346$^\star$ | 16 | 0 | 6 |
| [114] | 256 | Spartan $6^6$ | 72 | 193 | 35 | 8 | 0 | 24 |
| [80] | 192 | Virtex $4^4$ | 7020 | 12435 | 3545 | 8 | 0 | 4 |
| [80] | 224 | Virtex $4^4$ | 7020 | 12435 | 3545 | 8 | 0 | 4 |
| [80] | 256 | Virtex $4^4$ | 7020 | 12435 | 3545 | 8 | 0 | 4 |
| [80] | 384 | Virtex $4^4$ | 7020 | 12435 | 3545 | 8 | 0 | 4 |
| [80] | 521 | Virtex $4^4$ | 7020 | 12435 | 3545 | 8 | 0 | 4 |
| [4] | 192 | Virtex $6^6$♣ | 11.2k | 32.9k | 89.6k$^\star$ | 289 | 0 | 256 |
| [4] | 224 | Virtex $6^6$♣ | 11.2k | 32.9k | 89.6k$^\star$ | 289 | 0 | 256 |
| [4] | 256 | Virtex $6^6$♣ | 11.2k | 32.9k | 89.6k$^\star$ | 289 | 0 | 256 |
| [4] | 384 | Virtex $6^6$♣ | 11.2k | 32.9k | 89.6k$^\star$ | 289 | 0 | 256 |
| [4] | 521 | Virtex $6^6$♣ | 11.2k | 32.9k | 89.6k$^\star$ | 289 | 0 | 256 |
| [58] | 224 | Virtex $4^4$ | 1580 | 1825 | 1892 | 26 | 0 | 11 |
| [58] | 256 | Virtex $4^4$ | 1715 | 2589 | 2028 | 32 | 0 | 11 |
| For only Edwards or Twisted Edwards curves | | | | | | | | |
| [10] | 192 | Spartan 3E $^4$ | 4654 | 9308$^\star$ | 9308$^\star$ | 0 | 0 | 0 |
| [121] | 256 | Zynq$^6$♣ | 1029 | 2783 | 3592 | 20 | 0 | 4 |
| For only specific field size, but works with any prime | | | | | | | | |
| [135] | 256 | Virtex II Pro$^4$ | 1832 | 3664$^\star$ | 3664$^\star$ | 2 | 0 | 9 |
| [135] | 256 | Virtex II Pro$^4$ | 2085 | 4170$^\star$ | 4170$^\star$ | 7 | 0 | 9 |
| [57] | 192 | Stratix II$^4$ | 6203 | 12406$^\star$ | 12406$^\star$ | 92 | 0 | 0 |
| [57] | 256 | Stratix II$^4$ | 9177 | 18354$^\star$ | 18354$^\star$ | 96 | 0 | 0 |
| [57] | 384 | Stratix II$^4$ | 12958 | 25916$^\star$ | 25916$^\star$ | 177 | 0 | 0 |
| [57] | 512 | Stratix II$^4$ | 17017 | 34034$^\star$ | 34034$^\star$ | 244 | 0 | 0 |
| [90] | 256 | Virtex II Pro$^4$ | 15755 | 31510$^\star$ | 31510$^\star$ | 256 | 0 | 0 |
| [83] | 256 | Virtex $4^4$ | 4655 | 5740 | 4876 | 37 | 0 | 11 |

$^\star$ Maximum possible value assumed from the number of slices. Virtex II Pro and Spartan 3E slice is 2 LUTs and FFs, Virtex 5 is 4 LUTs and FFs, finally Virtex 6 is 4 LUTs and 8 FFs. Stratix II ALM can be configured into 2 LUTs and FFs.
$^4$ $^6$ indicates LUT size.
♣ BRAMs of Virtex 5, 6 and Zynq are 1k×36, so they account as 2 independent 1k×18.

Table 3.5: Comparison of our results to the literature on hardware implementations for ECC in terms of time. The speed results are for 1 scalar multiplication.

| Work | Field | FPGA | Freq. | Scalar Mult. | |
|---|---|---|---|---|---|
| | | | (MHz) | Cycles | (ms) |
| For all prime order short Weierstrass curves | | | | | |
| Our | 192 | IGLOO 2 | 100 | 728448 | 7.28 |
| Our | 224 | IGLOO 2 | 100 | 1036224 | 10.36 |
| Our | 256 | IGLOO 2 | 100 | 1421312 | 14.21 |
| Our | 320 | IGLOO 2 | 100 | 2498560 | 24.99 |
| Our | 384 | IGLOO 2 | 100 | 3744768 | 37.45 |
| Our | 512 | IGLOO 2 | 100 | 8187904 | 81.88 |
| Our | 521 | IGLOO 2 | 100 | 8331832 | 83.32 |
| For NIST curves [100] only | | | | | |
| [134] | 224 | SmartFusion | 109 | 1722088 | 15.8 |
| [134] | 256 | SmartFusion | 109 | 2103941 | 19.3 |
| [134] | 224 | Virtex II Pro | 210 | 1722088 | 8.2 |
| [134] | 256 | Virtex II Pro | 210 | 2103941 | 10.02 |
| [134] | 224 | Virtex II Pro | 210 | 765072 | 3.64 |
| [134] | 256 | Virtex II Pro | 210 | 949951 | 4.52 |
| [108] | 256 | Virtex 5 | 210 | 830000 | 3.95 |
| [48] | 192 | Virtex II Pro | 93 | 920700† | 9.90 |
| [114] | 256 | Spartan 6 | 156.25 | 1906250† | 12.2 |
| [80] | 192 | Virtex 4 | 182 | 429702† | 2.361 |
| [80] | 224 | Virtex 4 | 182 | 666666† | 3.663 |
| [80] | 256 | Virtex 4 | 182 | 993174† | 5.457 |
| [80] | 384 | Virtex 4 | 182 | 2968420† | 16.31 |
| [80] | 521 | Virtex 4 | 182 | 7048860† | 38.73 |
| [4] | 192 | Virtex 6 | 100 | 29948 | 0.30 |
| [4] | 224 | Virtex 6 | 100 | 34999 | 0.35 |
| [4] | 256 | Virtex 6 | 100 | 39922 | 0.40 |
| [4] | 384 | Virtex 6 | 100 | 11722 | 1.18 |
| [4] | 521 | Virtex 6 | 100 | 159959 | 1.60 |
| [58] | 224 | Virtex 4 | 487 | 219878 | 0.451 |
| [58] | 256 | Virtex 4 | 490 | 303450 | 0.619 |
| For only Edwards or Twisted Edwards curves | | | | | |
| [10] | 192 | Spartan 3E | 10 | 125430† | 12.543 |
| [121] | 256 | Zynq | 200 | 64770 | 0.324 |
| For only specific field size, but works with any prime | | | | | |
| [135] | 256 | Virtex II Pro | 108.2 | 3227993 | 29.83 |
| [135] | 256 | Virtex II Pro | 68.17 | 1074625 | 15.76 |
| [57] | 192 | Stratix II | 160.5 | 70620† | 0.44 |
| [57] | 256 | Stratix II | 157.2 | 106896† | 0.68 |
| [57] | 384 | Stratix II | 150.9 | 203715† | 1.35 |
| [57] | 512 | Stratix II | 144.97 | 323283† | 2.23 |
| [90] | 256 | Virtex II Pro | 39.46 | 151360 | 3.86 |
| [83] | 256 | Virtex 4 | 250 | 109297 | 0.44 |

† Values estimated by multiplying time by frequency.

however as we can see in Algorithm 14, this is not necessary, since the output of the multiplication is later reduced in other multiplications.

The bounds for the addition were computed by adding the lowest and biggest bounds together, the subtraction is done the same way but subtracting the smallest to get the biggest and vice versa. The multiplication bounds were computed using the relation in Theorem 1 in Walter's paper [137]:

$$a \cdot b/r \leq o < p + a \cdot b/r$$

In our case we know that $r > 64p$, thus we can replace $r$ and round the division to the worst integer values.

$$\lfloor (a \cdot b/64p) \rfloor \leq o \leq p + \lceil (a \cdot b/64p) \rceil$$

**Algorithm 14** Parallelized complete addition formulas for a prime order elliptic curve in Weierstrass form, using *three* processors with values bounds.

| | | | | | |
|---|---|---|---|---|---|
| | $t_0 \leftarrow X_1 \cdot X_2;$ | $[0, 36p^2] \rightarrow [0, 2p]$ | | $t_3 \leftarrow X_1 + Y_1;$ | $[0, 12p]$ |
| 1. | $t_1 \leftarrow Y_1 \cdot Y_2;$ | $[0, 36p^2] \rightarrow [0, 2p]$ | 2. | $t_4 \leftarrow X_2 + Y_2;$ | $[0, 12p]$ |
| | $t_2 \leftarrow Z_1 \cdot Z_2;$ | $[0, 36p^2] \rightarrow [0, 2p]$ | | $t_5 \leftarrow Y_1 + Z_1;$ | $[0, 12p]$ |
| | $t_6 \leftarrow Y_2 + Z_2;$ | $[0, 12p]$ | | $t_9 \leftarrow t_3 \cdot t_4;$ | $[0, 144p^2] \rightarrow [0, 4p]$ |
| 3. | $t_7 \leftarrow X_1 + Z_1;$ | $[0, 12p]$ | 4. | $t_{10} \leftarrow t_5 \cdot t_6;$ | $[0, 144p^2] \rightarrow [0, 4p]$ |
| | $t_8 \leftarrow X_2 + Z_2;$ | $[0, 12p]$ | | $t_{11} \leftarrow t_7 \cdot t_8;$ | $[0, 144p^2] \rightarrow [0, 4p]$ |
| | $t_3 \leftarrow t_0 + t_1;$ | $[0, 4p]$ | | | |
| 5. | $t_4 \leftarrow t_1 + t_2;$ | $[0, 4p]$ | 6. | $t_6 \leftarrow b_3 \cdot t_2;$ | $[0, 12p^2] \rightarrow [0, 2p]$ |
| | $t_5 \leftarrow t_0 + t_2;$ | $[0, 4p]$ | | $t_8 \leftarrow a \cdot t_2;$ | $[0, 12p^2] \rightarrow [0, 2p]$ |
| | $t_2 \leftarrow t_9 - t_3 + 4p;$ | $[0, 8p]$ | | $t_{10} \leftarrow t_9 + t_0;$ | $[0, 6p]$ |
| 7. | $t_9 \leftarrow t_0 + t_0;$ | $[0, 4p]$ | 8. | $t_4 \leftarrow t_{11} - t_5 + 4p;$ | $[0, 8p]$ |
| | $t_3 \leftarrow t_{10} - t_4 + 4p;$ | $[0, 8p]$ | | $t_7 \leftarrow t_0 - t_8 + 4p;$ | $[2p, 6p]$ |
| | $t_0 \leftarrow a \cdot t_4;$ | $[0, 48p^2] \rightarrow [0, 2p]$ | | $t_4 \leftarrow t_0 + t_6;$ | $[0, 4p]$ |
| 9. | $t_5 \leftarrow b_3 \cdot t_4;$ | $[0, 48p^2] \rightarrow [0, 2p]$ | 10. | $t_7 \leftarrow t_5 + t_9;$ | $[0, 4p]$ |
| | $t_9 \leftarrow a \cdot t_7;$ | $[0, 36p^2] \rightarrow [0, 2p]$ | | $t_0 \leftarrow t_8 + t_{10};$ | $[0, 8p]$ |
| | | | | $t_1 \leftarrow t_5 \cdot t_6;$ | $[0, 36p^2] \rightarrow [0, 2p]$ |
| 11. | $t_5 \leftarrow t_1 - t_4 + 4p;$ | $[0, 6p]$ | 12. | $t_4 \leftarrow t_0 \cdot t_7;$ | $[0, 32p^2] \rightarrow [0, 2p]$ |
| | $t_6 \leftarrow t_1 + t_4;$ | $[0, 6p]$ | | $t_8 \leftarrow t_3 \cdot t_7;$ | $[0, 32p^2] \rightarrow [0, 2p]$ |
| | $t_9 \leftarrow t_2 \cdot t_5;$ | $[0, 48p^2] \rightarrow [0, 2p]$ | | $X_3 \leftarrow t_9 - t_8 + 4p;$ | $[2p, 6p]$ |
| 13. | $t_{10} \leftarrow t_3 \cdot t_6;$ | $[0, 48p^2] \rightarrow [0, 2p]$ | 14. | $Y_3 \leftarrow t_1 + t_4;$ | $[0, 4p]$ |
| | $t_{11} \leftarrow t_0 \cdot t_2;$ | $[0, 64p^2] \rightarrow [0, 2p]$ | | $Z_3 \leftarrow t_{10} + t_{11};$ | $[0, 4p]$ |

While it might be possible to have more precise bounds, these bounds are easy to compute and verify. For example if $a, b \in [2p, 6p]$, then we have:

$$\begin{aligned} \lfloor (2p \cdot 2p/64p) \rfloor \leq o &\leq p + \lceil (6p \cdot 6p/64p) \rceil \\ \lfloor (4p/64) \rfloor \leq o &< p + \lceil (36p/64) \rceil \\ p \leq o &< p + p \end{aligned} \tag{3.2}$$

Therefore $o \in [p, 2p]$.

After the scalar multiplication the final points coordinates will be within $[0, 6p]$ and since $6p < (64p/4)$, therefore if we remove from the Montgomery domain, the output will be within $[0, p]$. However, usually after a scalar multiplication the next step is to convert the output point to affine coordinates, and thus we might keep it in the Montgomery domain for this conversion.

### Sloth reduction and bounds for other complete formulas

Scott [124] has given a name to the technique of not performing reductions on addition and subtraction as "Sloth reduction", an analogy to the already-known lazy reduction [123]. In the same paper, Scott also introduced the number of necessary

**Algorithm 15** Complete addition formula with size bounds for a prime order elliptic curve in Weierstrass form for curve constant $a = -3$.

| | | | | | |
|---|---|---|---|---|---|
| 1. | $t_0 \leftarrow X_1 \cdot X_2;$ | $[0, 64p^2] \rightarrow [0, 2p]$ | 2. | $t_1 \leftarrow Y_1 \cdot Y_2;$ | $[0, 64p^2] \rightarrow [0, 2p]$ |
| 3. | $t_2 \leftarrow Z_1 \cdot Z_2;$ | $[0, 64p^2] \rightarrow [0, 2p]$ | 4. | $t_3 \leftarrow X_1 + Y_1;$ | $[0, 16p]$ |
| 5. | $t_4 \leftarrow X_2 + Y_2;$ | $[0, 16p]$ | 6. | $t_3 \leftarrow t_3 \cdot t_4;$ | $[0, 256p^2] \rightarrow [0, 2p]$ |
| 7. | $t_4 \leftarrow t_0 + t_1;$ | $[0, 4p]$ | 8. | $t_3 \leftarrow t_3 - t_4 + 4p;$ | $[0, 6p]$ |
| 9. | $t_4 \leftarrow Y_1 + Z_1;$ | $[0, 16p]$ | 10. | $t_5 \leftarrow Y_2 + Z_2;$ | $[0, 16p]$ |
| 11. | $t_4 \leftarrow t_4 \cdot t_5;$ | $[0, 256p^2] \rightarrow [0, 2p]$ | 12. | $t_5 \leftarrow t_1 + t_2;$ | $[0, 4p]$ |
| 13. | $t_4 \leftarrow t_4 - t_5 + 4p;$ | $[0, 6p]$ | 14. | $t_5 \leftarrow X_1 + Z_1;$ | $[0, 16p]$ |
| 15. | $t_6 \leftarrow X_2 + Z_2;$ | $[0, 16p]$ | 16. | $t_5 \leftarrow t_5 \cdot t_6;$ | $[0, 256p^2] \rightarrow [0, 2p]$ |
| 17. | $t_6 \leftarrow t_0 + t_2;$ | $[0, 4p]$ | 18. | $t_6 \leftarrow t_5 - t_6 + 4p;$ | $[0, 6p]$ |
| 19. | $t_7 \leftarrow b \cdot t_2;$ | $[0, 2p^2] \rightarrow [0, 2p]$ | 20. | $t_5 \leftarrow t_6 - t_7 + 2p;$ | $[0, 8p]$ |
| 21. | $t_7 \leftarrow t_5 + t_5;$ | $[0, 16p]$ | 22. | $t_5 \leftarrow t_5 + t_7;$ | $[0, 24p]$ |
| 23. | $t_7 \leftarrow t_1 - t_5 + 24p;$ | $[0, 26p]$ | 24. | $t_5 \leftarrow t_1 + t_5;$ | $[0, 26p]$ |
| 25. | $t_6 \leftarrow b \cdot t_6;$ | $[0, 12p^2] \rightarrow [0, 2p]$ | 26. | $t_1 \leftarrow t_2 + t_2;$ | $[0, 4p]$ |
| 27. | $t_2 \leftarrow t_1 + t_2;$ | $[0, 6p]$ | 28. | $t_6 \leftarrow t_6 - t_2 + 6p;$ | $[0, 8p]$ |
| 29. | $t_6 \leftarrow t_6 - t_0 + 2p;$ | $[0, 10p]$ | 30. | $t_1 \leftarrow t_6 + t_6;$ | $[0, 20p]$ |
| 31. | $t_6 \leftarrow t_1 + t_6;$ | $[0, 30p]$ | 32. | $t_1 \leftarrow t_0 + t_0;$ | $[0, 4p]$ |
| 33. | $t_0 \leftarrow t_1 + t_0;$ | $[0, 6p]$ | 34. | $t_0 \leftarrow t_0 - t_2 + 2p;$ | $[0, 12p]$ |
| 35. | $t_1 \leftarrow t_4 \cdot t_6;$ | $[0, 180p^2] \rightarrow [0, 2p]$ | 36. | $t_2 \leftarrow t_0 \cdot t_6;$ | $[0, 360p^2] \rightarrow [0, 3p]$ |
| 37. | $t_6 \leftarrow t_5 \cdot t_7;$ | $[0, 676p^2] \rightarrow [0, 4p]$ | 38. | $Y_3 \leftarrow t_6 + t_2;$ | $[0, 7p]$ |
| 39. | $t_5 \leftarrow t_3 \cdot t_5;$ | $[0, 156p^2] \rightarrow [0, 2p]$ | 40. | $X_3 \leftarrow t_5 - t_1 + 2p;$ | $[0, 4p]$ |
| 41. | $t_7 \leftarrow t_4 \cdot t_7;$ | $[0, 156p^2] \rightarrow [0, 2p]$ | 42. | $t_1 \leftarrow t_3 \cdot t_0;$ | $[0, 72p^2] \rightarrow [0, 2p]$ |
| 43. | $Z_3 \leftarrow t_7 + t_1;$ | $[0, 4p]$ | | | |

extra bits for different elliptic curve point addition/doubling formulas, and one of them being the ones from Renes et al. [111]. In his analysis, the number of extra bits necessary is 10, but it is possible to be less conservative and do it with 8 bits. Algorithms 15 and 16 shows the bounds for point addition formula, assuming the input and output points to be $X_i, Y_i, Z_i \in [0, 8p], i \in \{1, 2, 3\}$.

Reducing the number of bits from 8 to 10 can reduce the amount of resources employed, since the elliptic curve coordinates and scalars to be stored in memory will require the prime size plus 8 more instead of 10. For example in an 256-bit prime order elliptic storing each coordinate will require a register of 264 bits. If we store all the elliptic curve coordinates in a 8 bits words memory, then the entire 264 bits value can fit into 33 words, instead of 34 words if it was used with 10 extra bits. However, if the memory employed has words of more than 10 bits, then the reduction from 8 to 10 probably has no impact.

## 3.5 Final considerations of the chapter

This chapter shows a proof of concept elliptic curve scalar multiplication with the parallelized complete formulas of Renes et al. [111], while all paralleized version can be found in Rene's thesis [110]. The formulas can be applied not only in hardware architectures with a great array of processors, but also in software implementations that are using vector instructions. As shown in our proof of concept implementation, the formulas have competitive results with other implementations in the literature.

**Algorithm 16** Complete doubling formula with size bounds for a prime order elliptic curve in Weierstrass form for curve constant $a = -3$.

| | | | | | |
|---|---|---|---|---|---|
| 1 | $t_0 \leftarrow X_1 \cdot X_1;$ | $[0, 64p^2] \rightarrow [0, 2p]$ | 2 | $t_1 \leftarrow Y_1 \cdot Y_1;$ | $[0, 64p^2] \rightarrow [0, 2p]$ |
| 3 | $t_2 \leftarrow Z_1 \cdot Z_1;$ | $[0, 64p^2] \rightarrow [0, 2p]$ | 4 | $t_3 \leftarrow X_1 \cdot Y_1;$ | $[0, 64p^2] \rightarrow [0, 2p]$ |
| 5 | $t_3 \leftarrow t_3 + t_3;$ | $[0, 4p]$ | 6 | $t_6 \leftarrow X_1 \cdot Z_1;$ | $[0, 64p^2] \rightarrow [0, 2p]$ |
| 7 | $t_6 \leftarrow t_6 + t_6;$ | $[0, 4p]$ | 8 | $t_5 \leftarrow b \cdot t_2;$ | $[0, 2p^2] \rightarrow [0, 2p]$ |
| 9 | $t_5 \leftarrow t_5 - t_6 + 4p;$ | $[0, 6p]$ | 10 | $t_4 \leftarrow t_5 + t_5;$ | $[0, 12p]$ |
| 11 | $t_5 \leftarrow t_4 + t_5;$ | $[0, 18p]$ | 12 | $t_4 \leftarrow t_1 - t_5 + 18p;$ | $[0, 20p]$ |
| 13 | $t_5 \leftarrow t_1 + t_5;$ | $[0, 20p]$ | 14 | $t_5 \leftarrow t_4 \cdot t_5;$ | $[0, 400p^2] \rightarrow [0, 3p]$ |
| 15 | $t_4 \leftarrow t_4 \cdot t_3;$ | $[0, 80p^2] \rightarrow [0, 2p]$ | 16 | $t_3 \leftarrow t_2 + t_2;$ | $[0, 4p]$ |
| 17 | $t_2 \leftarrow t_2 + t_3;$ | $[0, 6p]$ | 18 | $t_6 \leftarrow b \cdot t_6;$ | $[0, 4p^2] \rightarrow [0, 2p]$ |
| 19 | $t_6 \leftarrow t_6 - t_2 + 6p;$ | $[0, 8p]$ | 20 | $t_6 \leftarrow t_6 - t_0 + 2p;$ | $[0, 10p]$ |
| 21 | $t_3 \leftarrow t_6 + t_6;$ | $[0, 20p]$ | 22 | $t_6 \leftarrow t_6 + t_3;$ | $[0, 30p]$ |
| 23 | $t_3 \leftarrow t_0 + t_0;$ | $[0, 4p]$ | 24 | $t_0 \leftarrow t_3 + t_0;$ | $[0, 6p]$ |
| 25 | $t_0 \leftarrow t_0 - t_2 + 6p;$ | $[0, 12p]$ | 26 | $t_0 \leftarrow t_0 \cdot t_6$ | $[0, 360p^2] \rightarrow [0, 3p]$ |
| 27 | $Y_3 \leftarrow t_5 + t_0;$ | $[0, 6p]$ | 28 | $t_0 \leftarrow Y_1 \cdot Z_1;$ | $[0, 64p^2] \rightarrow [0, 2p]$ |
| 29 | $t_0 \leftarrow t_0 + t_0;$ | $[0, 4p]$ | 30 | $t_6 \leftarrow t_0 \cdot t_6;$ | $[0, 120p^2] \rightarrow [0, 2p]$ |
| 31 | $X_3 \leftarrow t_4 - t_6 + 2p;$ | $[0, 4p]$ | 32 | $t_6 \leftarrow t_0 \cdot t_1;$ | $[0, 8p^2] \rightarrow [0, 2p]$ |
| 33 | $t_6 \leftarrow t_6 + t_6;$ | $[0, 4p]$ | 34 | $Z_3 \leftarrow t_6 + t_6;$ | $[0, 8p]$ |

Since our implementation is still a proof of concept, several further optimizations could be made to achieve even better results.

# Chapter 4

# HW-SIKE

In this chapter, a hardware and software co-design for the cryptosystem SIKE, based on isogenies of supersingular elliptic curves, is described and evaluated. To better understand the implementation, first, some details of SIKE and SIDH cryptosystems are given. Throughout this chapter, the notation followed is from the SIKE proposal [8].

**Chapter organization.**

This chapter starts with preliminaries on SIDH and SIKE given in Section 4.1. Later, in Section 4.2 we show the proposed SIKE hardware and software co-design architecture. Then Section 4.3 compares our proposal with previous SIKE literature and other post-quantum cryptosystem implementations. Finally, Section 4.4 contains the final considerations for this chapter with the full instruction set displayed at the end.

## 4.1 Preliminaries for SIDH and SIKE

This section starts by introducing the notation used in this chapter, followed by a brief explanation of the cryptosystem SIDH and then SIKE.

### Notations and definitions

The symbol $\cong$ is defining the congruence between two algebraic structures. Given an elliptic curve in Weierstrass form $E/\mathbb{F}_q : y^2 = x^3 + ax + b$ or in Montgomery form $E/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$ and let $\mathbb{F}_q$ be a field with a prime characteristic $p$, then the elliptic curve $E$ is supersingular if $p \mid (q + 1 - \#E(\mathbb{F}_q))$ [8] and otherwise it is ordinary. The $j$-invariant of an elliptic curve $E$, which is unique up to isomorphism of the elliptic curve can be computed as [8]:

$$j(E) = \frac{256(A^2 - 3)^3}{A^2 - 4}$$

The set of points $E[m]$ for a positive integer $m$ is called "$m$-torsion elements" if the same set of points instantiated over $E(\bar{\mathbb{F}}_q)$ have $[m]P_i = \mathcal{O}$ [8]. An isogeny $\phi : E_1 \to E_2$ is a non-constant rational map defined over $\mathbb{F}_q$ between two elliptic

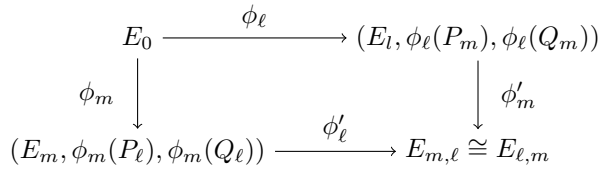



Figure 4.1: Supersingular isogeny Diffie-Hellman key exchange (SIDH).

curves $E_1$ and $E_2$ that have the same order $\#E_1(\mathbb{F}_q) = \#E_2(\mathbb{F}_q)$ [8]. Given an isogeny $\phi : E_1 \to E_2$, the kernel can be defined as [8]:

$$ker(\phi) = \{P \in E_1 : \phi(P) = \mathcal{O}\}$$

The inside brackets $\langle, \rangle$ notation is defined according to the SIKE specification [8] as: "For any group $G$, and a set of elements $\{P_1, P_2, \ldots, P_t\} \subseteq G$ we can define the subgroup $\langle P_1, P_2, ...P_t\rangle$ generated by this set to be the smallest subgroup of $G$ containing the elements $P_1, P_2, ...P_t$".

## SIDH

Let $e_2, e_3$ be positive integers such that $p = 2^{e_2}3^{e_3} - 1$ is prime. Let $E_0/\mathbb{F}_p : y^2 = x^3 + 6x^2 + x$ be an elliptic curve, which is supersingular [127, Theorem V.3.1(a)] as $p \equiv 3 \bmod 4$. Then one can show that $\#E_0(\mathbb{F}_{p^2}) = (p+1)^2$, and in particular $E_0(\mathbb{F}_{p^2})[2^{e_2}] \cong (\mathbb{Z}/2^{e_2}\mathbb{Z}) \times (\mathbb{Z}/2^{e_2}\mathbb{Z})$ and $E_0(\mathbb{F}_{p^2})[3^{e_3}] \cong (\mathbb{Z}/3^{e_3}\mathbb{Z}) \times (\mathbb{Z}/3^{e_3}\mathbb{Z})$ [127, Corollary III.6.4], where $(\mathbb{Z}/l^e\mathbb{Z})$ is the cyclic subgroup of the integers with order $l^e$.

Let Alice choose $m \in \{2, 3\}$ and fix basis points $P_m, Q_m \in E_0(\mathbb{F}_{p^2})$ such that $E_0(\mathbb{F}_{p^2})[m^{e_m}] = \langle P_m, Q_m\rangle$. As a result, any $sk_m \in \mathbb{Z}/m^{e_m}\mathbb{Z}$ defines a unique subgroup $\langle P_m + [sk_m]Q_m\rangle \subset E_0(\mathbb{F}_{p^2})[m^{e_m}]$ of order $m^{e_m}$. In turn, this determines (up to isomorphism) an elliptic curve $E_m = E_0/\langle P_m + [sk_m]Q_m\rangle$ and an $m^{e_m}$-isogeny $\phi_m : E_0 \to E_m$. We then define Alice's secret key to be $sk_m$ while her public key is chosen to be $pk_m = (E_m, \phi_m(P_\ell), \phi_m(Q_\ell))$, where $P_\ell, Q_\ell \in E_0(\mathbb{F}_{p^2})$ are Bob's basis points. Bob proceeds analogously with $\ell \in \{2, 3\}$ such that $\ell \neq m$. We define Bob's secret key to be $sk_\ell$ while his public key is chosen to be $pk_\ell = (E_l, \phi_\ell(P_m), \phi_\ell(Q_m))$.

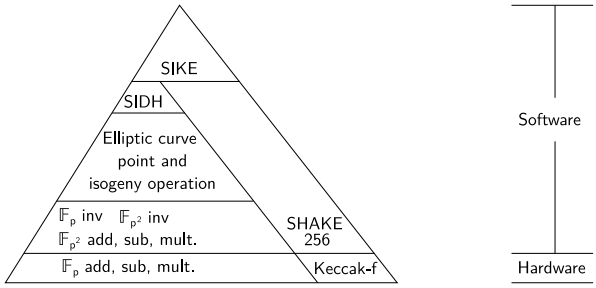
To compute a shared secret, Alice and Bob then proceed according to Figure 4.1, where

$$\ker \phi'_m = \langle \phi_\ell(P_m) + [sk_m]\phi_\ell(Q_m)\rangle\,, \quad \ker \phi'_\ell = \langle \phi_m(P_\ell) + [sk_\ell]\phi_m(Q_\ell)\rangle\,.$$

Correctness follows from the fact that $E_{\ell,m}$ and $E_{m,\ell}$ are both the co-domain of the isogeny from $E_0$ with kernel $\langle P_m + [sk_m]Q_m, P_\ell + [sk_\ell]Q_\ell\rangle$, which is unique up to post-composition with an isomorphism [127, Exercise III.3.13(e)]. Therefore $K = j(E_{\ell,m}) = j(E_{m,\ell})$ can be used as the shared secret.

The resulting shared key $K$ is indistinguishable from a random value under a chosen-plaintext attack (IND-CPA) assuming the Supersingular Decision Diffie-Hellman (SSDDH) problem [50, Problem 5.4] is hard. However, there is no active security as demonstrated by the adaptive attacks by Galbraith et al. [53] that allow the recovery of the secret key. As a result, one can only securely exchange keys with ephemeral public keys.

Figure 4.2: The high-level procedural hierarchy of the SIKE processor.




### SIKE

To obtain active security one can apply a standard transformation to transform the scheme into an IND-CCA secure key encapsulation mechanism (KEM) referred to as SIKE [8]. This is achieved in two steps; first, one transforms the IND-CPA secure key exchange into an IND-CPA secure public-key encryption (PKE) using hashed ElGamal [8, Proposition 1], after which one applies a transformation à la Fujisaki–Okamoto [52, 60] to obtain a secure key encapsulation mechanism (KEM) indistinguishable under chosen-ciphertext attacks (IND-CCA). These transformations do not significantly affect the efficiency of the protocol, and we refer to the SIKE specification [8] for more details.

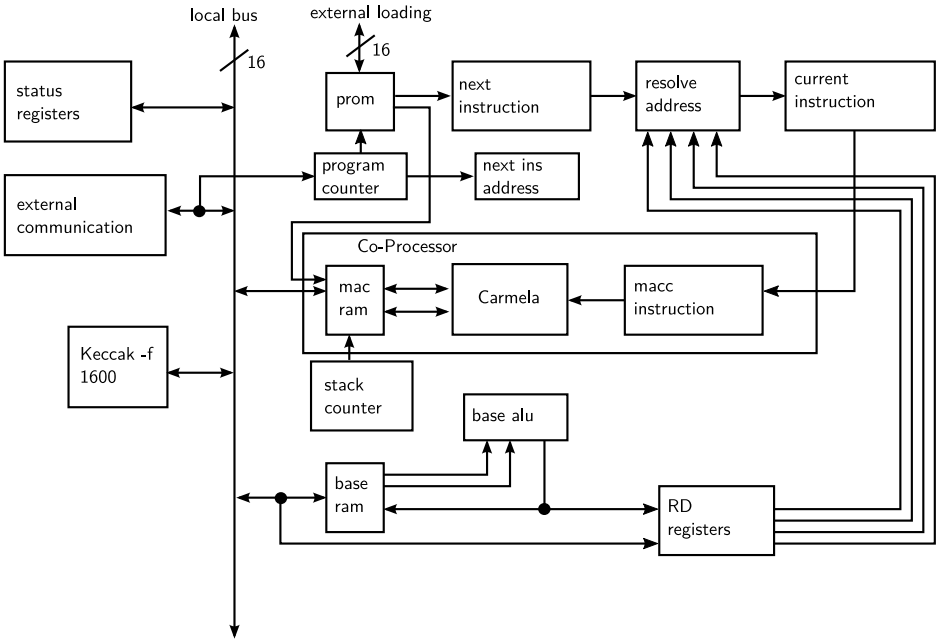
For this work, we have implemented the parameter sets `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751` defined in [8], and we precisely follow their design decisions. This includes the use of Montgomery representation [35, 110] and performing all arithmetic on the Kummer line [37]. Again, for details, we refer to [8, §1.3].

## 4.2 Proposed scalable architecture

The SIKE key encapsulation protocol is a complex algorithm that involves many different operations. Therefore, we opted for hardware/software co-design, instead of an approach solely consisting of a control unit plus combinational circuitry. In this way, we split the computational aspects of the protocol into hardware and software logic, as shown in Figure 4.2. By running the software on top of the fast hardware instructions, we gain flexibility and facilitate maintenance and debugging, while keeping an optimized hardware execution. All the low-level $\mathbb{F}_p$ operations (i.e., multiplication, squaring, and addition/subtraction) are implemented directly in hardware, while the inversion in $\mathbb{F}_p$ is done in software. The remainder operations are built on top of the $\mathbb{F}_p$ arithmetic (i.e., the $\mathbb{F}_{p^2}$ computations, the elliptic-curve point and isogeny algorithms, and the SIDH algorithm itself) are implemented in software. Finally, SIKE can be obtained by applying the same methodology where, additionally, Keccak [23] is implemented in hardware, and the necessary SHAKE [101] calls are done in software.

Our architecture, depicted in Figure 4.3, is split into two big parts. The first part is the Multiplier Accumulator (MACC) unit, called `Carmela`, and its respective memory (MACC RAM). It is responsible for the instructions related to the $\mathbb{F}_p$ operations,

Figure 4.3: A high-level architecture of the SIKE processor.

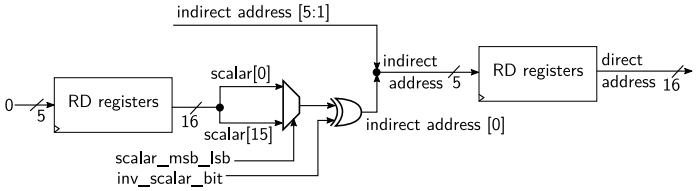



such as multiplication and addition/subtraction. This unit does not have an internal memory for the input and output operands and, therefore, needs the MACC RAM to work. The MACC RAM is composed of two true dual-port RAMs, which together can perform two reads and one write, or two writes, in one cycle. As this unit only performs the arithmetic operations, all the control flow instructions are handled by the main unit and its base ALU and base RAM. Thus, `Carmela` behaves like a coprocessor of the main unit.

The second part (i.e., the main unit) includes the base ALU, the base RAM, the RD registers, address resolution, the local bus, the status registers, and the program counter. It is a 16-bit signed/unsigned integer processor that can perform direct and conditional jumps, stack push and pop, copy values through the local bus, load values directly into the memories, call/return program functions, and execute arithmetic operations such as addition/subtraction and multiplication, and logical operations such as AND, NOT, OR, XOR. The instruction set was crafted to provide the necessary operations for the SIDH functions, but should also work with other programs or procedures. Future work could involve the change of the instruction set to a well-known and widely adopted ISA, such as RISC-V.

The RD registers unit works as a cache for the first 32 values of the base RAM. This was done because those registers can be read asynchronously, while the base RAM values have 1 cycle of delay. These values are mainly used as memory pointers for the MACC or the main processor operations. Therefore, it is possible to access an operand that is located on the base RAM or the MACC RAM pointed to by the first 32 positions of the base RAM itself. The first RD register has a special purpose

Figure 4.4: Description of how RD register 0 is used to make an indirect address from one bit and one extra word.




(see Figure 4.4): bit 0 or 15 can be used in conjunction with another pointer register to point to a position that is dependent on one of these bits. For example, if there are two pointers stored in positions 4 and 5, respectively, the indirect memory access can take the pointer stored in position $(10S)_2$, where $S$ is bit 15 or 0 of the RD register.

The base Arithmetic Logic Unit (ALU) is the FPGA Digital Signal Processor (DSP) that can perform addition, subtraction, and multiplication. Comparisons are accomplished via the DSP subtraction operation. Finally, the shifts and rotations are done through the mux-based data-reversal barrel shift as showed in Pillmeier et al. [107].

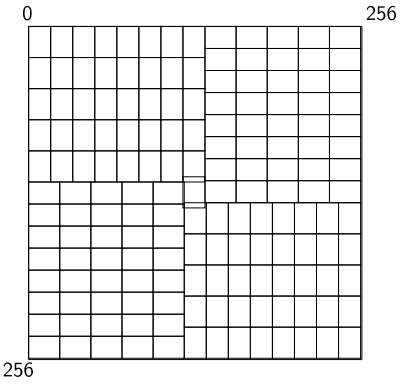
The PROM unit holds all the instructions to be executed by the main unit and the MACC unit. An instruction first has its address resolved and is then executed by the main or MACC unit, respectively. In case it is a MACC instruction, the main unit serially loads 8 instructions (in case of multiplications or instructions that use the multiplier) or 4 instructions (for additions/subtractions). The PROM unit can be accessed through an external loading bus, which is the same as the small bus for external communication. To distinguish between loading and reading the small bus and the PROM memory, an extra external signal is used alongside the address position. This can be seen as the memory system having a 17-bit address space, instead of 16 bits.

For the Keccak core, we use Bertoni et al.'s VHDL implementation, specifically their *middle-area coprocessor* option [22]. We made some adaptations and added an interface for our small bus, in which we can absorb and squeeze data and perform all 24 Keccak rounds. To obtain the SHAKE-256 [101] output, we use the main unit to prepare the data and only use the Keccak core for the permutation itself.

### Multiplier-accumulator unit: Carmela

We have targeted two distinct architectures; the first relies on a 128-bit MACC (called `Carmela128`), while the second is built upon a 256-bit MACC unit (called `Carmela256`). The design principles are essentially identical, the only difference being the relative size of the units. For that reason, we shall mostly only provide detailed descriptions of the architecture based on `Carmela256` and expect that an understanding of the `Carmela128`-based architecture can be easily inferred, though we highlight the differences wherever necessary. We emphasize that both MACC units were designed with compactness and scalability in mind, but `Carmela128` of course does so more aggressively.

Figure 4.5: The 257-bit signed multiplier, based on the 256-bit multiplier from [113].




**Carmela256.** The 256-bit MACC unit features an 8-stage pipeline architecture with 6 stages for the 257-bit signed multiplier and an optimized 546-bit adder. Although the multiplier is constructed with 257 signed bits, it is used as a 256-bit multiplier for both the signed and unsigned cases. Therefore, one of the bits is not fully used in the signed case in exchange for a single bus of 256 bits and a single 256-bit memory word. Indicating whether the input value is signed or not is done by the state machine controlling `Carmela256` directly. The adder, on the other hand, is 546-bit since it is $257 \times 2 + 32$. These extra 32 bits for the accumulator are necessary to perform successive accumulation of several multiplications. While 32 bits are more than strictly necessary to fit the biggest multiplication algorithm of 1024 bits, it would also be enough for potentially different but similar algorithms (as the arithmetic operations of SIKE could still be developed further, possibly requiring more additions).

The 257-bit signed multiplier is constructed from small multipliers with the non-standard tiling technique from Roy et al. [113]. The tiling technique exploits the rectangular shape of the multipliers on more modern FPGA models, which are $25 \times 18$-bit signed multipliers. This is done by visualizing the multiplication as a big square, or even a rectangle, and then trying to fill the shape with the minimum amount of rectangles, and being allowed to rotate the rectangles by 90 degrees. In our case we need a 257-bit signed multiplier, which can be done by extending the 256-bit unsigned multiplier of Roy et al., with an additional one-bit multiplier to solve the sign. This results in the tiling shown in Figure 4.5, which has the two extra 1 bit multipliers (though it can be difficult to notice since Figure 4.5 is on scale).

After multiplication and generation of all partial products, it is necessary to sum and compress into a single value. For this task, we employ a 30:5 compressor, followed by a 5:3 compressor, then a 3:2 compressor, and finished with an adder. The entire compression tree of 30:1 has intermediate registers plus 2 extra registers at the end, as shown in Figure 4.6. Those last registers are believed to be rebalanced by the FPGA synthesis tool since this is not the bottleneck of the maximum period of the circuit.

The final adder in the multiplication was done with a special adder, instead of the carry propagation adder inferred by the synthesis tool. This special adder architecture

Figure 4.6: The 257-bit signed multiplier with compressor.

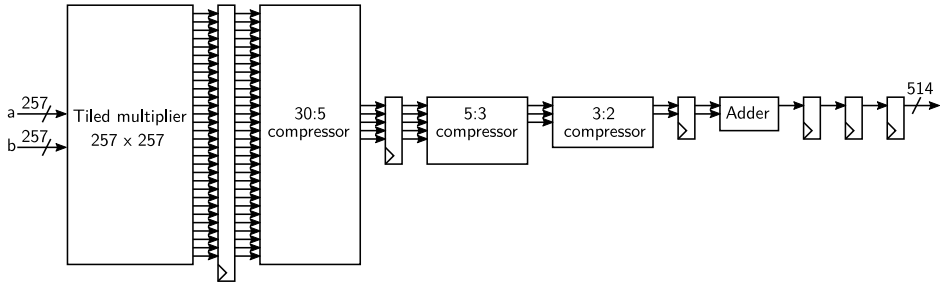
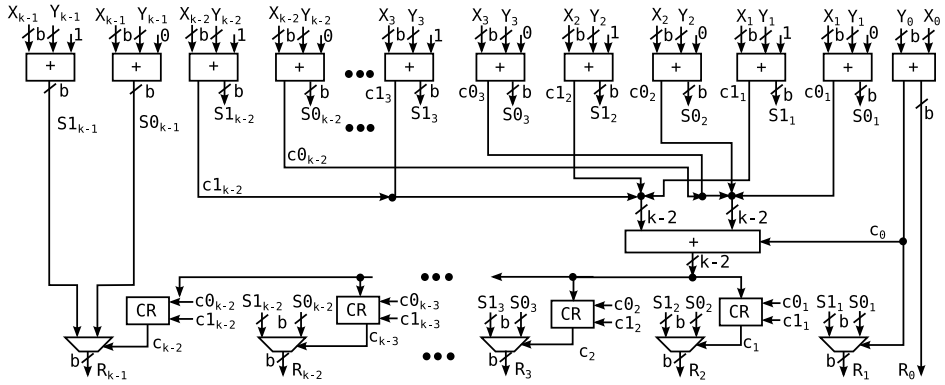



Figure 4.7: The optimized wide adder architecture Add-Add-Multiplex (AAM) from Nguyen et al. [102]. In our implementation b = 2.

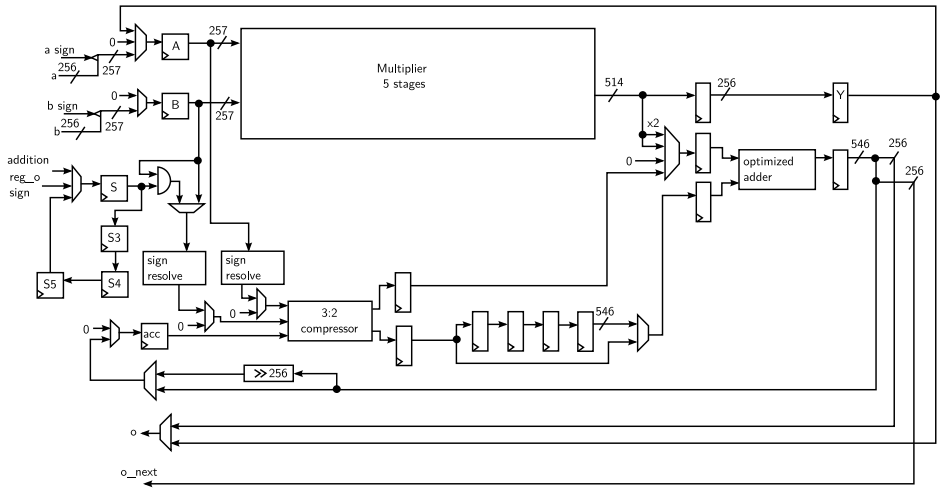



called Add-Add-Multiplex (AAM) is given in Figure 4.7 and the architecture was originally proposed by Nguyen et al. [102]. The adder is a carry-select architecture, where both options for the carry are computed, and then solved in a carry look-ahead architecture. In the end, with the final carry for each bit, the sum output is then generated. This architecture has the lowest latency in Nguyen et al. [102] and the results are very close to those from other architectures as reviewed by Preußer and Krause [109].

We show the final architecture with the 257-bit signed multiplier and 546-bit accumulator in Figure 4.8. The "optimized adder" can either add the multiplier output, the multiplication output multiplied by 2 (just a left shift by 1), or the compressor output. The output can then be fed back into the accumulator without change or right shift by the multiplier word size, which is needed for multi-word multiplication. The compressor is combined with the "optimized adder" to create an addition/subtraction operation with 3 operands, $out = acc + (s \cdot b) \pm a$ or $out = acc - (s \cdot b) + a$. The register "s" is used as a mask for register "b" and to create operations that may or may not add the value of register "b". Finally, the "optimized adder" uses the same AAM architecture discussed before in Figure 4.7.

While doing additions/subtraction, the pipeline operates with 4 stages because it does not need to wait for the multiplication stages to finish. This 4-stage mode is

Figure 4.8: The 8-stage pipeline multiplier accumulator `Carmela256`.



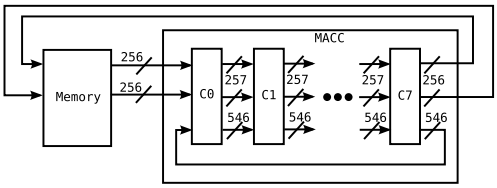

achieved by the multiplexer added onto the compressor output and the final addition. The register "s" is only used during addition/subtraction, and therefore only needs 4 register stages to remain synchronized with the pipeline unit.

**Carmela128.** The biggest difference between the two MACCs is the construction of the multiplier itself. `Carmela128` features an 8-stage pipeline architecture with 6 stages for the 129-bit signed multiplier and a 290-bit optimized adder. Similar to `Carmela256`, the multiplier uses 128 bits for both the signed and unsigned cases, while words consist of 16 bits. In contrast to the more complex hand-optimized 257-bit multiplier, the 129-bit signed multiplier with six pipeline stages was constructed by the FPGA synthesis tool. Although we expect this will need more DSPs than what we could achieve by hand, this architecture can easily be deployed in older FPGAs (which do not have rectangular DSPs) like the Xilinx Spartan 6 family. Because it has the same architecture as `Carmela256`, Figure 4.3 can also be used for `Carmela128`, only changing the sizes of the components.

### From a MACC to a finite field unit for up to 1008-bit primes

While this section focuses on `Carmela256` everything follows completely analogously for `Carmela128`. Except for multiplications with/without accumulations and additions/subtractions of 256-bit signed/unsigned values, the MACC cannot perform $\mathbb{F}_p$ operations directly. To construct the operations on top of `Carmela`, there are two options: create a state machine to control the hardware pipeline and provide $\mathbb{F}_p$ arithmetic at a higher level, or provide a pipeline in a higher level of `Carmela` and implement the operations in $\mathbb{F}_p$ in software. While both options seem very similar, they can lead to very different constructions. In the case of a software approach, it is necessary for the software to be synchronized with the pipeline or to have a memory that can load/store the 546-bit accumulator values. However, this high bandwidth

Figure 4.9: Pipeline with feedback to achieve $\mathbb{F}_p$ operations.




memory will require a lot of resources in the design so, as a result, we opted for the state machine.

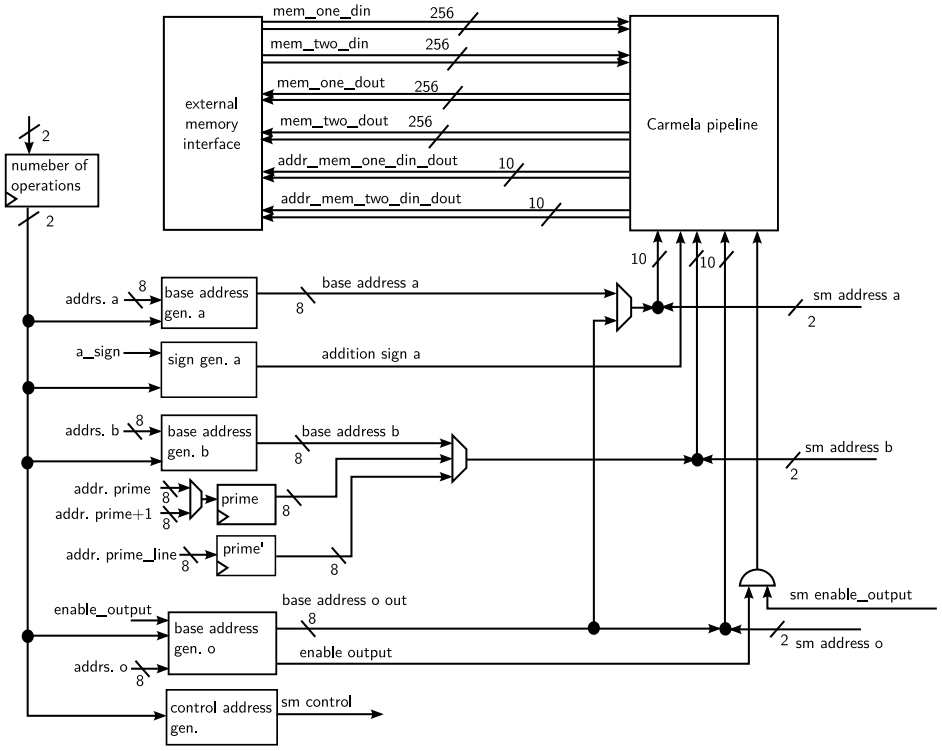
The state machine can use the internal pipeline accumulator registers to synchronize the instructions while offering the $\mathbb{F}_p$ arithmetic. However, in this case, the software cannot be optimized in terms of 257-bit multiplications, but only in terms of $\mathbb{F}_p$ operations. Moreover, to synchronize the operations with the pipeline, the state machine has to offer the same number of operations as the pipeline size, as seen in Figure 4.9. Therefore, 8 $\mathbb{F}_p$ multiplications have to be performed in parallel, since `Carmela` operates in a mode with an 8-stage pipeline. On the other hand, $\mathbb{F}_p$ addition/subtractions occur as 4 operations in parallel, because `Carmela` operates in 4 pipeline stages. It is, of course, possible in both cases to perform fewer operations, but the time it will take is the same if they were 8/4 operations since all parallel operations happen but only the ones needed to have the output are enabled.

To keep track of the operands and their addresses, the address resolution unit was added together with the state machine and the `Carmela` pipeline in Figure 4.10. The address resolution works by receiving an external base address that will be joined by the state machine address to find the final address. The MACC RAM consists of 1024 words of 256 bits (2048 words of 128 bits for `Carmela128`), but for `Carmela`, it is subdivided logically as 256 operands of 1024 bits. Thus, it is cheaper to resolve the final address by just appending two (three) bits into the least significant part of the base address. The base address generator also has to behave as an 8/4-stage pipeline, since the base address has to change together with their respective operands.

The state machine implements the operations addition/subtraction without reduction, multiplication without reduction, iterative modular reduction, Montgomery multiplication for any prime $p$ up to 1008 bits, Montgomery multiplication for so-called "Montgomery-friendly" primes $p$ for which $p' = -p^{-1} \bmod R$ is 1, and an optimized squaring variant for each of the multiplications. The addition/subtraction with no reduction instruction is straightforward. The iterative modular reduction is used whenever a public value is between $[-2p, 2p]$, in which case the procedure corrects to a value in the interval $[0, p-1]$. This procedure is only applied to correct data in the very last processing stage.

The multiplication without reduction applies the product scanning method. In this method, the multiplication is done by scanning the product output instead of the operands. This is usually more expensive in terms of loop control logic, but in our case, the loops are unrolled. In the case of the Montgomery multiplication for Montgomery-friendly primes, we apply the optimization technique presented in Costello et al. [37], which exploits the special form of SIDH/SIKE primes and saves several multiplications by computing the radix-$2^w$ Montgomery computation ($a +$

Figure 4.10: `Carmela256` with address resolution and memory interfaces.

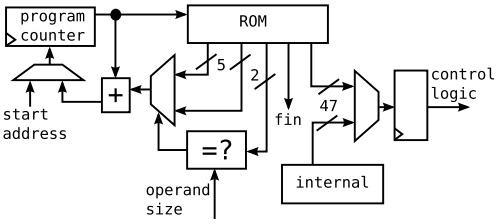



$(ap' \bmod 2^w) \cdot p)/2^w$ as $(a + (a \bmod 2^w) \cdot \hat{p})/2^w$, when $p' = -p^{-1} \bmod 2^w = 1$ for a given word-size $w$, where $a$ is the input and $\hat{p} = 2^{e_2}3^{e_3}$. In our implementations, we include the optimization as a separate procedure in the state machine and only apply it to the SIDH/SIKE primes for which $p' = -p^{-1} \bmod 2^w = 1$. Concretely, we apply it to all the cases with exception of `p434` and `p503` on `Carmela256` (i.e., when using $w = 256$). For both cases of the Montgomery multiplication, i.e., with and without exploiting the special prime form, we apply a product-scanning method called Finely Integrated Product Scanning (FIPS) by Koç et al. [72]. The FIPS method was chosen over the previous studied methods CIOS and FIOS, because it is more suitable for multiplier accumulator architectures that have the entire control unrolled.

Besides, the multiplication method that we apply is for signed numbers; not for unsigned numbers, as usually seen in the literature. Montgomery multiplication with signed numbers has been applied before to accelerate the multiplication process for Radix-4 multipliers [62, 132]. However, in our case by choosing signed numbers instead of unsigned numbers, it is possible to avoid adding multiples of the prime during the subtractions [124]. Therefore, just as it is possible to use additions with no reduction, the same happens with subtraction. In our case, it was opted to apply the Montgomery algorithm with more than 16 extra bits for all 4 primes, but more extra bits can be added if required.

The squaring operations are similar to the respective multiplication operations,

Figure 4.11: Optimized state machine to operate `Carmela256`.




except that they are optimized for the same operand input. This optimization makes squares faster than multiplications, yet this operation did not end up being used in the $\mathbb{F}_{p^2}$ operations. This has to do with how the squaring in $\mathbb{F}_{p^2}$ works, which will be explained in Section 4.2 in more detail.

Because the state machine has many states (more than 300) we opted to design the state machine as well, and not to use the FPGA synthesis tool. Figure 4.11 shows the state machine as a ROM which stores all the instructions and a simple control mechanism. The control mechanism is an adder that adds one to move on to the next instruction, or adds a bigger value in case a jump might be chosen. The jumps are usually when the algorithm takes a different path, which in our case only happens because of the operand size. The algorithms for operand sizes of 1, 2, 3, and 4 words share some parts, so instead of implementing the 4 algorithms separately, we opted to optimize the number of states by trying to share some instructions. The "fin" signal indicates when the last instruction has been executed, and the state machine is ready for the next $\mathbb{F}_p$ operation.

The 129-bit multiplier for `Carmela128` has the same state machine strategy, it just operates with operands of size up to 8, thus it has a lot more states than the 257-bit multiplier. Because our strategy compacts the states into a ROM, the increase in resources is negligible.

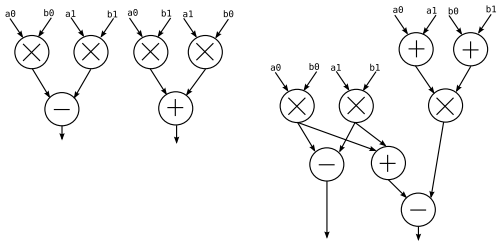
### Extension field operations

As $p \equiv 3 \bmod 4$, we have an isomorphism $\mathbb{F}_{p^2} \cong \mathbb{F}_p(i)$ where $i^2 = -1$. Fixing the basis $\{1, i\}$ of $\mathbb{F}_{p^2}$ as a 2-dimensional vector space over $\mathbb{F}_p$, we can represent any element $a = a_0 + a_1 \cdot i \in \mathbb{F}_{p^2}$ by the corresponding vector $(a_0, a_1)$. In that case, for elements $a = (a_0, a_1)$ and $b = (b_0, b_1)$ in $\mathbb{F}_{p^2}$, the field operations are given by

$$\begin{aligned} a + b &= (a_0 + b_0, a_1 + b_1)\,, \\ a - b &= (a_0 - b_0, a_1 - b_1)\,, \\ a \cdot b &= (a_0 \cdot b_0 - a_1 \cdot b_1, a_0 \cdot b_1 + a_1 \cdot b_0)\,, \\ a^2 &= ((a_0 + a_1) \cdot (a_0 - a_1), 2 \cdot a_0 \cdot a_1)\,, \\ a^{-1} &= (a_0 \cdot (a_0^2 + a_1^2)^{-1}, -a_1 \cdot (a_0^2 + a_1^2)^{-1})\,, \end{aligned}$$

where the operations on the right-hand side are in $\mathbb{F}_p$. We compute the operations in $\mathbb{F}_{p^2}$ directly using the equations above. That is, writing $\mathbf{m}$, $\mathbf{s}$, $\mathbf{a}$, and $\mathbf{i}$ for the cost of a multiplication, squaring, addition/subtraction, and inversion in $\mathbb{F}_p$, respectively,

Figure 4.12: $\mathbb{F}_{p^2}$-multiplication – schoolbook (left), Karatsuba (right).




and **M**, **S**, **A**, and **I** for the cost of a multiplication, squaring, addition/subtraction and inversion in $\mathbb{F}_{p^2}$, respectively, we have $\mathbf{M} = 4\mathbf{m} + 2\mathbf{a}$, $\mathbf{S} = 2\mathbf{m} + 3\mathbf{a}$, $\mathbf{A} = 2\mathbf{a}$ and $\mathbf{I} = 2\mathbf{m} + 2\mathbf{s} + 2\mathbf{a} + \mathbf{i}$. Inversions in $\mathbb{F}_p$ are computed via Fermat's Little Theorem using $t^{-1} = t^{p-2}$ for any (non-zero) $t \in \mathbb{F}_p$. Note that we use projective coordinates for all elliptic-curve operations, making the efficiency of the inversion operation of limited interest. We, therefore, favor Fermat inversion over more complex methods such as the constant-time gcd-based modular inversion of Bernstein and Yang [18].

Typically, since multiplications are the most expensive operation in $\mathbb{F}_p$, implementations perform multiplication in $\mathbb{F}_{p^2}$ at $3\mathbf{m}+5\mathbf{a}$ by using the Karatsuba method [68]. However, this trade-off comes at the expense of more dependent operations (see Figure 4.12). As depicted in Figure 4.12, the schoolbook method can be performed with a latency of $1\mathbf{m} + 1\mathbf{a}$, while the Karatsuba method has a latency of $1\mathbf{m} + 3\mathbf{a}$. Although the extra additions/subtractions do not contribute significantly to the total time of the multiplication, the overhead is not negligible either. More importantly, the multiplier unit `Carmela` is designed to perform 8 $\mathbb{F}_p$ multiplications in parallel, fitting exactly 2 $\mathbb{F}_{p^2}$ operations if the schoolbook method is adopted.

The square operation in $\mathbb{F}_{p^2}$ is also not performed with the Karatsuba optimization, nor schoolbook, but simply as a $\mathbb{F}_{p^2}$ multiplication. This is because the $\mathbb{F}_{p^2}$ squaring needs to perform at least one regular multiplication (i.e., $a_0 \cdot a_1$). Because the multiplier always performs 8 $\mathbb{F}_p$ multiplications in parallel, the cost of squaring by using the square operation will be higher than using the regular multiplication directly. For this reason, we do not use the optimized $\mathbb{F}_{p^2}$ squaring in this case.

In terms of improvements, there are several possibilities and trade-offs. One possibility is to implement a pipeline aimed at the Karatsuba method. In this case, one would need 6 stages if aiming at two $\mathbb{F}_{p^2}$ operations in parallel, or even 3 stages for one operation at a time. It is also possible to reduce the pipeline to 4 stages and limit it to one operation at a time using the schoolbook algorithm. Note that reducing the number of stages would increase the time each cycle takes, with the control instructions having a bigger contribution to the total time. Another potential modification in our architecture is not to have the multiplier output the multiplication with one value, but instead the carry-save notation, and then having only one large adder that is used in conjunction with the accumulator. This optimization could reduce the amount of resources, since the adder at the end of the compression tree in the multiplier would not be required. Other ideas involve the design of a basic 64-bit MACC instead of relatively large 128- and 256-bit MACCs and the direct implementation of the $\mathbb{F}_{p^2}$ arithmetic in the state machine.

Figure 4.13: Main instruction template for the SIKE processor.

| main flag | append | main internal | operand o | | operand b | | | | operand a | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 63 62 | 61 | 60 ... 55 | 54 | 53 ... 38 | 37 | 36 | 35 | 34 ... 19 | 18 | 17 | 16 | 15 ... 0 |
| 00 | 0 | type | Dir | memo | Sign | Dir | Cx | memb | Sign | Dir | Cx | mema |

### Instruction set and memory arrangement

The template for the instructions of the main processor is depicted in Figure 4.13. The main processor instructions should have the two most significant bits (63 and 62) be zero, otherwise, it is a coprocessor instruction. The 61st bit is reserved and should be 0, while the next 6 bits (60 to 55) are used to infer the instruction type. The input operands are identified between bits 0 to 18 and 19 to 37 for "a" and "b", respectively. The output operand "o" is determined by bits 38 to 54. For each input operand, there is the "Cx" flag on bits 16 and 35 respectively, which distinguishes the value "mem(a,b)" as a constant or a memory address. The bit "Dir" indicates when the operand value is a memory address that contains the value applied in the operation or if the operand value is a memory address that points to the memory address that has the real value. Finally, the bit "Sign" shows if the operand values are signed or unsigned. Not all bits are active in all instructions, since the SIKE protocol does not need all possible combinations. By not implementing all possible options, we reduce the number of resources in the decoding unit.

Some of the main instructions have different variations according to which operand is going to be used. For example, the push instruction has variants "pushf" and "pushm". The appended "f" and "m" indicate instructions for 256-bit data (or 128-bit data for `Carmela128`) and operands up to 1024 bits, respectively. The regular push instruction will take a 16-bit value and push it into memory. The "f" variant is meant to be applied to the MACC RAM values, which can be addressed in 1024 words of 256 bits (or 2048 words of 128 bits for `Carmela128`). However, most of the time we are not working on the bare 256 or 128-bit words, but on the $\mathbb{F}_p$ operands which can be any 256-bit or 128-bit multiple (up to 1024 bits). For those cases, the preferred instruction is the one that ends with an "m", as it will work directly with operand values. The full set of instructions of the main unit are in Figure 4.17.

The instruction template for the coprocessor `Carmela` is shown in Figure 4.14. The `Carmela` coprocessor instruction is identified as "01" in bits 63 to 62. Because `Carmela` has fewer instructions, only 4 bits are used to identify the type with 2 inputs and 1 output. Just as in the main instruction template, the bit "Dir" is used to distinguish values that hold memory addresses or values that hold a memory address that points to the effective memory address. The "Dir" bit is also used to show if the output is disabled or not by setting it as 1 and "mo" as 0. The bit "Sign" in this case is not used to identify an operation as signed or unsigned, but to distinguish addition and subtraction during the addition/subtraction instruction (mo = mb $\pm$ ma). Since addition/subtraction instructions only need 4 operations in parallel, the instruction in this case only uses the first 4 operands (0 to 3). The instruction operand not used for addition/subtraction could be optimized to not have the "Sign" bit, but it was added to the instruction for consistency. The "Dir" bit can also work with the "RD

Figure 4.14: Instruction template for the coprocessor `Carmela`.

| carmela flag | append | carmela internal | operand o | | operand b | | | | operand a | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 63 62 | 61 ... 59 | 58 ... 55 | 54 | 53 ... 38 | 37 | 36 | 35 | 34 ... 19 | 18 | 17 | 16 | 15 ... 0 |
| 01 | 000 | type | Dir | Mo | 0 | Dir | En | Mb | Sign | Dir | En | Ma |

register 0" least significant bit (LSB) or most significant bit (MSB). In this mode, the "RD register 0" bit replaces the LSB in the instruction memory value. This makes it possible to change which memory pointer will be used depending on the "RD register 0" bit.

To be able to perform all 8/4 operations in parallel, `Carmela` instructions must be loaded serially. Therefore, all 8/4 operations are loaded one after another. In case one wants to perform fewer operations, it is still necessary to load all 8/4 operations, but the ones not used should have the output disabled.

The internal small bus can access all SIKE components through a 16-bit address region, which is detailed in Figure 4.15. The first positions belong to the MACC RAM, which includes some reserved space in case it is necessary to increase its size. The program part holds the SIKE processor instructions. The ALU RAM is the area used for control and to perform the isogeny tree algorithm of SIDH. The Keccak core memory positions are to send/receive data and commands to the core. The status register only shows if the SIKE processor is free or not. Operands size is for the size of the operands primes (0 = up to 256, 1 = up to 512, 2 = up to 768, 3 = up to 1024, and in the case of `Carmela128` the values go from 0 to 7 up to 1024). The flag is another register that is used to help debug the processor, which can be also seen as the external communication output of the processor.

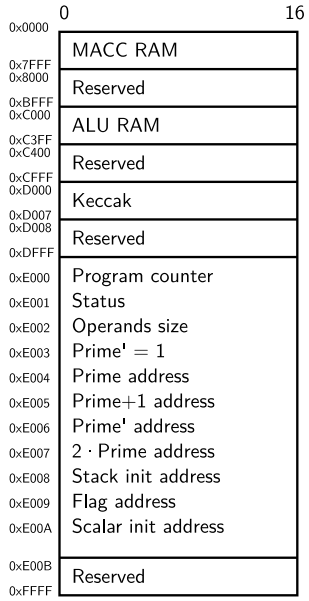
The program counter register has two purposes. The first is to keep track of the current instruction address, and the second purpose is to start any procedure inside the SIKE processor. For example, if the desired function to run is in position 3 of the program memory, we just write in the program counter register the value 3 and it will start the execution. The execution continues until the last instruction called "fin" is executed, then the processor stops and waits until a new value is written on the program counter. The full set of instructions of the coprocessor are in Figure 4.18.

### Software elliptic curve and isogeny computations

As discussed in section 4.2, elliptic curve and isogeny computations, including the three-point differential ladder and large-degree isogeny computation, are done in software; see Figure 4.2. Our software implementation is very similar to the C implementation of SIKE in the SIDH library [38] since we use the same structure of function calls to implement key generation, encapsulation, and decapsulation. The only difference is that we optimize the isogeny computation, three-point differential ladder, and field inversion to perform two $\mathbb{F}_{p^2}$ operations in parallel whenever possible. Only in very few instances do we have to perform one $\mathbb{F}_{p^2}$ operation at a time.

Because the entire underlying arithmetic described in Section 4.2 executes in a constant amount of clock cycles, and the software uses constant-time routines exactly like in the C implementation of the SIDH library, our implementation is also constant-

Figure 4.15: Memory arrangement of the SIKE processor.




time. The only source of timing variability is the parameter size since the arithmetic unit operates in multiples of 256 for `Carmela256` and 128 for `Carmela128`. However, these sizes are public and do not leak sensitive information. For the three-point differential ladder (the only function that directly handles the secret key), we always assume the maximum number of bits for the expected secret. Some timing variable components like caches are not present, and memory accesses take the same amount of time regardless of the position. Finally, the main CPU itself is also constant time, since all instructions take the same amount of time and every instruction waits for the previous one to finish before starting.

Although the design is constant-time, it is not safe against differential side-channel attacks like DPA. In those cases, it is necessary to apply countermeasures such as scalar and projective coordinate randomization to the ladder steps. These countermeasures can be implemented in software together with a random number generator in the architecture. It is worthy mentioning that some previous works already implemented such countermeasures [75].

## 4.3 Implementation results and analysis

The entire hardware architecture was designed and tested in VHDL. The testing procedure applied is depicted in Figure 4.16. The SIKE protocol was first implemented in Python with the Sage environment [117] for testing purposes given the easy availability of mathematical operations. Then, a second Sage code that mimics the procedures used by the hardware, such as the Montgomery multiplication, was done and then tested against the higher-level Sage code. Finally, the same tests applied to test the second Sage code were used to test the VHDL code. When the VHDL was simu-

Figure 4.16: Testing and coding for VHDL.

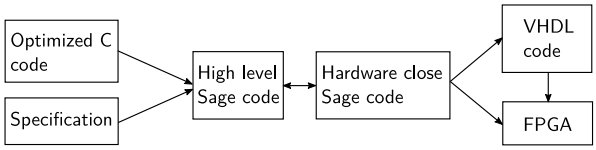



Table 4.1: Resources required for the `Carmela128` and `Carmela256`-based architectures on several platforms. The max period is measured in nanoseconds (ns).

| Mul. | FPGA | Slices | LUTs | FFs | BRAMs | DSPs | Max period |
|---|---|---|---|---|---|---|---|
| 128 | Virtex 7 690T | 3 855 | 11 984 | 7 268 | 21 | 57 | 6.496 |
| | Artix 7 100T | 3 491 | 11 943 | 7 202 | 21 | 57 | 6.892 |
| 256 | Virtex 7 690T | 8 131 | 21 321 | 13 756 | 39 | 162 | 7.061 |
| | Artix 7 100T | 7 329 | 22 673 | 11 661 | 37 | 162 | 9.170 |

lated, the code was synthesized in the Zynq platform and then the same tests used for the VHDL were applied. All tests were done in the whole KEM and in parts, thus testing each function separately. This approach eased the debugging process of the architecture.

The post-place and route results that we obtained for our architectures with Xilinx ISE 14.7 (Virtex 7 and Artix 7) and Vivado 2019.1 (Zynq 7020) are displayed in Table 4.1. For our tests on the Zynq platform, we used a Zedboard development kit [7]. For Artix 7, which is the platform recommended by NIST in the ongoing PQC standardization process [98], we used an Arty A7 development kit [46] that contains the Artix 7 FPGA model XC7A100TCSG324-3. For Virtex 7, we targeted the FPGA model XC7V690T, which was chosen to ease comparisons with previous work by Koziel et al. [77, 76, 75] and Roy and Mukhopadhyay [116].

Table 4.1 nicely illustrates the effect of using each of our two `Carmela` options. As expected, the 128-bit architecture consumes significantly fewer slices and DSPs, reflecting the fact that the `Carmela` unit needs less logic, multipliers, and registers. It also requires fewer BRAMs, due to the reduced bandwidth: to retrieve one 128-bit word, it needs 4 BRAMs of 32 bits, while for 256 bits it is twice that number. In the case of the operating frequency, the `Carmela128`-based architecture only achieves a slightly higher frequency, given that the entire control mechanism remains expensive even though the `Carmela` unit is smaller.

**Timing results for $\mathbb{F}_p$ operations.** At first, there are the intermediate cycle counts for the $\mathbb{F}_p$ operations performed by `Carmela128` and `Carmela256` (i.e., multiplication, squaring, and addition/subtraction, see Table 4.2). Since the architecture is aimed at always performing 2 $\mathbb{F}_{p^2}$ operations in parallel, the multiplication and squaring operations execute 8 $\mathbb{F}_p$ multiplications and squarings in parallel, respectively, while addition/subtraction and iterative reduction perform 4 operations in parallel. More details can be found in Section 4.2.

Table 4.2 has three different results for `Carmela128` for multiplication and squaring in the case of $p' = 1$. Each result is the number of words of all-zero words in the parameter $p + 1$, in this case, it supports 1, 2, and 3 zeroes. In the SIKE implementation, `SIKEp434` and `SIKEp503` have 1 word and `SIKEp610` and `SIKEp751` have 2 words.

As `Carmela128` and `Carmela256` support arbitrary primes (i.e., not only SIKE primes), we can compare the results of the multiplication routine with existing ECC literature. For example, Järvinen et al. [64] construct a 127-bit solution consisting of a 64-bit pipelined multiplier accumulator that has 7 stages, for support of the Four$\mathbb{Q}$ [36] curve defined over a quadratic extension of $\mathbb{F}_{2^{127}-1}$. As such, it supports up to 7 multiplications in parallel and achieves a latency of 20 cycles per multiplication with 16 DSPs. For comparison, `Carmela128` requires 42 cycles for 8 multiplications (5.25 cycles per multiplication) with 57 DSPs. We achieve a similar speed/area, where we lean more towards resources to be able to support larger primes at low latency as well. The 256-bit version `Carmela256` also takes 42 cycles for 8 multiplications with 162 DSPs, though it is too large for the intended prime size.

Similarly, Sasdrich and Güneysu [120] aim for a specific 448-bit prime to support Curve448 [78]. They achieve 36 cycles per multiplication with 33 DSPs. In our case, `Carmela128` takes 298 cycles for 8 multiplications (37.25 cycles per multiplication) at 57 DSPs, while `Carmela256` requires 90 cycles for 8 multiplications (11.25 cycles per multiplication) at the cost of 162 DSPs. As was the case for the work of Järvinen et al. [64], we note that the modular reductions from Sasdrich and Güneysu significantly benefit from the simple prime structure, which we do not exploit to be able to support all primes up to 1022 bits.

Table 4.3 shows the number of cycles our architecture takes for all SIKE parameters. In some functions, such as `inversion` (in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$) and `Ladder 3 Points`, the growing factor between parameters is bigger than functions such as `xDBL` and `xTPL`. This difference is because the `inversion` in $\mathbb{F}_p$ functions grows not only in the number of words used by `Carmela` but also by the size of the prime which affects the internal loop size.

**Comparison with other SIDH/SIKE implementations.** We compare our SIKE results with other results in the literature in Table 4.4. We focus on the Virtex 7 690T platform because, to our knowledge, it has been the only target of previous SIDH and SIKE implementations. As advertised, existing works [77, 76, 8, 116, 75] achieve high-speed at the expense of a much higher amount of resources. For example, in comparison with our 256-bit architecture, the most recent SIKE implementation by [75] is 73% and 56% faster for `SIKEp503` and `SIKEp751`, respectively. However, for the largest parameter set (i.e., `SIKEp751`) our architecture requires up to 54% fewer slices, 10% fewer BRAMs, and 68% fewer DSPs. We remark that DSPs are very expensive hardware resources. Moreover, our architecture supports all four different SIKE parameters set (and any other primes if necessary), while that of [75] only supports `SIKEp751` unless the device is reprogrammed. For our 128-bit architecture, the implementation of [75] is 85% faster for both `SIKEp503` and `SIKEp751`, but we need 78% fewer slices, 52% fewer BRAMs, and 89% fewer DSPs to support `SIKEp751` (and other parameter sets).

Table 4.2: Cycle counts of $\mathbb{F}_p$ operations for the `Carmela128` and `Carmela256`-based architectures. The top column shows the maximum bit size of the operands. The operations that use the multiplier: multiplication (Mult.) and squaring (Square) with/without modular reduction (red.) perform 8 operations in parallel. The cycles are measured after all 8 outputs have been written to memory. The addition/subtraction (Add/Sub) and iterative reduction only operate on 4 inputs.

| Mul. | Operation | 126 | 254 | 382 | 510 | 638 | 766 | 894 | 1022 |
|---|---|---|---|---|---|---|---|---|---|
| 128 | Mult. no red. | 18 | 42 | 82 | 138 | 210 | 298 | 402 | 522 |
| | Mult. with red. | 42 | 90 | 178 | 298 | 450 | 634 | 850 | 1098 |
| | Mult. with red. $p' = 1$ (1) | – | 58 | 130 | 234 | 370 | 538 | 738 | 970 |
| | Mult. with red. $p' = 1$ (2) | – | – | 106 | 202 | 330 | 490 | 682 | 906 |
| | Mult. with red. $p' = 1$ (3) | – | – | – | 170 | 290 | 442 | 626 | 842 |
| | Square no red. | 18 | 34 | 58 | 90 | 130 | 178 | 234 | 298 |
| | Square with red. | 42 | 82 | 154 | 250 | 370 | 514 | 682 | 874 |
| | Square with red. $p' = 1$ (1) | – | 50 | 106 | 186 | 290 | 418 | 570 | 746 |
| | Square with red. $p' = 1$ (2) | – | – | 82 | 154 | 250 | 370 | 514 | 682 |
| | Square with red. $p' = 1$ (3) | – | – | – | 122 | 210 | 322 | 458 | 618 |
| | Add/Sub no red. | 10 | 14 | 18 | 22 | 26 | 30 | 34 | 38 |
| | Add/Sub with red. | 22 | 38 | 54 | 70 | 86 | 102 | 118 | 134 |
| | Iterative red. | 22 | 34 | 46 | 58 | 70 | 82 | 94 | 106 |
| 256 | Mult. no red. | 18 | 18 | 42 | 42 | 82 | 82 | 138 | 138 |
| | Mult. with red. | 42 | 42 | 90 | 90 | 178 | 178 | 298 | 298 |
| | Mult. with red. $p' = 1$ | – | – | 58 | 58 | 130 | 130 | 234 | 234 |
| | Square no red. | 18 | 18 | 34 | 34 | 58 | 58 | 90 | 90 |
| | Square with red. | 42 | 42 | 82 | 82 | 154 | 154 | 250 | 250 |
| | Square with red. $p' = 1$ | – | – | 50 | 50 | 106 | 106 | 186 | 186 |
| | Add/Sub no red. | 10 | 10 | 14 | 14 | 18 | 18 | 22 | 22 |
| | Add/Sub with red. | 22 | 22 | 38 | 38 | 54 | 54 | 70 | 70 |
| | Iterative red. | 22 | 22 | 34 | 34 | 46 | 46 | 58 | 58 |

Since we present a speed/area trade-off, we also include a proper metric to compare against existing literature. As previous work used the product of slices and time (ST) as a measure, we include this as well. In this metric, `Carmela256` has a fairly similar performance to `Carmela128`, but scales much better to larger primes, as can be seen from the results for `SIKEp751`. In comparison to the best recent work on SIKE [75], this one is worse for `SIKEp503` and `SIKEp751`. However, this metric excludes the expensive DSP resources. For that purpose, the product of DSPs and time (DT) is included as a metric. In that case, `Carmela128` performs better than `Carmela256` for the smaller SIKE primes, while `Carmela256` is superior for `SIKEp751`. Another observation is that `Carmela128` achieves better results for `SIKEp751` than the work of Koziel et al. [75], while `Carmela256` outperforms it. The work of Roy and Mukhopadhyay [116] has even better results than `Carmela256` for `SIKEp751`, though it is worse in the ST metric. We emphasize that neither of the metrics provides a complete picture; the relevant trade-off depends on the context in which the architecture is used and the relative cost of the different components.

A similar trade-off can be observed against related work only implementing the

Table 4.3: Cycle counts for each operation and respective SIKE prime.

| Mul. | Operation | SIKEp434 | SIKEp503 | SIKEp610 | SIKEp751 |
|---|---|---|---|---|---|
| | inversion (in $\mathbb{F}_p$) | 111646 | 197278 | 215235 | 379663 |
| | inversion (in $\mathbb{F}_{p^2}$) | 112241 | 198121 | 215998 | 380714 |
| | j_invariant | 113476 | 199940 | 217657 | 383053 |
| | Get_A | 113220 | 199536 | 217293 | 382517 |
| | Ladder 3 Points (oa bits) | 364678 | 636152 | 702747 | 1229162 |
| | Ladder 3 Points (ob bits-1) | 366363 | 641234 | 700426 | 1248968 |
| 128 | xDBL | 870 | 1302 | 1182 | 1686 |
| | Get_4_Isogenies | 605 | 897 | 817 | 1157 |
| | Eval_4_Isogenies | 1125 | 1697 | 1537 | 2205 |
| | xTPL | 1698 | 2558 | 2318 | 3322 |
| | Get_3_Isogenies | 889 | 1325 | 1205 | 1713 |
| | Eval_3_Isogenies | 841 | 1269 | 1149 | 1649 |
| | Get_2_Isogenies | 297 | 441 | 401 | 569 |
| | Eval_2_Isogenies | 605 | 897 | 817 | 1157 |
| | Key generation | 2192928 | 3894735 | 4351847 | 7963047 |
| 128 | Encapsulation | 3711255 | 6556297 | 8098600 | 13152312 |
| | Decapsulation | 3948665 | 6993275 | 8208911 | 14173546 |
| | inversion (in $\mathbb{F}_p$) | 47098 | 99114 | 90575 | 108903 |
| | inversion (in $\mathbb{F}_{p^2}$) | 47385 | 99569 | 90934 | 109262 |
| | j_invariant | 47948 | 100524 | 91697 | 110025 |
| | Get_A | 47840 | 100316 | 91537 | 109865 |
| | Ladder 3 Points (oa bits) | 160566 | 327896 | 312083 | 380570 |
| | Ladder 3 Points (ob bits-1) | 161307 | 330514 | 311042 | 386696 |
| 256 | xDBL | 388 | 676 | 532 | 532 |
| | Get_4_Isogenies | 277 | 473 | 377 | 377 |
| | Eval_4_Isogenies | 493 | 873 | 681 | 681 |
| | xTPL | 744 | 1316 | 1028 | 1028 |
| | Get_3_Isogenies | 401 | 693 | 549 | 549 |
| | Eval_3_Isogenies | 369 | 653 | 509 | 509 |
| | Get_2_Isogenies | 137 | 233 | 185 | 185 |
| | Eval_2_Isogenies | 277 | 473 | 377 | 377 |
| | Key generation | 987560 | 2027615 | 1961543 | 2511535 |
| 256 | Encapsulation | 1672151 | 3416721 | 3654472 | 4158492 |
| | Decapsulation | 1782657 | 3646083 | 3706471 | 4480970 |

Table 4.4: Comparison of SIKE implementations on the Xilinx Virtex 7 690T. The frequency ("Freq.") is measured in Megahertz (MHz) and the total time (`Encaps` + `Decaps` for SIKE, a full ephemeral key exchange for SIDH) in milliseconds (ms). The ST resp. DT column displays the number of slices and DSPs respectively times the time divided by 1000, rounded to the nearest multiple of 1000 resp. 100. This architecture works directly for any parameter set `pXXX` for `XXX` $\in \{434, 503, 610, 751\}$ and does not need to be reprogrammed. Works emphasized with a † symbol only implement SIDH, while the others do SIKE.

| Reference | Par. | Slices | DSP | BRAM | Freq. | Time | ST | DT |
|---|---|---|---|---|---|---|---|---|
| [77]† | p503 | 8 918 | 192 | 40.0 | 181.4 | 20.9 | 186 | 4.0 |
| | p751 | 11 801 | 282 | 47.0 | 177.3 | 46.3 | 546 | 13.1 |
| [76]† | p503 | 7 491 | 192 | 43.5 | 202.1 | 16.5 | 124 | 3.2 |
| | p751 | 11 277 | 288 | 60.5 | 204.9 | 36.4 | 410 | 10.5 |
| [116]† | p751 | 18 711 | 294 | 22.5 | 225.7 | 31.6 | 591 | 9.3 |
| [8] | p751 | 16 756 | 376 | 56.5 | 198.0 | 33.4 | 560 | 12.6 |
| [75] | p503 | 9 514 | 264 | 34.0 | 171.2 | 13.6 | 129 | 3.6 |
| | p751 | 17 530 | 512 | 43.5 | 167.4 | 26.9 | 472 | 13.8 |
| Ours (128) | pXXX | 3 855 | 57 | 21.0 | 153.9 | 49.8 | 192 | 2.8 (p434) |
| | | | | | | 88.0 | 339 | 5.0 (p503) |
| | | | | | | 105.9 | 408 | 6.0 (p610) |
| | | | | | | 177.5 | 684 | 10.1 (p751) |
| Ours (256) | pXXX | 8 131 | 162 | 39.0 | 141.6 | 24.4 | 198 | 4.0 (p434) |
| | | | | | | 49.9 | 405 | 8.1 (p503) |
| | | | | | | 52.0 | 423 | 8.4 (p610) |
| | | | | | | 61.0 | 496 | 9.9 (p751) |

SIDH protocol, such as [76] and [116]. In this case, however, one needs to take into account some additional costs to support SIKE, including the use of the Keccak core to implement SHAKE-256.

Koziel et al. [75] include two countermeasures against differential power analysis, namely, scalar randomization and projective coordinate randomization. The cost of scalar randomization is one random number of 64 or more bits and one extra scalar multiplication, while projective coordinate randomization costs one $\mathbb{F}_{p^2}$ random number and 3 multiplications per element. These techniques, which are applied during key generation and decapsulation, exhibit a very small overhead and are easy to implement in the proposed architectures using our hardware/software co-design. As stated before, unlike previous results, this architecture can work on all four currently available primes (and any other future prime as long as it is smaller than 1008 bits) without having to be re-synthesized. For any new prime, only the relevant constants and public parameters need to be uploaded.

Table 4.5: Comparison of PQC implementations on the Xilinx Artix 7 100T. The frequency ("Freq.") is measured in Megahertz (MHz), the total time (`Encaps` + `Decaps` for BIKE, FrodoKEM and SIKE, a full ephemeral key exchange for NewHope) in milliseconds (ms). This architecture works directly for any parameter set `pXXX` for `XXX` $\in \{434, 503, 610, 751\}$ and does not need to be regenerated.

| Reference | Algorithm | Slices | DSPs | BRAMs | Freq. | Time |
|---|---|---|---|---|---|---|
| [103] | NewHope-Simple | 1 708 | 2 | 4 | 125.0 | 2.9 |
| | (Server/Client) | 1 483 | 2 | 4 | 117.0 | |
| [6] | BIKE | 1 559 | - | 13 | 161.3 | 10.2 |
| [63] | FrodoKEM-640 | 1 855 | 1 | 11 | 167.0 | 40.0 |
| | (Encaps/Decaps) | 1 992 | 1 | 16 | 162.0 | |
| Ours (128) | SIKE | 3 491 | 57 | 21 | 145.1 | 52.8 (`p434`)<br>93.4 (`p503`)<br>112.4 (`p610`)<br>188.3 (`p751`) |
| Ours (256) | SIKE | 7 329 | 162 | 37 | 109.1 | 31.7 (`p434`)<br>64.8 (`p503`)<br>67.5 (`p610`)<br>79.2 (`p751`) |

**Comparison with other PQC implementations.** I compare this SIKE results with other representative post-quantum key-exchange algorithms in the literature listed in Table 4.5. I focus on the Artix 7 100T platform, which is widely used and has been recommended by NIST in the ongoing PQC standardization process.

As can be seen, SIKE demands more resources and is computationally more expensive than competing alternatives. However, it should be noted that the proposed architectures advance the state-of-the-art and reduce that gap significantly, arguably demonstrating that SIKE, which features the smallest public keys in the NIST PQC process, can be efficiently implemented for embedded applications.

**Results for NIST curves scalar multiplication.** Classic elliptic curve support is necessary in case the system is deployed before elliptic curves are phased out, and also to give support for a hybrid key exchange solution. As a proof-of-concept that this hardware architecture can work with classic elliptic curves and SIKE, elliptic curve scalar multiplication for all NIST FIPS 186-4 primes [100] was implemented. Our implementation was made with the Montgomery power ladder and the complete formulas by Renes et al. [111]. Table 4.6 has the results for our two architectures `Carmela256` and `Carmela128` and Amiet et al [5].

Table 4.6: Comparison of ECC implementations on the Xilinx Virtex 7 690T. The frequency ("Freq.") is measured in Megahertz (MHz), the total time (scalar multiplication for a random point and random scalar) in milliseconds (ms). This architecture and Amiet et al. [5] works directly for any NIST parameter set and does not need to be regenerated.

| Reference | Curve | Slices | DSPs | BRAMs | Freq. | Time |
|---|---|---|---|---|---|---|
| [5] | NIST | 1 704 | 20 | - | 225 | 1.5 (p256)<br>4.1 (p384)<br>9.7 (p521) |
| [5] | NIST | 2 068 | 64 | - | 161 | 2.1 (p384)<br>5.0 (p521) |
| Ours (128) | NIST | 3 855 | 57 | 21 | 153.9 | 1.3 (p224)<br>2.6 (p256)<br>6.2 (p384)<br>8.5 (p521) |
| Ours (256) | NIST | 8 131 | 162 | 39 | 141.6 | 0.8 (p224)<br>1.6 (p256)<br>2.5 (p384)<br>4.6 (p521) |

## 4.4 Final considerations of the chapter

I presented the first hardware/software co-design for the SIKE scheme, and the first hardware implementation of the new round 2 parameters `SIKEp434` and `SIKEp610` (in addition to `SIKEp503` and `SIKEp751`). In contrast to earlier works, this was focused on compactness and scalability instead of raw speed. I believe this work presents interesting new trade-offs; the 128-bit architecture is roughly 80% smaller and 80% slower, while the 256-bit design is about 55% smaller and 55% slower compared to existing implementations. Besides being more compact, the co-design approach leads to attractive scalability properties. For any prime $p$ of at most 1008 bits, this supports $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic, and any SIDH and SIKE implementation is built on top of it. Therefore, instead of regenerating the circuit to change parameter sets, we can simply upload the respective SIKE constants and public parameters. In particular, this makes it very easy to update the parameters sets defined in round 2 of SIKE's NIST submission if they are modified for round 3.

Moreover, the proposed coprocessor can also be used for performing the prime field arithmetic as used in classical elliptic-curve cryptography. For example, adding a single additional routine in software can very cheaply lead to the support of discrete-log-based protocols via the Montgomery curve "BigMont" proposed in [37] or any of the classical prime-field curves standardized by NIST [100]. Combining this with SIKE gives rise to a hybrid protocol that may be of serious interest while post-quantum protocols are still being analyzed and standardized. Though out of scope for this work, we consider this an interesting research direction.

Another interesting line of research would be to include more extensive side-channel countermeasures. Although our implementation is constant-time and, as such, protected against simple side-channel and timing attacks, more sophisticated countermeasures would include those that protect against more advanced attacks. These kinds of protections are considered out of scope for this work, but the hardware/software co-design makes some obvious candidates (e.g., projective coordinate and scalar randomization) straightforward to adopt.

Finally, the instruction set was designed to provide the operations needed for the SIDH/SIKE functions, but should also work with other cryptosystems, such as C-SIDH [28] or B-SIDH [34] procedures. Future work could involve the change of the instruction set to a well-known and widely adopted ISA, such as RISC-V.

As a side note, a few selected SIKE implementation have been done after or at a similar time as this research was publicly available which is worthy of mentioning [79, 81, 49, 115]. Liu et al. [79] implemented an optimized multiplier for SIDH and SIKE cryptosystems, through a different radix representation and optimized Barret reduction method. Liu et al. [79] results show a very small and fast multiplier implementation when compared to other literature works, but as described the implementation has to be synthesized for each parameter. Another implementation that has to synthesized for different parameters is by Farzam et al. [49], where the authors improve on the multiplier of Koziel et al. [75] and achieve faster and better area-time product. With a similar goal of improving the area-time product metric, Longa et al. [81] created a RISC-V hardware and software co-design architecture for SIKE. The authors of Longa et al. [81] improved on the area-time product in order to understand more the costs of breaking SIKE with specialized hardware, not only to showcase a better implementation. Roy et al. [115] also used a RISC-V to create a SIKE solution, but instead of creating a separate co-processor as Longa et al. [81], they added new instructions to accelerate SIKE.

## Main processor and Carmela instructions

Figure 4.17: The main instructions for the SIKE processor.

| | main flag | main append | main internal | | | operand o | | operand b | | | | operand a | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| name | 63 62 00 | 61 0 | 60 ... 55 type | | | 54 Dir | 53 ... 38 memo | 37 Sign | 36 Dir | 35 Cx | 34 ... 19 memb | 18 Sign | 17 Dir | 16 Cx | 15 ... 0 mema |
| nop | 00 | 0 | 000000 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| jump | 00 | 0 | 000001 | | | 0 | memo | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| jumpeq | 00 | 0 | 000001 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| jumpl | 00 | 0 | 000010 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| jumpls | 00 | 0 | 000010 | | | 0 | memo | 1 | 0 | Cx | memb | 1 | 0 | Cx | mema |
| jumpeql | 00 | 0 | 000011 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| jumpeqls | 00 | 0 | 000011 | | | 0 | memo | 1 | 0 | Cx | memb | 1 | 0 | Cx | mema |
| push | 00 | 0 | 000100 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Dir | Cx | mema |
| pop | 00 | 0 | 000101 | | | Dir | memo | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pushf | 00 | 0 | 000110 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Dir | 1 | mema |
| popf | 00 | 0 | 000111 | | | Dir | memo | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pushm | 00 | 0 | 001000 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Dir | 1 | mema |
| popm | 00 | 0 | 001001 | | | Dir | memo | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| copy | 00 | 0 | 001010 | | | Dir | memo | 0 | 0 | 0 | 0 | 0 | Dir | Cx | mema |
| copyf | 00 | 0 | 001011 | | | Dir | memo | 0 | 0 | 0 | 0 | 0 | Dir | 1 | mema |
| copym | 00 | 0 | 001100 | | | Dir | memo | 0 | 0 | 0 | 0 | 0 | Dir | 1 | mema |
| lconstf | 00 | 0 | 001101 | | | Dir | memo | 0 | 0 | 0 | 0 | Sign | 0 | 0 | value |
| lconstm | 00 | 0 | 001110 | | | Dir | memo | 0 | 0 | 0 | 0 | Sign | 0 | 0 | value |
| call | 00 | 0 | 001111 | | | 0 | memo | 0 | 0 | 0 | 0 | 0 | 0 | 1 | pc |
| ret | 00 | 0 | 010000 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| keccak_init | 00 | 0 | 010010 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| keccak_go | 00 | 0 | 010011 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| copya (inc) | 00 | 0 | 0110 | Cd | Cs | Dir | memo | 0 | 0 | 0 | size | 0 | Dir | 1 | mema |
| copya (dec) | 00 | 0 | 0111 | Cd | Cs | Dir | memo | 0 | 0 | 0 | size | 0 | Dir | 1 | mema |
| badd | 00 | 0 | 100000 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| bsub | 00 | 0 | 100001 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| bsmul | 00 | 0 | 100010 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| bsmuls | 00 | 0 | 100010 | | | 0 | memo | 1 | 0 | Cx | memb | 1 | 0 | Cx | mema |
| bshiftr | 00 | 0 | 100100 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| brotr | 00 | 0 | 100101 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| bshiftl | 00 | 0 | 100110 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| brotl | 00 | 0 | 100111 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| bland | 00 | 0 | 101000 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| blor | 00 | 0 | 101001 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| blxor | 00 | 0 | 101010 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | Cx | mema |
| blnot | 00 | 0 | 101010 | | | 0 | memo | 0 | 0 | Cx | memb | 0 | 0 | 0 | 1 ... 1 |
| fin | 00 | 0 | 111111 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 4.18: The `Carmela` coprocessor instructions for the SIKE processor.

| | carmela flag | carmela append | carmela internal | output o | | input b | | | | input a | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| name | 63 62<br>01 | 61 ... 59<br>000 | 58 ... 55<br>type | 54<br>Dir | 53 ... 38<br>Mo | 37<br>0 | 36<br>Dir | 35<br>En | 34 ... 19<br>Mb | 18<br>Sign | 17<br>Dir | 16<br>En | 15 ... 0<br>Ma |
| mmuld | 01 | 000 | 0000 | Dir | Mo | 0 | Dir | En | Mb | 0 | Dir | En | Ma |
| msqud | 01 | 000 | 0001 | Dir | Mo | 0 | Dir | En | Ma | 0 | Dir | En | Ma |
| mmulm | 01 | 000 | 0010 | Dir | Mo | 0 | Dir | En | Mb | 0 | Dir | En | Ma |
| msqum | 01 | 000 | 0011 | Dir | Mo | 0 | Dir | En | Ma | 0 | Dir | En | Ma |
| madd_subd | 01 | 000 | 0100 | Dir | Mo | 0 | Dir | En | Mb | Sign | Dir | En | Ma |
| mitred | 01 | 000 | 0101 | Dir | Mo | 0 | 0 | 0 | 0 | 0 | Dir | En | Ma |
| madd_subr | 01 | 000 | 0100 | Dir | Mo | 0 | Dir | En | Mb | Sign | Dir | En | Ma |

# Chapter 5

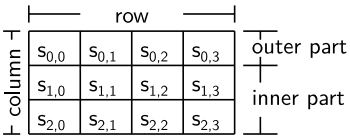
# Symmetric-key hardware implementations

Symmetric-key cryptography is a very broad subject and often the subject of many theses, but in this chapter, we will only focus on implementing three different proposals for lightweight authenticated encryption with or without hashing or eXtendable-Output Function (XOF). Lightweight symmetric-key cryptography started after the standardization of AES, where researchers tried to create block-ciphers, hash functions, authenticated encryption, message authentication codes, or permutations that were better in terms of resources, energy, latency, masking, and/or throughput than the standards. The Gimli permutation was proposed as a lightweight permutation when compared with the standard SHA-3 [101]. Gimli's small state was able to get competitive results in a wide array of hardware and software platforms in microcontrollers and servers. In another research direction, the session authenticated encryption (SAE) scheme Friet was proposed to detect single fault attacks during it is computation. Research into lightweight cryptography became so important, that NIST started a competition for lightweight authenticated encryption with optional hashing [133]. To address this call, the cipher suite Subterranean 2.0 that is aimed at hardware deployment was proposed with an SAE and XOF. For the NIST competition, the permutation Gimli was later proposed in an authenticated encryption and hashing modes called Gimli-24-Hash and Gimli-24-Cipher.

**Chapter organization.**

This chapter starts explaining the Gimli permutation and the modes Gimli-24-Hash and Gimli-24-Cipher which provide hash and Authenticated Encryption (AE) in Section 5.1. Later there is a similar explanation for the Friet SAE scheme and the underlying permutation Friet-P in Section 5.2. The last scheme to be shown is Subterranean 2.0 which provides SAE, eXtended Output Function (XOF), and Message Authentication Code (MAC) in Section 5.3. The hardware architecture compatible with the NIST lightweight competition interface of all three is shown in Section 5.4. Finally, Section 5.5 has the final considerations for this chapter.

Figure 5.1: GIMLI state representation




**Common notation applied in Gimli, Friet, and Subterranean 2.0 sections.**

Sections 5.1, 5.2, and 5.3 apply an extra set of operators that did not appear in previous Chapters. Concatenation of arrays (words, bytes, or bits) is denoted by $||$, thus $a||b$ is the addition of an array $b$ to the end of an array $a$. Accessing an element in an array is done through subscript, for example, $a_i$ refers to the i-th position in array $a$, and with a colon : it is possible to slice the array, therefore $a_{1:3}$ is a new array composed of the elements in position 1,2 and 3 of array $a$. And the size of an array or element can be given by enclosing it in vertical bars, thus $|a|$ will be the size of $a$.

## 5.1 Gimli

### Gimli permutation

GIMLI applies a sequence of 24 rounds to a 384-bit state. The state is represented as a $3 \times 4$ matrix of 32-bit words, as shown in Figure 5.1. Each word is an element of $\mathbb{Z}_{2^{32}}$, thus a 32-bit little-endian unsigned integer. The matrix parts can be defined as:

- a column $j$ is a sequence of 96 bits such that $s_j = \{s_{0,j}; s_{1,j}; s_{2,j}\}$
- a row $i$ is a sequence of 128 bits such that $s_i = \{s_{i,0}; s_{i,1}; s_{i,2}; s_{i,3}\}$

GIMLI round is a sequence of three operations:

1. a non-linear layer, specifically a 96-bit SP-box applied to each column.
2. a linear mixing layer in every second round.
3. a constant addition in every fourth round.

The SP-box shown in Figure 5.2 is composed of three sub-operations: rotations of the first and second words; a 3-input nonlinear T-function; and a swap of the first and third words. The linear layer as detailed in Figure 5.3 consists of two swap operations, namely *Small-Swap* and *Big-Swap*. *Small-Swap* occurs every 4 rounds starting from the 1st round. *Big-Swap* occurs every 4 rounds starting from the 3rd round. The constant addition is an XOR of the round number $r$, the round constant `0x9e377900` with the first state word $s_{0,0}$. There are 24 rounds in GIMLI, numbered $24, 23, \ldots, 1$, thus the round number starts at 24 and ends at 1, which is the last round.

Algorithm 17 presents pseudo-code for the full GIMLI permutation. In Algorithm 17 it was opted to unroll four GIMLI rounds, because not every GIMLI round is the same, but four rounds are always the same.

Figure 5.2: GIMLI SP-box applied to a column

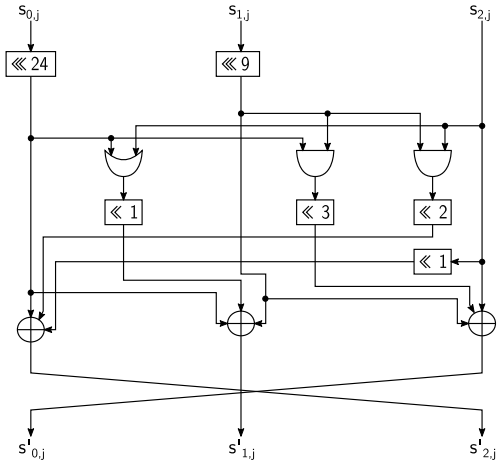



Figure 5.3: GIMLI linear layer


## Gimli Hash and AE

GIMLI-24-HASH and GIMLI-24-CIPHER are respectively the hash function and AE scheme constructed with the GIMLI permutation. The two modes are constructed using the SpongeWrap [23] with minor tweaks as seen in the Ascon family of AE and hashing algorithms [47]. The hash function, GIMLI-24-HASH, applies the permutation GIMLI with 24 rounds and outputs a 256-bit hash. The AE, GIMLI-24-CIPHER, initializes with a 128-bit nonce and 256-bit key, then processes an associated data and a plaintext, and outputs the ciphertext and a 256-bit tag, all with a 24 round GIMLI permutation. In this sponge construction, the outer part is 128-bits and inner part is 256-bits, where the outer part is row 0 and inner part is in rows 1 and 2, as seen in Figure 5.1.

Algorithm 18 specifies three internal procedures applied during hash and encryption/decryption that are used to convert or handle data. The procedure *touint32* is used to convert the input from an array of bytes into 32-bit unsigned integer words, as seen in the GIMLI state words. The procedure *padByte10** adds one byte `0x01` to the input and enough `0x00` bytes to always output a byte array of 128-bits size.

Algorithms 19 and 20 specify the internal sponge procedures used in GIMLI-24-HASH and GIMLI-24-CIPHER. *gAbsorbRate* is only used from the other absorb procedures and can absorb only 128-bits of data, which is the sponge rate. *gAbsorbNone* can absorb a byte array of any size, which is applied for processing the input message during hashing and the associated data during the authenticated encryption. The *gAbsorbNone* first pads the input so it will have a size which is a multiple of 128-

**Algorithm 17** The GIMLI permutation

**Require:** $s = (s_{i,j}), s \in \mathbb{Z}_{2^{32}}^{3\times 4}$
**Ensure:** GIMLI$(s) = (s_{i,j})$
  **for** $r \leftarrow 24$ **downto** 4 **by** 4 **do**
    $s \leftarrow$ SP-box$(s)$ ▷ SP-box
    $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,1}, s_{0,0}, s_{0,3}, s_{0,2}$ ▷ *Small-Swap*
    $s_{0,0} \leftarrow s_{0,0} \oplus$ `0x9e377900` $\oplus\, r$ ▷ Add constant
    $s \leftarrow$ SP-box$(s)$ ▷ SP-box
    $s \leftarrow$ SP-box$(s)$ ▷ SP-box
    $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,2}, s_{0,3}, s_{0,0}, s_{0,1}$ ▷ *Big-Swap*
    $s \leftarrow$ SP-box$(s)$ ▷ SP-box
  **end for**
  **return** $s$

**Interface:** $s \leftarrow$ SP-box$(s)$
  **for** $j \leftarrow 0$ **to** 3 **by** 1 **do**
    $x \leftarrow (s_{0,j} \lll 24)$; $y \leftarrow (s_{1,j} \lll 9)$; $z \leftarrow s_{2,j}$
    $s_{2,j} \leftarrow x \oplus (z \ll 1) \oplus ((y \wedge z) \ll 2)$
    $s_{1,j} \leftarrow y \oplus \quad x \quad \oplus ((x \vee z) \ll 1)$
    $s_{0,j} \leftarrow z \oplus \quad y \quad \oplus ((x \wedge y) \ll 3)$
  **end for**
  **return** $s$

bits, then it absorbs blocks of 128-bits, until the last block where it will flip one bit in $s_{2,3}$ before absorbing it. The authenticated encryption also applies another absorption procedure for the plaintext/ciphertext called *gAbsorbEncrypt*/*gAbsorbDecrypt*, which not only absorbs the value into the state but also generates the ciphertext/plaintext. The *gAbsorbEncrypt*/*gAbsorbDecrypt* works almost the same as the *gAbsorbNone*, except it extracts 128-bits of data which are used to generate the ciphertext/plaintext and absorbs the plaintext. The *gExtract* procedure obtains 128-bits of data out of the sponge, which is required for the hash output and the tag in authenticated encryption.

**Algorithm 18** GIMLI common procedures.

**Internal interface:** $o \leftarrow$ touint32$(a_0, a_1, a_2, a_3)$, $a \in \mathbb{Z}_{2^8}^4$, $o \in \mathbb{Z}_{2^{32}}$
  $o \leftarrow a_0 + 2^8 \cdot a_1 + 2^{16} \cdot a_2 + 2^{24} \cdot a_3$
  **return** $o$

**Internal interface:** $o \leftarrow$ padByte10*$(a)$, $a \in \mathbb{Z}_{2^8}^{al}$, $o \in \mathbb{Z}_{2^8}^{ol}$
  $ol \leftarrow \lceil (al+1)/16 \rceil \cdot 16$
  $o \leftarrow a || $ `0x01` $|| ($`0x00`$)^{ol-al-1}$
  **return** $o$

GIMLI-24-HASH, presented by Algorithm 21 pseudo-code, starts by initializing a 384-bit GIMLI state to all-zero, followed by absorbing the message as an array of bytes. Then the next step is to extract data from the sponge by copying the outer

**Algorithm 19** GIMLI sponge procedures part 1.

**Internal interface:** $s \leftarrow \text{gAbsorbRate}(s, X, z)$, $X \in \mathbb{Z}_{2^8}^t$, $s \in \mathbb{Z}_{2^{32}}^{3\times4}$
  **for** $j \leftarrow 0$ **to** 3 **by** 1 **do**
    $s_{0,j} \leftarrow s_{0,j} \oplus \text{touint32}(X_{4j+z}, X_{4j+1+z}, X_{4j+2+z}, X_{4j+3+z})$
  **end for**
  $s \leftarrow \text{GIMLI}(s)$
  **return** $s$

**Interface:** $s \leftarrow \text{gAbsorbNone}(s, X)$, $X \in \mathbb{Z}_{2^8}^t$, $s \in \mathbb{Z}_{2^{32}}^{3\times4}$
  $X \leftarrow \text{padByte10*}(X)$
  **for** $z \leftarrow 0$ **to** $t - 16$ **by** 16 **do**
    $s \leftarrow \text{gAbsorbRate}(s, X, z)$
  **end for**
  $s_{2,3} \leftarrow s_{2,3} \oplus 0x00000001$
  $s \leftarrow \text{gAbsorbRate}(s, X, t - 16)$
  **return** $s$

**Interface:** $o \leftarrow \text{gExtract}(s), o \in \mathbb{Z}_{2^8}^{16}, s \in \mathbb{Z}_{2^{32}}^{3\times4}$
  **for** $j \leftarrow 0$ **to** 3 **by** 1 **do**
    **for** $z \leftarrow 0$ **to** 3 **by** 1 **do**
      $o_{4j+z} \leftarrow (s_{0,j}/2^{8z}) \bmod 2^8$
    **end for**
  **end for**
  **return** $o$

part into the output hash, but since the outer part only gives 128-bits after the first copy a GIMLI permutation is performed in the whole state and another 128-bits outer part is appended to the first, thus giving 256-bits.

GIMLI-24-CIPHER is summarized in Algorithms 22 and 23. Encryption is done in Algorithm 22, which starts by filling the 384-bits state with a 128-bit nonce and 256-bit key, just as in the Salsa stream cipher [15], then a GIMLI permutation is applied. After the state is initialized, then the associated data $A$ is absorbed followed by the plaintext $P$, which outputs the ciphertext $C$. After processing all data, then a final extract is performed to get the tag $T$. Decryption in Algorithm 23 behaves almost the same as encryption with two exceptions: first, the algorithm obtains the original plaintext and then absorbs it to have a matching state with encryption, and second, the tag is compared with the one received, thus validating the plaintext.

### Hardware implementation of the Gimli permutation

Three main hardware architectures of the GIMLI permutation are presented to address different hardware applications. These different architectures are a tradeoff between resources, maximum operational frequency, and the number of cycles necessary to perform the full permutation. Even with these differences, all three architectures share a common simple communication interface that can be expanded to offer different operation modes. All this was done in VHDL and tested in ModelSim for behavioral

**Algorithm 20** GIMLI sponge procedures part 2.

**Interface:** $s, C \leftarrow \text{gAbsorbEncrypt}(s, P)$, $P \in \mathbb{Z}_{2^8}^t$, $s \in \mathbb{Z}_{2^{32}}^{3\times4}$, $C \in \mathbb{Z}_{2^8}^t$.
  $P \leftarrow \text{padByte10*}(P)$
  **for** $z \leftarrow 0$ **to** $t-16$ **by** 16 **do**
    $k \leftarrow \text{gExtract}(s)$
    **for** $w \leftarrow 0$ **to** 15 **by** 1 **do**
      $C_{w+z} \leftarrow k_w \oplus P_{w+z}$
    **end for**
    $s \leftarrow \text{gAbsorbRate}(s, P, z)$
  **end for**
  $k \leftarrow \text{gExtract}(s)$
  **for** $w \leftarrow 0$ **to** 15 **by** 1 **do**
    $C_{w+t-16} \leftarrow k_w \oplus P_{w+t-16}$
  **end for**
  $s_{2,3} \leftarrow s_{2,3} \oplus 0x00000001$
  $s \leftarrow \text{gAbsorbRate}(s, P, t-16)$
  $C \leftarrow C_{0:t}$ ▷ Only the first t-bytes of $C$ are returned.
  **return** $s, C$

**Interface:** $s, P \leftarrow \text{gAbsorbDecrypt}(s, C)$, $C \in \mathbb{Z}_{2^8}^t$, $s \in \mathbb{Z}_{2^{32}}^{3\times4}$, $P \in \mathbb{Z}_{2^8}^t$.
  $C \leftarrow \text{padByte10*}(C)$
  **for** $z \leftarrow 0$ **to** $t-16$ **by** 16 **do**
    $k \leftarrow \text{gExtract}(s)$
    **for** $w \leftarrow 0$ **to** 15 **by** 1 **do**
      $P_{w+z} \leftarrow k_w \oplus C_{w+z}$
    **end for**
    $s \leftarrow \text{gAbsorbRate}(s, P, z)$
  **end for**
  $k \leftarrow \text{gExtract}(s)$
  **for** $w \leftarrow 0$ **to** 15 **by** 1 **do**
    $P_{w+t-16} \leftarrow k_w \oplus C_{w+t-16}$
  **end for**
  $s_{2,3} \leftarrow s_{2,3} \oplus 0x00000001$
  $s \leftarrow \text{gAbsorbRate}(s, P, t-16)$
  $P \leftarrow P_{0:t}$ ▷ Only the first t-bytes of $P$ are returned.
  **return** $s, P$

**Algorithm 21** The GIMLI-24-HASH.

**Require:** $P \in \mathbb{Z}_{2^8}^{pl}$, $s \in \mathbb{Z}_{2^{32}}^{3\times4}$
**Ensure:** $\text{GIMLI-24-HASH}(P) = h \in \mathbb{Z}_{2^8}^{32}$
  $s \leftarrow 0$ ▷ Initialize all state positions with 0
  $s \leftarrow \text{gAbsorbNone}(s, P)$
  $h_0..h_{15} \leftarrow \text{gExtract}(s)$
  $s \leftarrow \text{GIMLI}(s)$
  $h_{16}..h_{31} \leftarrow \text{gExtract}(s)$
  **return** $h$

**Algorithm 22** The GIMLI-24-CIPHER encryption.

**Require:** $K \in \mathbb{Z}_{2^8}^{32}$, $N \in \mathbb{Z}_{2^8}^{16}$, $P \in \mathbb{Z}_{2^8}^{pl}$, $A \in \mathbb{Z}_{2^8}^{al}$, $s \in \mathbb{Z}_{2^{32}}^{3\times4}$
**Ensure:** GIMLI-24-CIPHER-ENCRYPT$(K, N, A, P) = C, T$ $C \in \mathbb{Z}_{2^8}^{pl}$, $T \in \mathbb{Z}_{2^8}^{16}$
  **for** $j \leftarrow 0$ **to** 3 **by** 1 **do** ▷ Initialize all state positions with $N$ and $K$
    $s_0, j \leftarrow \text{touint32}(N_{4j}, N_{4j+1}, N_{4j+2}, N_{4j+3})$
    $s_1, j \leftarrow \text{touint32}(K_{4j}, K_{4j+1}, K_{4j+2}, K_{4j+3})$
    $s_2, j \leftarrow \text{touint32}(K_{4j+16}, K_{4j+1+16}, K_{4j+2+16}, K_{4j+3+16})$
  **end for**
  $s \leftarrow \text{GIMLI}(s)$
  $s \leftarrow \text{gAbsorbNone}(s, A)$
  $s, C \leftarrow \text{gAbsorbEncrypt}(s, P)$
  $T \leftarrow \text{gExtract}(s)$
  **return** $C, T$

**Algorithm 23** The GIMLI-24-CIPHER decryption.

**Require:** $K \in \mathbb{Z}_{2^8}^{32}$, $N \in \mathbb{Z}_{2^8}^{16}$, $T \in \mathbb{Z}_{2^8}^{16}$, $C \in \mathbb{Z}_{2^8}^{pl}$, $A \in \mathbb{Z}_{2^8}^{al}$, $s \in \mathbb{Z}_{2^{32}}^{3\times4}$
**Ensure:** GIMLI-24-CIPHER-DECRYPT$(K, N, A, C, T) = P \in \mathbb{Z}_{2^8}^{pl}$
  **for** $z \leftarrow 0$ **to** 3 **by** 1 **do** ▷ Initialize all state positions with $N$ and $K$
    $s_0, z \leftarrow \text{touint32}(N_{4\cdot z}, N_{4\cdot z+1}, N_{4\cdot z+2}, N_{4\cdot z+3})$
    $s_1, z \leftarrow \text{touint32}(K_{4\cdot z}, K_{4\cdot z+1}, K_{4\cdot z+2}, K_{4\cdot z+3})$
    $s_2, z \leftarrow \text{touint32}(K_{16+4\cdot z}, K_{16+4\cdot z+1}, K_{16+4\cdot z+2}, K_{16+4\cdot z+3})$
  **end for**
  $s \leftarrow \text{GIMLI}(s)$
  $s \leftarrow \text{gAbsorbNone}(s, A)$
  $s, P \leftarrow \text{gAbsorbDecrypt}(s, C)$
  $T' \leftarrow \text{gExtract}(s)$
  **if** $(T' \neq T)$ **then return** error
  **return** $M$

results, synthesized and tested for FPGAs with Xilinx ISE 14.7. In the case of ASICs, this was done through Synopsis Ultra and Simple Compiler with 180nm UMC L180, and Encounter RTL Compiler with ST 28nm FDSOI technology.

The first architecture, depicted in Figure 5.4, performs a certain number of rounds in one clock cycle and stores the output in the same buffer as the input. The number of rounds it can perform in one cycle is chosen before the synthesis process and can be 1, 2, 3, 4, 6, or 8. In the case of 12 or 24 combinational rounds, optimized architectures for these cases were done to have better results. The rounds themselves are computed as shown in Figure 5.5. In every round, there is one SP-box application on the whole state, followed by the linear layer. In the linear layer, the operation can be a small swap with round constant addition, a big swap, or no operation, which are chosen according to the two least significant bits of the round number. The round number starts from 24 and is decremented by one in each combinational round block.

Besides the round and the optimized half and full combinational architectures, there is also a serial-based architecture illustrated in Figure 5.6. The serial-based

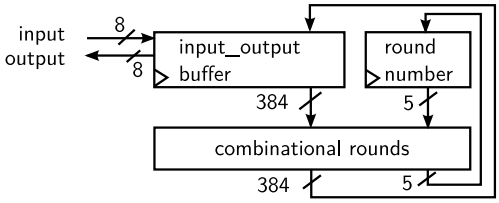



Figure 5.4: Round-based architecture

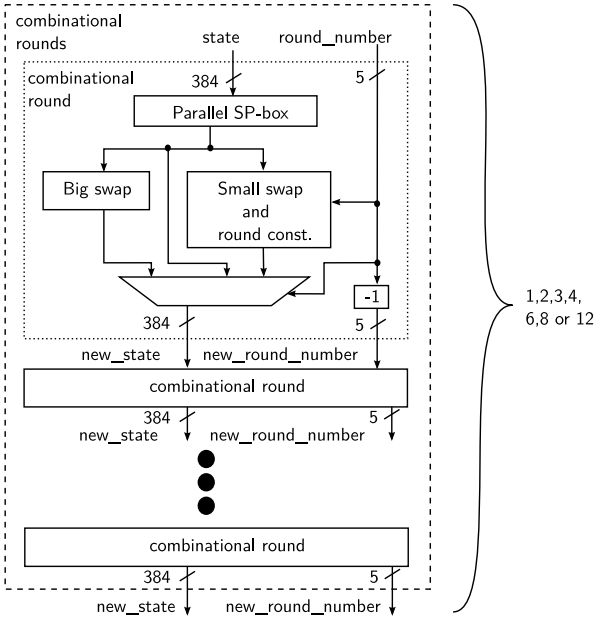



Figure 5.5: Combinational round in round-based architecture

architecture performs one SP-box application per cycle, through a circular-shift-based architecture, therefore taking in total four cycles. In the case of the linear layer, it is still executed in one cycle in parallel. The reason for not being done in a serial-based manner is because the parallel version cost is very low.

All hardware results for the Gimli and other permutations comparison are shown in Table 5.1 from Bernstein et al .[17]. For FPGAs, the lowest latency is the one with four combinational rounds in one cycle, and the one with the best Resources×Time/State is the one with two combinational rounds. For ASICs, the results change as the lowest latency is the one with full combinational setting, and the one with best Resources×Time/State is the one with eight combinational rounds for 180nm and four combinational rounds for 28nm. This difference illustrates that each technology can give different results, making it difficult to compare results on different technologies.

Hardware variants that do two or four rounds in one cycle appear to be attractive

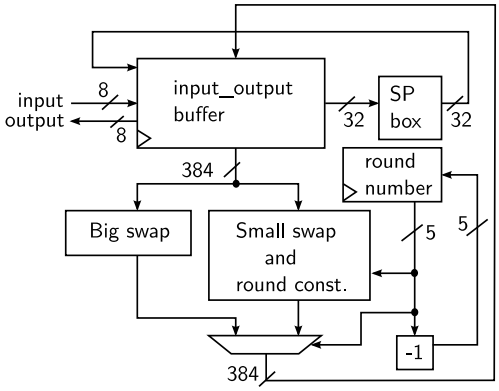



Figure 5.6: Serial-based architecture

choices, depending on the application scenario. The serial version needs 4.5 times more cycles than the 1-round version while saving around 28% of the gate equivalents (GE) in the 28nm ASIC technology, and less in the other ASIC technology and FPGA. If resource constraints are extreme enough to justify the serial version then it would be useful to develop a new version optimized for the target technology, for better results.

To compare the Gimli permutation to other permutations in the literature, the other permutations were all synthesized with similar half-combinational architectures, taking exactly 2 cycles to perform a permutation. The permutations that were chosen for comparison were selected close to Gimli in terms of size, even though in the end the final metric was divided by the permutation size to try to "normalize" the results.

The best results in Resources×Time/State are from 24-round Gimli and 12-round Ascon-128, with Ascon slightly more efficient in the FPGA results and Gimli more efficient in the ASIC results. Both permutations in all 3 technologies had very similar results, while Keccak-$f$[400] is worse in all 3 technologies. The permutations SPONGENT-256/256/128, Photon-256/32/32, and C-Quark have a much higher resource utilization in all technologies. This is because they were designed to work with little resources in exchange for a very high response time (e.g., SPONGENT is reported to use 2641 GE for 18720 cycles, or 5011 GE for 195 cycles), therefore changing the resource utilization from logic gates to time. Gimli and Ascon are the most efficient in the sense of offering a similar security level to SPONGENT, Photon and C-Quark, with a lower product of time and logic resources.

Table 5.1: Hardware results for Gimli and competitors.
Gates Equivalent(GE). Slice(S). LUT(L). Flip-Flop(F).
* Could not finish the place and route.

| Perm. | State size | Version | Cycles | Resources | Period (ns) | Time (ns) | Res.×Time/ State |
|---|---|---|---|---|---|---|---|
| FPGA – Xilinx Spartan 6 LX75 | | | | | | | |
| Ascon | 320 | | 2 | 732 S(2700 L+325 F) | 34.570 | 70 | 158.2 |
| Gimli | 384 | 12 | 2 | 1224 S(4398 L+389 F) | 27.597 | 56 | 175.9 |
| Keccak | 400 | | 2 | 1520 S(5555 L+405 F) | 77.281 | 155 | 587.3 |
| C-quark* | 384 | | 2 | 2630 S(9718 L+389 F) | 98.680 | 198 | 1351.7 |
| Photon | 288 | | 2 | 2774 S(9430 L+293 F) | 74.587 | 150 | 1436.8 |
| Spongent* | 384 | | 2 | 7763 S(19419 L+389 F) | 292.160 | 585 | 11812.7 |
| Gimli | 384 | 24 | 1 | 2395 S(8769 L+385 F) | 56.496 | 57 | 352.4 |
| Gimli | 384 | 8 | 3 | 831 S(2924 L+390 F) | 24.531 | 74 | 159.3 |
| Gimli | 384 | 6 | 4 | 646 S(2398 L+390 F) | 18.669 | 75 | 125.6 |
| Gimli | 384 | 4 | 6 | 415 S(1486 L+391 F) | 8.565 | 52 | 55.5 |
| Gimli | 384 | 3 | 8 | 428 S(1587 L+393 F) | 10.908 | 88 | 97.3 |
| Gimli | 384 | 2 | 12 | 221 S(815 L+392 F) | 5.569 | 67 | 38.5 |
| Gimli | 384 | 1 | 24 | 178 S(587 L+394 F) | 4.941 | 119 | 55.0 |
| Gimli | 384 | Serial | 108 | 139 S(492 L+397 F) | 3.996 | 432 | 156.2 |
| 28nm ASIC – ST 28nm FDSOI technology | | | | | | | |
| Gimli | 384 | 12 | 2 | 35452GE | 2.2672 | 5 | 418.6 |
| Ascon | 320 | | 2 | 32476GE | 2.8457 | 6 | 577.6 |
| Keccak | 400 | | 2 | 55683GE | 5.6117 | 12 | 1562.4 |
| C-quark | 384 | | 2 | 111852GE | 9.9962 | 20 | 5823.4 |
| Photon | 288 | | 2 | 296420GE | 10.0000 | 20 | 20584.7 |
| Spongent | 384 | | 2 | 1432047GE | 12.0684 | 25 | 90013.1 |
| Gimli | 384 | 24 | 1 | 66205GE | 4.2870 | 5 | 739.1 |
| Gimli | 384 | 8 | 3 | 25224GE | 1.5921 | 5 | 313.7 |
| Gimli | 384 | 6 | 4 | 21675GE | 2.1315 | 9 | 481.2 |
| Gimli | 384 | 4 | 6 | 14999GE | 1.0549 | 7 | 247.2 |
| Gimli | 384 | 3 | 8 | 14808GE | 2.0119 | 17 | 620.6 |
| Gimli | 384 | 2 | 12 | 10398GE | 1.0598 | 13 | 344.4 |
| Gimli | 384 | 1 | 24 | 8097GE | 1.0642 | 26 | 538.5 |
| Gimli | 384 | Serial | 108 | 5843GE | 1.5352 | 166 | 2522.7 |
| 180nm ASIC – UMC L180 | | | | | | | |
| Gimli | 384 | 12 | 2 | 26685 | 9.9500 | 20 | 1382.9 |
| Ascon | 320 | | 2 | 23381 | 11.4400 | 23 | 1671.7 |
| Keccak | 400 | | 2 | 37102 | 22.4300 | 45 | 4161.0 |
| C-quark | 384 | | 2 | 62190 | 37.2400 | 75 | 12062.1 |
| Photon | 288 | | 2 | 163656 | 99.5900 | 200 | 113183.8 |
| Spongent | 384 | | 2 | 234556 | 99.9900 | 200 | 122151.9 |
| Gimli | 384 | 24 | 1 | 53686 | 17.4500 | 18 | 2439.6 |
| Gimli | 384 | 8 | 3 | 19393 | 7.9100 | 24 | 1198.4 |
| Gimli | 384 | 6 | 4 | 15886 | 12.5100 | 51 | 2070.0 |
| Gimli | 384 | 4 | 6 | 11008 | 10.1700 | 62 | 1749.1 |
| Gimli | 384 | 3 | 8 | 10106 | 10.0500 | 81 | 2115.8 |
| Gimli | 384 | 2 | 12 | 7112 | 15.2000 | 183 | 3377.8 |
| Gimli | 384 | 1 | 24 | 5314 | 9.5200 | 229 | 3161.4 |
| Gimli | 384 | Serial | 108 | 3846 | 11.2300 | 1213 | 12146.0 |

## 5.2 Friet

Friet [128] is a session authenticated encryption scheme that is based on the sponge duplex construction with Friet-P as an internal permutation. Friet-C is analogous to Friet, with the only difference being the use of the permutation Friet-PC instead of Friet-P. The latter is in fact Friet-PC, but without the fault detection mechanism. Because the fault detection mechanism does not add any extra information that affects the cryptanalysis, then the same was done in Friet-C and Friet-PC, and then assumed to be the same in Friet and Friet-P. During the evaluation of Friet and Friet-P, Friet-C and Friet-PC are used as our baseline comparison to show how much it costs, in terms of area and time, to add the fault detection mechanism.

In this section, the term *limb* denotes a 128-bit word or an element in $\mathbb{F}_2^{128}$. The bits in the limbs are indexed by a subscript $i$ ranging from 0 to 127.

First, the permutation Friet-PC is introduced, followed by Friet-P, and then Friet and Friet-C. After the four schemes have been presented, then the hardware architecture of Friet and Friet-P is described.

### The Permutation Friet-PC

The state of Friet-PC has a width of 384 bits that can be split into three limbs denoted $a$, $b$ and $c$ as depicted in Figure 5.7. The outer part of the sponge, which are the bits that are externally affected, belongs to limb $a$ and the bits of limb $b$ with indices 0 and 1, while the remaining bits are the inner part, which are not directly manipulated or seen. The permutation nominal number of rounds is 24 and the round function $\mathrm{RF}_i$ is composed of six steps: $\mathrm{RF}_i = \xi \circ \tau_2 \circ \mu_2 \circ \mu_1 \circ \tau_1 \circ \delta_i(a, b, c)$ which can be seen graphically in Figure 5.8. The full permutation is specified in Algorithm 24 and the round constants in Table 5.2.

Figure 5.7: Friet-PC state composition of limbs $a$,$b$,$c$.

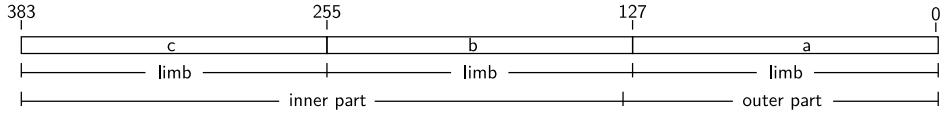



Table 5.2: Friet-P round constants $\mathrm{rc}_i \in \mathbb{F}_2^{128}$ in hexadecimal notation and big endian format, omitting the leading zero digits.

| $i$ | $\mathrm{rc}_i$ | $i$ | $\mathrm{rc}_i$ | $i$ | $\mathrm{rc}_i$ | $i$ | $\mathrm{rc}_i$ |
|---|---|---|---|---|---|---|---|
| 0 | 1111 | 6 | 110 | 12 | 1 | 18 | 1010 |
| 1 | 11100000 | 7 | 11000000 | 13 | 110000 | 19 | 1010000 |
| 2 | 1101 | 8 | 1001 | 14 | 111 | 20 | 1011 |
| 3 | 10100000 | 9 | 100000 | 15 | 11110000 | 21 | 1100000 |
| 4 | 101 | 10 | 100 | 16 | 1110 | 22 | 1100 |
| 5 | 10110000 | 11 | 10000000 | 17 | 11010000 | 23 | 10010000 |

**Algorithm 24** FRIET-PC.

**Require:** $a, b, c \in \mathbb{F}_2^{128}$
**Ensure:** $(a', b', c') \leftarrow \text{FRIET-PC}(a, b, c)$
**for** Round index $i$ from 0 to 23 **do**

$$
\begin{array}{lcll}
(a,b,c) & \leftarrow & (a,b,c \oplus \mathrm{rc}_i) & \delta_i \\
(a,b,c) & \leftarrow & (a \oplus b \oplus c, c, a) & \tau_1 \\
(a,b,c) & \leftarrow & (a, b \oplus (c \lll 1), c) & \mu_1 \\
(a,b,c) & \leftarrow & (a, b, c \oplus (b \lll 80)) & \mu_2 \\
(a,b,c) & \leftarrow & (a, a \oplus b \oplus c, c) & \tau_2 \\
(a,b,c) & \leftarrow & (a \oplus ((b \lll 36) \wedge (c \lll 67)), b, c) & \xi
\end{array}
$$

**end for**
**return** $(a, b, c)$

Figure 5.8: Round function of FRIET-PC.

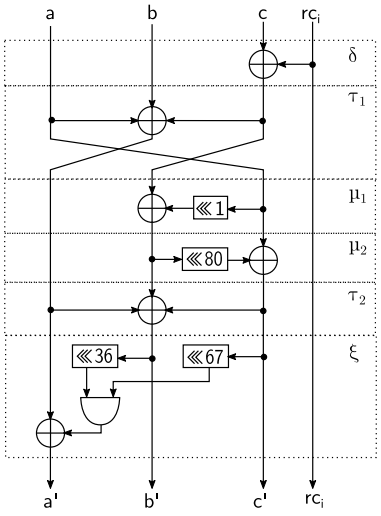
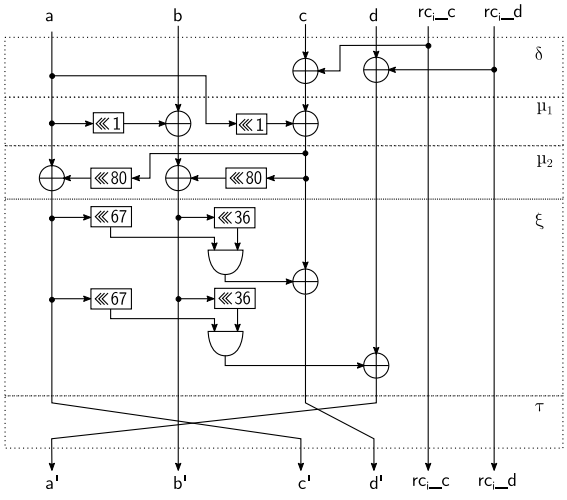



Figure 5.9: Round function of FRIET-P.




## The Round Function of Code-abiding Permutation Friet-P

A $[n, k, d]_q$ linear block code $C$ over $\mathbb{F}_q$ is a $k$-dimensional subspace of the $n$-dimensional vector space $V_n(\mathbb{F}_q) = \{(x_l, ..., x_n) : x_i \in \mathbb{F}_q\}$; $d = min\{d_H(x_i, x_j) : i, j \in [1, n], x_i \neq x_j, x_i, x_j \in V_n(\mathbb{F}_q)\}$ and $d_H(x, y) =$ the number of components which $x_i \neq y_i$; $n$ is called the length of the code, $k$ the dimension, d the minimum Hamming distance [89]. A single-parity-check code, or in short parity code, can be defined as $[n, n-1, d]_q$ where the sum of all elements inside a codeword is zero.

A code-abiding permutation can be defined as a permutation $f$ on the set of $n$-symbol vectors is *code-abiding* for code $C$ if $f(C) = C$ [128].

FRIET-P is a code-abiding permutation of FRIET-PC with the parity code $[4, 3, 2]_2$.

FRIET-P has width of 512 bits, i.e., 4 limbs.

Let $d$ be the parity limb that should satisfy the parity equation $d = a \oplus b \oplus c$ after any step of the round function. From that, one can directly derive the round function of FRIET-P from the round function that is specified in Algorithm 24, by substituting $(a \oplus b \oplus c)$ by $d$ in $\tau_1$ and $\tau_2$ steps and duplicating in the remainder steps $\delta_i$, $\mu_1$, $\mu_2$, and $\xi$. As a result, we have:

$$\begin{array}{lcll}
(a,b,c,d) & \leftarrow & (a,b,c \oplus \mathrm{rc}_i, d \oplus \mathrm{rc}_i) & \delta_i \\
(a,b,c,d) & \leftarrow & (d,c,a,b) & \tau_1 \\
(a,b,c,d) & \leftarrow & (a,b \oplus (c \lll 1), c, d \oplus (c \lll 1)) & \mu_1 \\
(a,b,c,d) & \leftarrow & (a,b,c \oplus (b \lll 80), d \oplus (b \lll 80)) & \mu_2 \\
(a,b,c,d) & \leftarrow & (a,d,c,b) & \tau_2 \\
(a,b,c,d) & \leftarrow & (a \oplus ((b \lll 36) \wedge (c \lll 67)), b, c, d \oplus ((b \lll 36) \wedge (c \lll 67))) & \xi
\end{array}$$

$\tau_1$ and $\tau_2$ are transferred to the end and merged, yielding:

$$\begin{array}{lcll}
(a,b,c,d) & \leftarrow & (a,b,c \oplus \mathrm{rc}_i, d \oplus \mathrm{rc}_i) & \delta_i \\
(a,b,c,d) & \leftarrow & (a,b \oplus (a \lll 1), c \oplus (a \lll 1), d) & \mu_1 \\
(a,b,c,d) & \leftarrow & (a \oplus (c \lll 80), b \oplus (c \lll 80), c, d) & \mu_2 \\
(a,b,c,d) & \leftarrow & (a,b,c \oplus ((b \lll 36) \wedge (a \lll 67)), d \oplus ((b \lll 36) \wedge (a \lll 67))) & \xi
\end{array}$$

Consequently, the round function of FRIET-P becomes $\mathrm{RF}_i = \xi \circ \mu_2 \circ \mu_1 \circ \delta_i(a,b,c,d)$ as depicted in Figure 5.9.

### The Session Authenticated Encryption Scheme Friet

FRIET and FRIET-C are permutation-based session authenticated encryption (SAE) schemes and use SpongeWrap [21] which is a mode on top of the duplex [21] construction. The duplex construction has been used in other proposals, for example, the CAESAR [9] candidate Ketje [24] and NIST lightweight competition submissions Ascon [47], Gimli [16] and Xoodyak [40].

The fault detection capabilities of the AE scheme FRIET are achieved thanks to the underlying permutation FRIET-P and the application of easily convertible code abiding operations. The security analysis of FRIET is done in the same molds as the permutation FRIET-P, where an equivalent scheme without fault detection is used, FRIET-C, with the permutation FRIET-PC in the SpongeWrap construction.

#### The Permutation SAE SpongeWrap Mode

The SAE mode relies on the SpongeWrap that is a mode on top of the Duplex construction [21]. More precisely, an SAE scheme converts sequences of messages, each consisting of associated data $A$ and a plaintext $P$, into sequences of cryptograms, each consisting of associated data, ciphertext $C$ (the enciphered plaintext), and a tag $T$. The messages and cryptograms are bit strings of arbitrary length, and the associated data is optional for the input messages and, as a consequence, the cryptogram. The session aspect is related to the tag $T$: this is not only computed on the associated data and ciphertext of its own cryptogram but the full sequence of cryptograms that were generated since the start of the session. In other words, an SAE scheme is *stateful* and it behaves exactly as an AE when there is only one session.

**Algorithm 25** FRIET-C instantiations of SpongeWrap$[f_C, \rho, \tau]$, with permutation $f_C$ =FRIET-PC, state $s$ with $b = 384$ bits, block length $\rho = 128$ and tag length $\tau = 128$. SAE interface.

**Interface:** $s, T \leftarrow \text{fStart}(K, D)$, $K \in \mathbb{F}_2^{kl}$, $D \in \mathbb{F}_2^{dl}$, $T \in \mathbb{F}_2^{128}$, $s \in \mathbb{F}_2^b$
  $s \leftarrow 0^{384}$
  $s \leftarrow \text{fAbsorbNone}(s, K)$
  $s \leftarrow \text{fAbsorbEncrypt}(s, D)$
  $s, T \leftarrow \text{fSqueeze}(s, \tau)$
  **return** $s, T$

**Interface:** $s, C, T \leftarrow \text{fWrap}(s, A, P)$, $A \in \mathbb{F}_2^{al}$, $P \in \mathbb{F}_2^{pl}$, $C \in \mathbb{F}_2^{pl}$, $T \in \mathbb{F}_2^{128}$, $s \in \mathbb{F}_2^b$
  $s \leftarrow \text{fAbsorbNone}(s, A)$
  $s, C \leftarrow \text{fAbsorbEncrypt}(s, P)$
  $s, T \leftarrow \text{fSqueeze}(s, \tau)$
  **return** $s, C, T$

**Interface:** $s, P \leftarrow \text{fUnwrap}(s, A, C, T)$, $A \in \mathbb{F}_2^{al}$, $C \in \mathbb{F}_2^{pl}$, $P \in \mathbb{F}_2^{pl}$, $T, T' \in \mathbb{F}_2^{128}$, $s \in \mathbb{F}_2^b$
  $s \leftarrow \text{fAbsorbNone}(s, A)$
  $s, P \leftarrow \text{fAbsorbDecrypt}(s, C)$
  $s, T' \leftarrow \text{fSqueeze}(s, \tau)$
  **if** $(T' \neq T)$ **then return** error
  **return** $s, P$

The SpongeWrap presented here contains three minor modifications compared to the original proposal by Bertoni et al. [21]. For the first modification, in the session startup we absorb a dedicated non-secret diversifier $D$ that is a nonce for sessions started with the same key $K$. Second, after the key and diversifier is set up, we return a tag for them which can be used to validate if the correct key and diversifier has been absorbed. Third, there is no restriction on the tag length, i.e., it can be longer than the sponge outer part. This modified version of the SpongeWrap mode with the duplex construction integrated is described in Algorithms 25, 26, and 27. In the latter, all parameters are arbitrary-length bit strings with $|X|$ denoting the length of a string $X$ in bits.

SpongeWrap has a $b$-bit state, with $b$ the width of the underlying permutation $f_C$. It has a *block length* $\rho$ and all input strings are first split up into $\rho$-bit blocks, where the last block can be shorter than $\rho$ bits. Before the absorption of a block in the state, SpongeWrap appends a domain separation bit to indicate whether the next output is used as keystream or as a tag (`1`), or not at all (`0`). Then, the block is padded with a single `1` followed by `0`s. The so-called *duplex rate* $r$ is the size of the *outer part* of the state that is directly affected by the absorbing. Due to the domain separation bit and the first bit of the padding, we have $r = \rho + 2$. The remaining part of the state is called the *inner part* and its size is called the *capacity* $c$. We have $c = b - r = b - \rho - 2$.

In FRIET and FRIET-C, the block length $\rho = 128$ implies a rate $r = \rho + 2 = 130$

**Algorithm 26** FRIET-C instantiations of SpongeWrap$[f_C, \rho, \tau]$, with permutation $f_C$ =FRIET-PC, state $s$ with $b = 384$ bits, block length $\rho = 128$ and tag length $\tau = 128$. Internal functions part 1.

**Interface:** $s \leftarrow \text{fAbsorbNone}(s, X)$, $X \in \mathbb{F}_2^{xl}$, $s \in \mathbb{F}_2^b$
  **for** $i \leftarrow 0$ **to** $xl - 129$ **by** 128 **do**
    $s \leftarrow \text{fDuplex}(s, X_{i:i+127}||0)$
  **end for**
  $s \leftarrow \text{fDuplex}(s, X_{i:xl-1}||1)$
  **return** $s$

**Interface:** $s, Y \leftarrow \text{fAbsorbEncrypt}(s, X)$, $X \in \mathbb{F}_2^{xl}$, $Y \in \mathbb{F}_2^{xl}$, $s \in \mathbb{F}_2^b$
  **for** $i \leftarrow 0$ **to** $xl - 129$ **by** 128 **do**
    $Y_{i:i+127} \leftarrow X_{i:i+127} + s_{0:127}$
    $s \leftarrow \text{fDuplex}(s, X_{i:i+127}||1)$
  **end for**
  $Y_{i:xl-1} \leftarrow X_{i:xl-1} + s_{0:(xl-1-i)}$
  $s \leftarrow \text{fDuplex}(s, X_{i:xl-1}||0)$
  **return** $s, Y$

**Interface:** $s, Y \leftarrow \text{fAbsorbDecrypt}(s, X)$, $X \in \mathbb{F}_2^{xl}$, $Y \in \mathbb{F}_2^{xl}$, $s \in \mathbb{F}_2^b$
  **for** $i \leftarrow 0$ **to** $xl - 129$ **by** 128 **do**
    $Y_{i:i+127} \leftarrow X_{i:i+127} + s_{0:127}$
    $s \leftarrow \text{fDuplex}(s, Y_{i:i+127}||1)$
  **end for**
  $Y_{i:xl-1} \leftarrow X_{i:xl-1} + s_{0:(xl-1-i)}$
  $s \leftarrow \text{fDuplex}(s, Y_{i:xl-1}||0)$
  **return** $s, Y$

**Algorithm 27** FRIET-C instantiations of SpongeWrap$[f_C, \rho, \tau]$, with permutation $f_C$ =FRIET-PC, state $s$ with $b = 384$ bits, block length $\rho = 128$ and tag length $\tau = 128$. Internal functions part 2.

**Interface:** $s \leftarrow \text{fDuplex}(s, \sigma)$, $\sigma \in \mathbb{F}_2^t$, $1 \leq t \leq 129$, $s \in \mathbb{F}_2^b$
  $s \leftarrow s + \sigma||1||0^{129-t}$
  $s \leftarrow f_C(s)$
  **return** $s$

**Interface:** $s, Z \leftarrow \text{fSqueeze}(s, l)$, $Z \in \mathbb{F}_2^l$, $s \in \mathbb{F}_2^b$
  **for** $i \leftarrow 0$ **to** $l - 129$ **by** 128 **do**
    $Z_{i:i+127} \leftarrow s_{0:127}$
    $s \leftarrow \text{fDuplex}(s, 0)$
  **end for**
  $Z_{i:l} \leftarrow s_{0:l}$
  $s \leftarrow \text{fDuplex}(s, 0)$
  **return** $s, Z$

**Algorithm 28** Modifications in FRIET-C instantiation of SpongeWrap$[f, \rho, \tau]$ to work with FRIET. In this case the code abiding permutation $f$ =FRIET-P, state $s$ with $b = 512$ bits, parity state $s_d$, block length $\rho = 128$ and tag length $\tau = 128$. The remaining interfaces are the same as in FRIET-C Algorithms 25 and 26.

**Interface:** $T \leftarrow \text{fStart}(s, K, D)$, $K \in \mathbb{F}_2^{kl}$, $D \in \mathbb{F}_2^{dl}$, $T \in \mathbb{F}_2^{128}$, $s \in \mathbb{F}_2^{512}$
  $s \leftarrow 0^{512}$
  $s \leftarrow \text{fAbsorbNone}(s, K)$
  $s \leftarrow \text{fAbsorbEncrypt}(s, D)$
  $s, T \leftarrow \text{fSqueeze}(s, \tau)$
  **return** $T$

**Interface:** $\text{fDuplex}(s, \sigma)$, $\sigma \in \mathbb{F}_2^t$, $1 \leq t \leq 129$, $s \in \mathbb{F}_2^{512}$
  $s_{0:383} \leftarrow s_{0:383} + \sigma||1||0^{129-t}$
  $s_{384:511} \leftarrow s_{384:511} + \sigma||1||0^{129-t}$
  $s \leftarrow f(s)$
  **return** $s$

and a capacity $c = b - r = 254$. The key length is $k \geq 160$ and tag length $\tau = 128$.

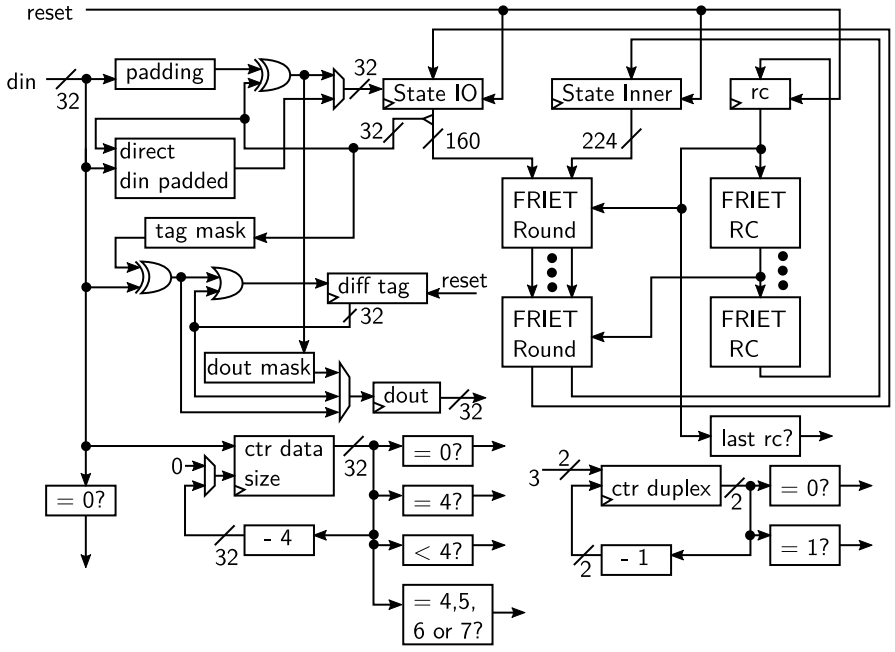
The encryption of a message simply consists of splitting $A$ and $P$ into blocks, padding each block, adding it to the state $s$, and performing the permutation $f_C$. Concurrently, each plaintext block is encrypted by bitwise addition to the outer part of the state. Finally, SpongeWrap squeezes the tag $T$ from the state with a (number of) duplex call(s). Decryption is a similar process to encryption. After a message has been encrypted or decrypted, one can continue the session with more messages.

The state is initialized by absorbing first the key $K$ and then the diversifier $D$. For confidentiality, the couple $(K, D)$ must be unique per session. If $K$ and $D$ are repeated, then it is not possible to guarantee the privacy of the ciphertext that is output, since if different plaintexts are encrypted with the same associated data, then it is possible to obtain the bitwise difference of the plaintexts by computing the bitwise difference of the ciphertexts [21].

Because it uses the duplex construction, SpongeWrap lends itself quite well to the use of a code-embedded permutation $f_C$ (FRIET-PC). We just have to instantiate the duplex with the code-abiding permutation $f$ (FRIET-P) and make some minor modifications as shown in Algorithm 28. The changes in more detail are:

- The state initialization must set the state to the codeword that encodes the all-0 vector. For linear codes, this is just the all-0 vector.
- When absorbing $\sigma$, it must first be converted to a valid codeword. If $\sigma$ is one limb (as it turns out in FRIET-PC), it suffices to (bitwise) add it to one limb and the parity limbs that depend on it.
- Before using the outer part of the state as a tag or keystream, one must check whether the state is a valid codeword and return an error if not.

Figure 5.10: Hardware architecture for FRIET.




## Hardware design

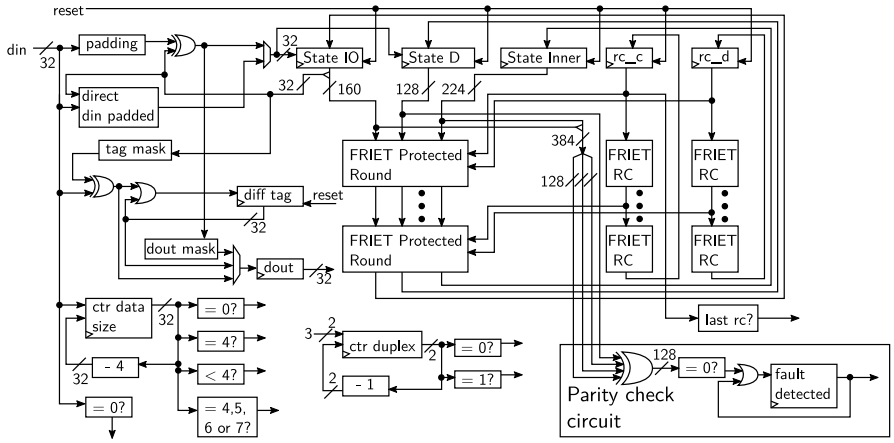
There are two hardware designs for FRIET, one made for the original publication [128] with a custom interface and another with the NIST lightweight competition (LWC) interface [67] in Section 5.4. The FRIET interface was updated for the LWC interface, thus it is possible to directly compare it with the other proposals, such as Gimli and Subterranean 2.0.

### Original interface

FRIET and FRIET-C were implemented in two versions, one with 1 round per clock cycle (1R), and another with 2 rounds per clock cycle (2R). Figures 5.11 and 5.10 show the full architecture of FRIET and FRIET-C, respectively. The two architectures are very similar, the differences are the use of the appropriate permutation (FRIET-P or FRIET-PC), the state register $d$ that holds the parity, and a parallel circuit that verifies if the content on all state registers is valid.

The FRIET circuit in Figure 5.11 has 5 registers: State_IO, State_D, State_Inner, rc_c, and rc_d. The State_IO register holds the outer part of the state, State_Inner the inner part. The State_IO register is a circular shift register that loads 32 bits at every clock cycle and has a size of 160 bits. The sponge rate is 130 bits instead of 160, and thus the remaining 30 bits in the State_IO register belong to the inner part that is supposedly in State_Inner. The State_D holds the parity limb. The rc_c and rc_d registers hold the round constants. The FRIET-C circuit differs from that of FRIET by the absence of registers State_D and rc_d.

Figure 5.11: Hardware architecture for FRIET.




The circuit communicates through a single 32-bit bus via a 3-field protocol: the command (4 bytes) encoding one of {reset, duplex-none, duplex-encrypt, duplex-decrypt, tag generate, tag verify}, the data length (4 bytes) and the data itself (variable). After receiving a command and data length, it takes 4 cycles to feed 16 bytes into the State_IO register. Then, it performs the FRIET-P permutation, during which the circuit does not acknowledge the data in the "din" port. This takes 24 cycles in the 1R version and 12 in the 2R version.

There a few aspects related to the parity check part of the FRIET circuit. When the circuit starts or receives a reset command, all state registers are reset with zeroes, thus satisfying the parity check. If the circuit receives data through the "din", then the new data is fed into State_IO and State_D simultaneously, keeping the parity unchanged. A dedicated circuit does a parity check at every clock cycle for detecting faults, and in the case of a detection, it sets a register "fault detected" to 1. It is assumed the FRIET circuit is used alongside another circuit that monitors the value of the "fault detected" register and performs the appropriate action. Finally, when designing the circuit fault redundancy part synthesis flags have been added, to tell the FPGA/ASIC tools to not remove them during the optimization steps.

Table 5.3 shows the FPGA and ASIC results for FRIET after place and route from Simon et al. [128]. The results for FRIET are compared with Ketje-Sr from Guido Bertoni's GitHub repository [19].

## 5.3 Subterranean 2.0

We specify the Subterranean 2.0 suite in a bottom-up fashion, starting with the round function, then input injection and output extraction, then the Subterranean 2.0 duplex object, then the XOF function, then the deck function, then the SAE scheme, and, finally, the parameters used in Subterranean 2.0.

Table 5.3: Xilinx Virtex-7 xc7vx485tffg1761-3 and ASIC Nangate 45 nm standard cell results for Ketje Sr., FRIET, FRIET-C.

| | FPGA | | | | |
|---|---|---|---|---|---|
| | Resources | | | Freq. | Throu. |
| AE Scheme | Slice | LUT | FF | (MHz) | (Mb/s) |
| Ketje-Sr[19] | 452 | 1680 | 448 | 282 | 9037 |
| FRIET (1R) | 450 | 1653 | 628 | 399 | 1828 |
| FRIET-C (1R) | 251 | 905 | 494 | 410 | 1874 |
| FRIET (2R) | 601 | 2258 | 628 | 363 | 2909 |
| FRIET-C (2R) | 385 | 1401 | 493 | 391 | 3135 |
| | ASIC | | | | |
| | Area | Freq. | Throu. | Power ($\mu$W) | |
| AE Scheme | (GE) | (MHz) | (Mb/s) | static | dynamic |
| Ketje-Sr[19] | 9478 | 503 | 16096 | 161 | 2152 |
| FRIET (1R) | 9253 | 508 | 2322 | 148 | 2226 |
| FRIET-C (1R) | 6943 | 508 | 2322 | 110 | 1724 |
| FRIET (2R) | 11100 | 508 | 4064 | 174 | 2245 |
| FRIET-C (2R) | 8890 | 508 | 4064 | 141 | 1737 |

### The round function RS, input injection, and output extraction

The round function RS operates on a 257-bit state and has four steps:

$$\mathrm{RS} = \pi \circ \theta \circ \iota \circ \chi \ . \tag{5.1}$$

Each step of the round function has a particular purpose: $\chi$ for non-linearity, $\iota$ for asymmetry, $\theta$ for mixing, and $\pi$ for dispersion.

We denote the state as $s$ and its bits as $s_i$ with position index $i$ ranging from 0 to 256, where any expression in the index must be taken modulo 257. For all $0 \leq i < 257$:

$$\begin{aligned} \chi: &\quad s_i \leftarrow s_i + (s_{i+1} + 1)s_{i+2} \ , \\ \iota: &\quad s_i \leftarrow s_i + \delta_i \ , \\ \theta: &\quad s_i \leftarrow s_i + s_{i+3} + s_{i+8} \ , \\ \pi: &\quad s_i \leftarrow s_{12i} \ . \end{aligned}$$

Here the addition and multiplication of state bits are in $\mathbb{F}_2$, and $\delta_i$ is a Kronecker delta: $\delta_i = 1$ if $i = 0$ and 0 otherwise. Figure 5.12 illustrates the round function by the computational graph of a single bit of the state.

At the core of the Subterranean duplex object is a simple (internal) *duplex call* that first applies the Subterranean round function RS to the state and then injects a string $\sigma$ of variable length of at most 32 bits. Before adding it into the state, it pads the string $\sigma$ to 33 bits with simple padding ($10^*$), and hence the absorbing rate is 33 bits. In between duplex calls, one may extract 32-bit strings $z$ from the state, so the squeezing rate is 32 bits.

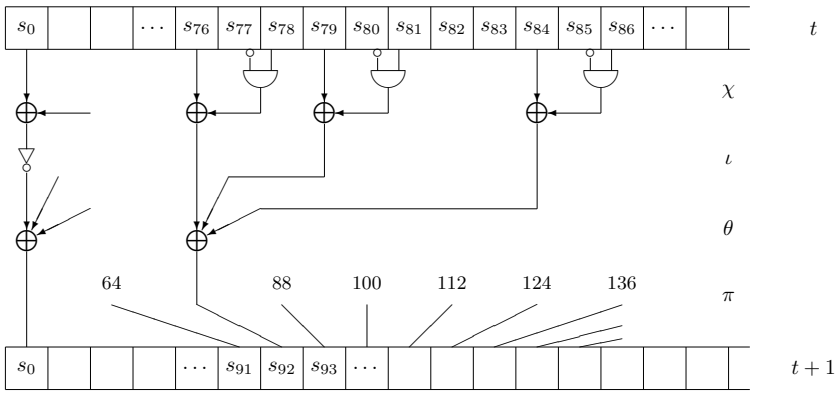



Figure 5.12: Subterranean round function, illustrated for bit $s_{92}$

The 33 bits of $\sigma$ after padding are injected into the state at positions that form the first 33 powers of $12^4$. For the unkeyed mode (hashing), the input $\sigma$ is limited to 8 bits bringing the effective absorbing rate to 9 bits due to the padding.

### The Subterranean duplex object

The Subterranean duplex object has at its core two internal functions: the duplex call and the output extraction. The duplex call applies the round function and injects the input together with the output extraction specified in the previous section.

On top of the duplex and extraction calls it has a thin wrapper consisting of three functions that facilitate the compact specification of cryptographic functions and schemes.

The wrapper supports absorbing and squeezing strings of arbitrary length and the integration of encryption and decryption with absorbing. It provides separators between absorbed strings by internally ensuring its last injected block is shorter (possibly empty) than 32 bits in keyed mode and 8 bits in unkeyed mode. When absorbing into a *secret* state (sAbsorbKeyed, sAbsorbEncrypt, sAbsorbDecrypt in Algorithm 30), Subterranean injects up to 32 bits per duplex call. In unkeyed absorbing (sAbsorbUnkeyed), it injects at most 8 bits per duplex call and applies duplex calls with no injected data in between.

We specify the Subterranean duplex object in Algorithms 29 and 30. There, any input or output is a bit string of arbitrary length, unless specified otherwise.

### The Subterranean-XOF function

We specify Subterranean-XOF in Algorithm 31. It is meant to be used for unkeyed hashing and takes as input a sequence of an arbitrary number ($n$) of arbitrary-length strings $M$ and returns a bit string of arbitrary length. Subterranean-XOF absorbs 8 bit block of data per permutation call.

**Algorithm 29** Subterranean duplex object procedures part 1

**Interface:** Constructor: $s \leftarrow \text{Subterranean}()$, $s \in \mathbb{F}_2^{257}$
  $s \leftarrow 0^{257}$
  **return** $s$

**Interface:** $\text{sBlank}(s, r)$, $r \in \mathbb{Z}^+$, $s \in \mathbb{F}_2^{257}$
  **for** $i \leftarrow 0$ **to** $r - 1$ **by** 1 **do**
    $s \leftarrow \text{sDuplex}(s, \epsilon)$
  **end for**

**Interface:** $s, Z \leftarrow \text{sSqueeze}(s, \ell)$, $Z \in \mathbb{F}_2^{\ell}$, $s \in \mathbb{F}_2^{257}$
  $Z \leftarrow \epsilon$
  **for** $i \leftarrow 0$ **to** $\ell - 33$ **by** 32 **do**
    $Z_{i:i+31} \leftarrow \text{sExtract}(s, 32)$
    $s \leftarrow \text{sDuplex}(s, \epsilon)$
  **end for**
  $Z_{i:\ell-1} \leftarrow \text{sExtract}(s, \ell - i)$
  $s \leftarrow \text{sDuplex}(s, \epsilon)$
  **return** $s, Z$

**Internal interface:** $s \leftarrow \text{sDuplex}(s, \sigma)$, $\sigma \in \mathbb{F}_2^{t}$, $0 \leq t \leq 32$, $s \in \mathbb{F}_2^{257}$
  $s \leftarrow \text{RS}(s)$
  $x \leftarrow \sigma || 1 || 0^{32-t}$
  **for** $j \leftarrow 0$ **to** 32 **by** 1 **do**
    $s_{12^{4j}} \leftarrow s_{12^{4j}} + x_j$
  **end for**
  **return** $s$

**Internal interface:** $z \leftarrow \text{sExtract}(s, \ell)$, $Z \in \mathbb{F}_2^{\ell}$, $1 \leq \ell \leq 32$, $s \in \mathbb{F}_2^{257}$
  $z \leftarrow \epsilon$
  **for** $j \leftarrow 0$ **to** $\ell - 1$ **by** 1 **do**
    $z \leftarrow z || (s_{12^{4j}} + s_{-12^{4j}})$
  **end for**
  **return** $Z$

**Algorithm 30** Subterranean duplex object procedures part 2

**Interface:** $s \leftarrow \text{sAbsorbUnkeyed}(s, X)$, $X \in \mathbb{F}_2^{xl}$, $s \in \mathbb{F}_2^{257}$
  **for** $i \leftarrow 0$ **to** $xl - 8$ **by** 8 **do**
    $s \leftarrow \text{sDuplex}(s, X_{i:i+7})$
    $s \leftarrow \text{sDuplex}(s, \epsilon)$
  **end for**
  **if** $i > xl - 1$ **then**
    $s \leftarrow \text{sDuplex}(s, X_{i:xl-1})$
  **else**
    $s \leftarrow \text{sDuplex}(s, \epsilon)$
  **end if**
  $s \leftarrow \text{sDuplex}(s, \epsilon)$
  **return** $s$

**Interface:** $s \leftarrow \text{sAbsorbKeyed}(s, X)$, $X \in \mathbb{F}_2^{xl}$, $s \in \mathbb{F}_2^{257}$
  **for** $i \leftarrow 0$ **to** $xl - 32$ **by** 32 **do**
    $s \leftarrow \text{sDuplex}(s, X_{i:i+31})$
  **end for**
  **if** $i > xl - 1$ **then**
    $s \leftarrow \text{sDuplex}(s, X_{i:xl-1})$
  **else**
    $s \leftarrow \text{sDuplex}(s, \epsilon)$
  **end if**
  **return** $s$

**Interface:** $s, Y \leftarrow \text{sAbsorbEncrypt}(s, X)$, $X, Y \in \mathbb{F}_2^{xl}$, $s \in \mathbb{F}_2^{257}$
  **for** $i \leftarrow 0$ **to** $xl - 33$ **by** 32 **do**
    $Y_{i:i+31} \leftarrow X_{i:i+31} + \text{sExtract}(s, 32)$
    $s \leftarrow \text{duplex}(s, X_{i:i+31})$
  **end for**
  $Y_{i:xl-1} \leftarrow X_{i:xl-1} + \text{sExtract}(s, xl - i)$
  $s \leftarrow \text{duplex}(s, X_{i:xl-1})$
  **return** $s, Y$

**Interface:** $s, Y \leftarrow \text{sAbsorbDecrypt}(s, X)$, $X, Y \in \mathbb{F}_2^{xl}$, $s \in \mathbb{F}_2^{257}$
  **for** $i \leftarrow 0$ **to** $xl - 33$ **by** 32 **do**
    $Y_{i:i+31} \leftarrow X_{i:i+31} + \text{sExtract}(s, 32)$
    $s \leftarrow \text{duplex}(s, Y_{i:i+31})$
  **end for**
  $Y_{i:xl-1} \leftarrow X_{i:xl-1} + \text{sExtract}(s, xl - i)$
  $s \leftarrow \text{duplex}(s, Y_{i:xl-1})$
  **return** $s, Y$

The hash function of the Subterranean 2.0 suite submitted to the NIST lightweight competition is Subterranean-XOF, with the input restricted to a string sequence of a single arbitrary-length byte string and the output length fixed to 256 bits.

**Algorithm 31** Subterranean-XOF

**Interface:** $Z \leftarrow$ Subterranean-XOF$(M, \ell)$ with $M_i \in \mathbb{F}_2^*$, where $i \in [0, n]$. $M$ is a collection of $n$ strings and $\ell \in \mathbb{Z}^+$
  $s \leftarrow$ Subterranean()
  **for** $i \leftarrow 0$ **to** $n - 1$ **by** 1 **do**
    $s \leftarrow$ sAbsorbUnkeyed$(s, M_i)$
  **end for**
  $s \leftarrow$ sBlank$(s, 8)$
  $s, Z \leftarrow$ sSqueeze$(s, \ell)$
  **return** $Z$

### The Subterranean-deck function

We specify Subterranean-deck in Algorithm 32. It takes as input an arbitrary-length key $K$ and a sequence of an arbitrary number ($n$) of arbitrary-length strings $M$ and returns a bit string of arbitrary length. It can readily be used as a stream cipher, a MAC function and for key derivation. The Farfalle paper [20] and the Xoodoo cookbook [40] specify several authenticated encryption modes for deck functions. Subterranean-deck absorbs the key and message in blocks of 32 bits.

**Algorithm 32** Subterranean-deck

**Interface:** $Z \leftarrow$ Subterranean-deck$(K, M, \ell)$ with $M_i \in \mathbb{F}_2^*$, where $i \in [0, n]$. $M$ is a collection of $n$ strings and $\ell \in \mathbb{Z}^+$
  $s \leftarrow$ Subterranean()
  $s \leftarrow$ sAbsorbKeyed$(s, K)$
  **for** $i \leftarrow 0$ **to** $n - 1$ **by** 1 **do**
    $s \leftarrow$ sAbsorbKeyed$(s, M_i)$
  **end for**
  $s \leftarrow$ sBlank$(s, 8)$
  $s, Z \leftarrow$ sSqueeze$(s, \ell)$
  **return** $Z$

### The Subterranean-SAE authenticated encryption scheme

We specify Subterranean-SAE in Algorithm 33. It takes a nonce when starting the session and can then encipher and authenticate a sequence of messages each consisting of plaintext and associated data. Compared to authenticated encryption modes based on Subterranean-deck, Subterranean-SAE has a smaller state and is better suited to offer protection against differential power analysis (DPA). In particular, the security is based on the secrecy of a state that evolves during the session rather than a static key. Across sessions, one can derive a fresh key per session using Subterranean-deck.

This protects against differential power analysis and provides fine-grained forward secrecy. If one wishes to use the same key for multiple sessions and DPA is a concern, one can absorb the nonce bit per bit, as was proposed by Taha and Schaumont [129].

The authenticated encryption (AE) scheme of the Subterranean 2.0 suite submitted to the NIST lightweight competition is Subterranean-SAE, with a key $K$ of length 128 bits, a nonce $N$ of length 128 bits, and a tag length $\tau = 128$ and associated data and plaintext limited to byte strings.

**Algorithm 33** Subterranean-SAE, with $\tau$ the tag length

**Interface:** $s \leftarrow \text{sStart}(K, N)$
  $s \leftarrow \text{Subterranean}()$
  $s \leftarrow \text{sAbsorbKeyed}(K)$
  $s \leftarrow \text{sAbsorbKeyed}(N)$
  $s \leftarrow \text{sBlank}(s, 8)$
  **return** $s$

**Interface:** $s, C, T \leftarrow \text{sWrap}(s, A, P)$
  $s \leftarrow \text{sAbsorbKeyed}(s, A)$
  $s, C \leftarrow \text{absorbEncrypt}(s, P)$
  $s \leftarrow \text{sBlank}(s, 8)$
  $s, T \leftarrow \text{sSqueeze}(s, \tau)$
  **return** $s, C, T$

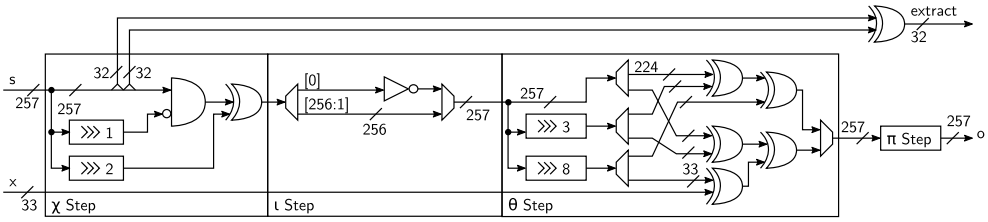
**Interface:** $s, P \leftarrow \text{sUnwrap}(s, A, C, T')$
  $s \leftarrow \text{sAbsorbKeyed}(s, A)$
  $s, P \leftarrow \text{absorbDecrypt}(s, C)$
  $s \leftarrow \text{sBlank}(s, 8)$
  $s, T \leftarrow \text{sSqueeze}(s, \tau)$
  **if** $(T' \neq T)$ **then**
    $s, P \leftarrow \epsilon, \epsilon$
  **end if**
  **return** $s, P$

## 5.4 NIST lightweight implementations

To compare all three proposals, Gimli, Friet and Subterranean 2.0, we implement all of them in hardware with the same lightweight interface in the NIST lightweight competition (LWC) [61]. Because the internal interface and components are very similar, we will begin explaining the implementation of Subterranean 2.0 and later the differences for Gimli and Friet. The explanation will use a bottom-up approach, thus it will start from the most basic circuits and then growing in complexity and functionality.

The Subterranean 2.0 cipher suite and Friet have a bit-wise specification, thus they can work with inputs in any bit size. However the Gimli specification and the

Figure 5.13: Hardware architecture for Subterranean 2.0 round function *RS*.




LWC interface only accept byte-size inputs, and therefore all circuits here assume byte-size inputs.

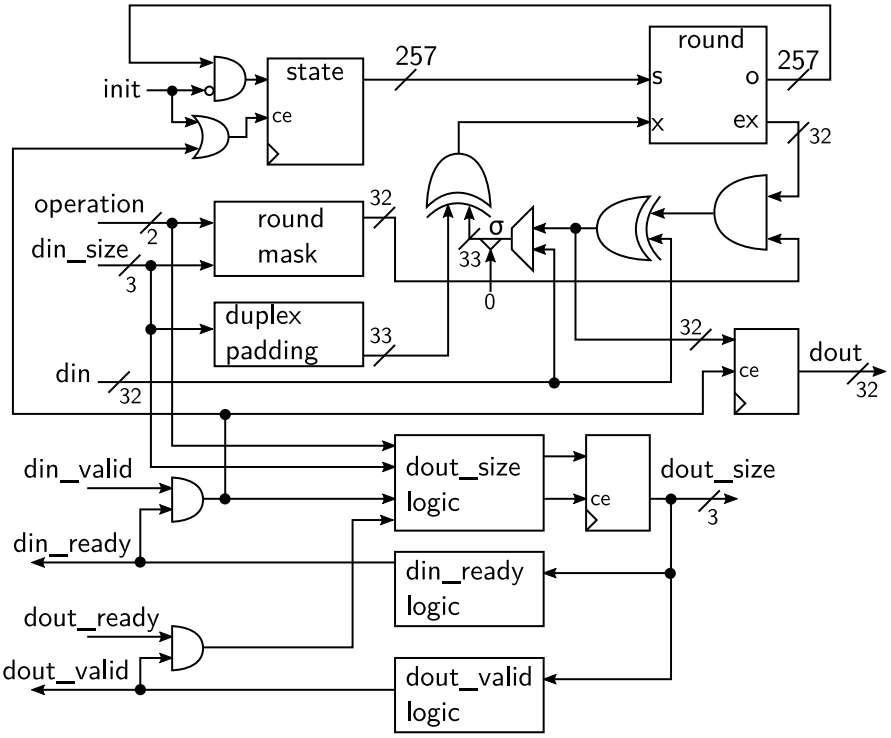
Figure 5.13 shows the Subterranean 2.0 round function *RS* circuit. The circuit is almost a direct copy of the Subterranean 2.0 round function specification Figure 5.12, except for the addition of the injection and extraction circuits. The injection of the bits could be done outside of the round function, however, it is possible to parallelize the injection in the $\theta$ step, which has a lower circuit depth, thus higher operating frequency, than performing consecutively outside. The extraction of the bits is done by executing an XOR between two inputs of 32 bits.

Figure 5.14 shows the Subterranean 2.0 duplex circuit. The duplex circuit stores and updates the internal Subterranean state according to the input operations without exposing the full state. The possible input operations are: init, duplex with/without output and duplex encrypt/decrypt. *Init* writes the value zero into all state bits, thus clearing the whole state. The circuit can duplex data inputs as specified in the *sDuplex* procedure by adding the data input after processing the round function, which in this case happens all in one clock cycle. The data input is also padded in the same cycle as the round function processing, and the padding is generated according to the size of the data input given by the "din_size" port. One special case for the duplex call is the *sSqueeze* procedure, where an empty block is given and an output is generated. In that case, then it is possible to send a command to the circuit to execute the duplex and extract 32 bits of data in one clock cycle, thus merging the *sDuplex* and *sExtract* functionality. As with the *sSqueeze* procedure, the duplex in *sEncrypt* and *sDecrypt* also requires some extra functionality to encrypt/decrypt data, thus a separate duplex command was created for them, joining the original *sDuplex* with parts of the *sAbsorbEncrypt* and *sAbsorbDecrypt*.

The Subterranean 2.0 duplex circuit in Figure 5.14 and the other high-level circuits in Figures 5.15 and 5.16 employ a valid/ready interface handshake. This makes sure the data is only transferred between different circuit parts when the sender makes a valid high and the receiver makes the ready high as well. By having the two parties synchronized, it is possible to have one side taking more time to process, send or receive the data and have the entire system synchronized. The entire circuit works on the same clock, therefore it is possible to perform the entire handshake in one state/cycle.

Subterranean stream, in Figure 5.15, is a circuit that can perform Subterranean 2.0-XOF and SAE with a stream of data as input. The circuit is composed of the Subterranean duplex circuit from Figure 5.14, one buffer for the input data, a comparison circuit, one counter, two logic flow controllers, and one state machine that controls

Figure 5.14: Hardware architecture for Subterranean 2.0 duplex object.

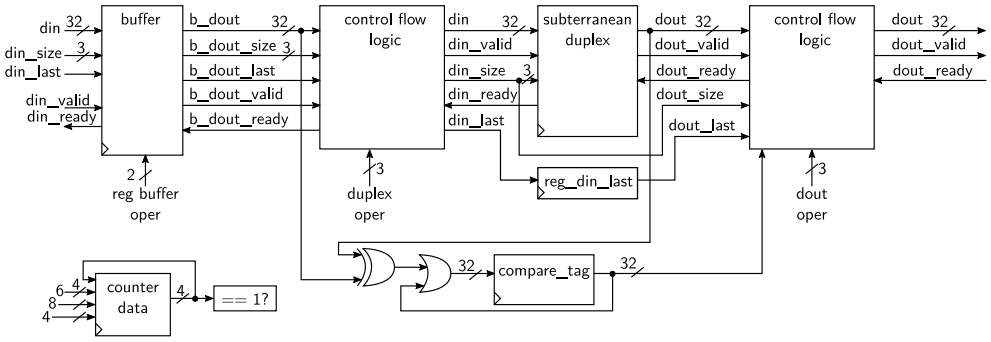



everything. The input buffer receives 32 bits blocks of data and outputs 32 or 8 bits blocks of data, where the 8 bit output is used during XOF message absorption and the 32 bits for all the other operations. The comparison circuit is only used during the tag verification, where it will compare the output of the Subterranean duplex circuit with the received value, and it will accumulate the difference in its internal register. The counter is used during the blank rounds to know if all eight rounds have been performed if the entire tag has been generated during encryption and if all the XOF has been output. The two logic flow controllers act as a glue logic between the components that are connected. This glue logic is necessary when some components require some forced constants, or different parts to be unconnected. The state machine controls all this circuitry according to one of the following external commands: loading a key, loading a message and generating a hash, performing encryption and generating a tag, and performing decryption and verifying a tag.

When loading a key, the stream circuit sends a command to clean the state in the duplex circuit, and prepares the input buffer to receive 32 bits of data and output 32 bits of data per clock cycle. The state machine also prepares the duplex circuit to absorb the data that comes from the input buffer uninterrupted and it continues until it receives the last data block. The state machine stays idle waiting to receive a command for encryption or decryption.

Encryption and decryption start in the same manner until the last block of ci-

Figure 5.15: Hardware architecture for Subterranean 2.0 that operates on a stream of data.

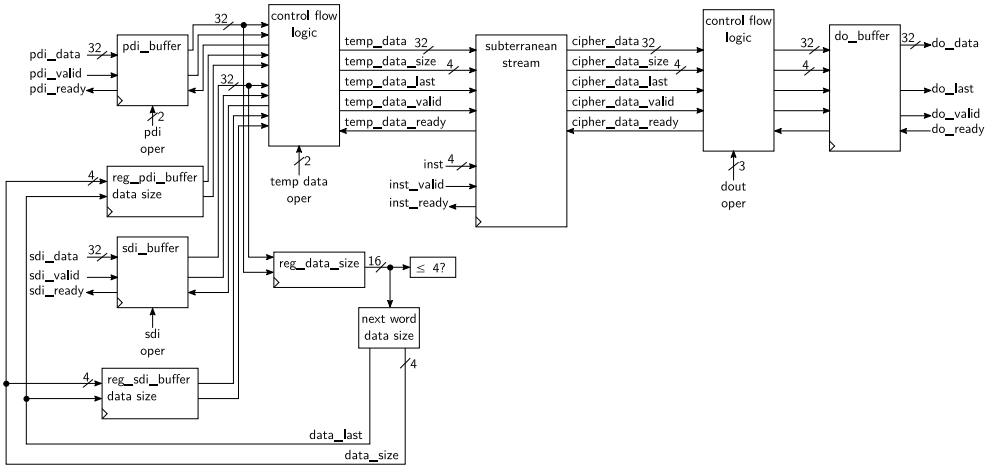



phertext/plaintext is generated, since for encryption the next step is to generate the session tag, and for decryption it is to verify the session tag. Assuming the key has already been loaded, the circuit then loads the nonce in the same way as the key, then it blocks the input and performs the necessary blank rounds, and then it loads the associated data just as the key. After loading the associated data, it loads the plaintext/ciphertext by also setting the input buffer to output 32 bit data blocks and the duplex circuit into encryption/decryption mode. In the case of encryption, after finishing processing the last plaintext, the state machine performs the blank rounds followed by the tag generation, where it will generate tag blocks by forcing a zero value into the duplex. Decryption is more complex because after generating the last plaintext and performing the blank rounds, the state machine has to enable the comparison tag circuit, connect the output of the input buffer and the output of the duplex circuit in the comparison tag circuit. After the comparison tag is enabled and the inputs connected, it keeps comparing the outputs until the last tag is received, then the state machine will use the comparison output to flag if the verification was valid or not.

When the state machine receives the hash command, the stream circuit sends a command to clean the state in the duplex circuit and prepares the input buffer to receive 32 bits of data and output 8 bits of data per clock cycle. With the input buffer ready to serialize data as 8 bit data blocks, the state machine interposes the input of the duplex circuit between the input buffer and an empty block, since, after every 8 bit data block, an empty duplex happens. This continues until the last data block arrives, then the state machine performs the necessary 8 blank rounds, and then prepares the duplex to receive empty blocks while it outputs the message hash.

The Subterranean compatible with the LWC interface [67] is shown in Figure 5.16. The circuit is composed of two input buffers, two control flow logic, one output buffer, and a state machine that controls the entire circuit. The LWC interface has two input buses, one for public data and another for secret data, and one common output bus [67]. While it is possible to merge the two input buses into one buffer, it is easier to control if each bus has it is own buffer, so then the buffer can store and hold messages while it is not being used. The glue logic is controlled by the state machine to control the connection of the two buffers and the stream circuit. The output buffer

Figure 5.16: Hardware architecture for Subterranean 2.0 that is compatible with the LWC interface.



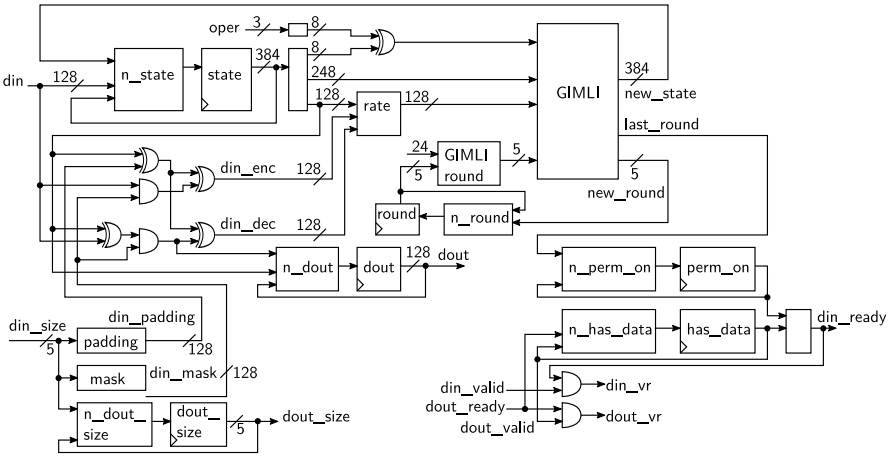

is also connected through a glue logic to the subterranean stream, which in this case is mostly applied to send some required control messages. The LWC interface also has a protocol, and therefore some control messages are necessary, and thus the glue logic is used to send such messages.

Because the entire circuit can operate in a streaming fashion, it can operate at the maximum throughput of the underlying permutation, thus in the case of Subterranean 2.0 it is 32 bits per cycle during encryption/decryption, and 8 bits per cycle for processing messages during hashing or XOF.

The Gimli hardware architecture compatible with the LWC interface has the same structure as the Subterranean 2.0, thus a duplex circuit, a stream circuit that can perform AE and hash, and a circuit compatible with the LWC interface. The duplex circuit of Gimli is shown in Figure 5.17. The circuit is based on the Subterranean 2.0 duplex circuit in Figure 5.14, as both separate the output from the permutation computation and generate padding and masking data inputs. However, the Gimli circuit handles a permutation that takes multiple cycles to finish (24 to 2 cycles), input/output values of 128 bits, and minor details of Gimli, such as flipping one bit in the state at the last absorbing.

The Gimli duplex circuit in Figure 5.17 has the following operations: absorb directly, absorb encryption, absorb decryption, squeeze, squeeze and permute, and initialize the first, second and third column. Absorb directly can absorb 128 bits with no output generated, just as *gAbsorbRate* in Algorithm 19. Absorb encryption is exactly as absorb directly, but in this case, the ciphertext is generated, and absorb decryption generates the plaintext and absorbs the plaintext. The squeeze extracts 128 bits from the internal state, just as *gExtract* in Algorithm 19, and the squeeze and permute also extracts 128 bits from the state, followed by a 24 round permutation. There are two different squeezes, because in the hash output generation, first we extract 16 bytes, followed by the 24 round permutation and then we extract another 16 bytes. While this could be solved by having one squeeze call and another for

Figure 5.17: Hardware architecture for GIMLI duplex object.




permute, it is faster to aggregate them into one squeeze and permute. The three columns initialization commands can load each of the three columns of 128 bits, which is used to load the key and nonce at the start of the AE encryption and decryption.

The GIMLI streaming and LWC circuits are very similar to the Subterranean 2.0 streaming and LWC ones in Figures 5.15 and 5.16. The differences are related to some nuances in the system's high-level calls and the operating size that is 128 bits for the duplex circuit. Even though the duplex circuit is 128 bits, the streaming and LWC of GIMLI is still 32 bits, thus the streaming buffers receive and send in 32 bits to the LWC circuit, while internally they operate at 128 bits with the duplex circuit.

FRIET is also similar to the Subterranean 2.0 circuit, since they have similar high level sponge calls, and to the GIMLI circuit since they both operate with a rate of around 128 bits. The only difference in the FRIET circuit is the fault detection signal. The fault detection signal is output by the streaming circuit and then handled by the LWC interface with a final status message. Therefore if there was a fault detected during the SAE, then the final message of the LWC interface will indicate there was a problem during encryption/decryption.

Table 5.4 shows the results after synthesizing the three proposals Subterranean 2.0, FRIET, FRIET-C, and GIMLI in the Xilinx Artix-7. All three are only compared for authenticated encryption since FRIET and FRIET-C do not specify a hash or XOF. The results show that Subterranean 2.0 is a very compact and high throughput proposal, while GIMLI, FRIET, and FRIET-C have very similar throughput for their one round unrolled version. Between GIMLI, FRIET-C, and FRIET the first two have very close costs in terms of slices, around 19% difference, but FRIET is 87% more expensive than FRIET-C while providing fault detection capabilities which is not on the other implementations.

GIMLI and Subterranean 2.0 are in the second round of the NIST lightweight competition and they have been evaluated with other candidates in FPGAs by Mohajerani et al. [96] and in ASIC by Khairallah et al. [70] and Aagaard and Zidaric [1]. In the evaluation of Mohajerani et al. [96], they not only evaluated the versions of this the-

Table 5.4: Xilinx Artix-7 xc7a12tcsg325-3 results for AE with Subterranean 2.0, FRIET, FRIET-C, GIMLI. The ($x$R) means the number unrolled rounds in the circuit.

| | Resources | | | Freq. | Throu. |
|---|---|---|---|---|---|
| AE Scheme | Slice | LUT | FF | (MHz) | (Mb/s) |
| Subterranean 2.0 | 241 | 885 | 606 | 176 | 5632 |
| GIMLI (1R) | 480 | 1648 | 1164 | 165 | 880 |
| GIMLI (2R) | 539 | 1867 | 1163 | 162 | 1728 |
| GIMLI (4R) | 687 | 2457 | 1166 | 136 | 2901 |
| FRIET (1R) | 755 | 2724 | 1285 | 170 | 906 |
| FRIET (2R) | 1001 | 3600 | 1285 | 124 | 1322 |
| FRIET-C (1R) | 404 | 1364 | 1147 | 178 | 949 |
| FRIET-C (2R) | 471 | 1629 | 1145 | 145 | 1546 |

sis but made faster implementations of GIMLI and Subterranean 2.0 in Bluespec [95]. In their results [96], the Subterranean 2.0 in Bluespec is the first for authenticated encryption with 848 LUTs and can process plaintext at 9.5 Gbits/s, while the version in this thesis requires 891 LUTs and can process only 6.0 Gbits/s, but it can also hash at 760 Mbits/s. The GIMLI in Bluespec is the first in hashing with 2357 LUTs and it can process 4.4Gbits/s of message in hashing or plaintext for authenticated encryption, while this thesis with 2510 LUTs can only handle 3.0 Gbits/s of messages in hashing or plaintext for authenticated encryption. In terms of ASIC, the results of Khairallah et al. [70] show that Subterranean 2.0 is the first in terms of energy and Gimli the third. More specifically, Subterranean 2.0 authenticated encryption for messages with 64 bytes in TSMC 65nm requires 17.74 pJ and 6419 GE, while GIMLI requires 61.50 pJ and 16620 GE. The results of Aagaard and Zidaric [1] also show that Subterranean 2.0 is the best in terms of energy, and when this thesis version is compared with the one written in Bluespec the results are very close. According to Aagaard and Zidaric [1], GIMLI is also one of the most energy-efficient together with Xoodyak and Ascon.

## 5.5 Final considerations of the chapter

This chapter presents three proposals called GIMLI, FRIET, and Subterranean 2.0 focusing on their hardware implementations and giving a comparison between all three of them. Each proposal has it is own advantages and disadvantages. GIMLI is an all-round solution with good results in different scenarios (software and hardware) with the best hardware results for hashing in the NIST lightweight competition. FRIET is an optimized solution for fault detection capabilities, that may not have the best results when compared to other proposals, however in applications that require fault detection it is cheaper. Subterranean 2.0 is aimed for hardware deployment and therefore is heavily optimized for these environments, which is shown by being one of the most energy-efficient AEs in the NIST lightweight competition.

# Discussion & Conclusions

Chapter 2 provided results and evaluates two Montgomery multiplication designs based on two word-based variants of the Montgomery multiplication algorithm. The two designs when compared to other works aiming at low resources achieved similar results. Between the two units, one was more conservative and required only one multiplier, but the second one had better performance results at the cost of two multipliers.

Chapter's 2 research could continue as: applying similar techniques to more modern FPGA multiplier, changing the memory block for revolving shift registers and using extra units to parallelize the multiplication. Chapter 2's work was tuned for the IGLOO2 FPGA multiplier unit, so when ported directly to Xilinx FPGAs the results were not that good, thus applying similar techniques and optimizing for the latest Xilinx or other manufacturers FPGAs might be interesting. The two presented designs could achieve a very high frequency, but the timing bottleneck was not the multiplier, but instead the memory block. In order to achieve even higher frequencies, the design could employ a circular shift register that could rotate the values of the operands. Shift registers would require more resources in the FPGA, but they would operate at higher frequency and provide better timings. Another future work idea would be to exploit more parallelization and multipliers, thus trying instead to use four or even more multipliers, but something less than the multiplier described in Chapter 4.

Chapter 3 provides a proof-of-concept hardware architecture for the complete formulas from Renes et al. [111]. The hardware architecture employed three parallel multipliers from Chapter 2 and it had good competitive results when compared to other architectures with similar amounts of resources. The architecture has some room for improvements or even more research on this topic.

Improvements to the elliptic curve scalar multiplication architecture described in Chapter 3 include doing a six or twelve multipliers version, making the entire architecture easily configurable to different numbers of multipliers, from 1 to 12, and finally, trying to find formulas that have the lowest multiplication dependency. The formulas from Renes et al. [111] and Renes' thesis [110] can be parallelized up to 6 multipliers and the scalar multiplication algorithm can be parallelized into 2 separate additions, then it can be up to 12 multipliers. It is still an open question on how much performance does one gain with an architecture employing six small multipliers or twelve. In the same direction of changing the number of multipliers, how difficult would it be making an architecture that can easily handle different numbers of multipliers. Also, can the architecture handle a different number of multipliers

after it has been deployed? If the architecture is employed in ASIC, it could be that one of the multipliers has a manufacturing fault, therefore such multiplier could be turned off, but then the architecture should be able to handle this condition. Renes et al. [111] can offer a good parallelism for his formulas, but it is still an open question if there are better formulas for other curve shapes, like Edwards, Montgomery, or Hessian, that offer the lowest multiplication dependency.

Chapter 4 provides a competitive and small hardware/software co-design implementation of the cryptosystem SIKE. The proposed design was able to reduce the amount of DSPs used in the literature but is slower than previous results. Because the implementation is a hardware/software co-design, the entire unit can work with all parameters without having to be reloaded in the FPGA, thus offering more flexibility than previous works.

The SIKE project in Chapter 4 has lots of opportunities in optimizing the architecture and other research directions. In the SIKE project, optimizations in the `Carmela` circuit will bring greater amount of benefits, since it is one of the biggest bottlenecks in the entire design. `Carmela` frequency could be increased by employing a full carry-save notation in the accumulator and pipeline, at the cost of more logic and not being able to perform double writes at the end of the cycle. `Carmela` could also operate in a FIFO fashion, where several instructions are fed and then it can go from one instruction after another. If `Carmela` operates in a FIFO fashion, it will operate in parallel with the main CPU, which saves some cycles by having the CPU executing a few instructions while it waits for Carmela to finish. Also, the main CPU could be replaced with a RISC-V CPU or some other CPU with an instruction set that has support from compilers like GCC [130] and clang [131]. Finally, side-channel evaluation and adding countermeasures is also open for this design.

Chapter 5 encompasses 3 different symmetric cipher proposals: the permutation Gimli, the session authenticated encryption scheme Friet and the cipher suite Subterranean 2.0. The lightweight permutation Gimli when compared with other works in the literature showed to be not the best, but competitive in a wide array of platforms. When implemented as hash and AE in the FPGA, it shows good results for AE and the best results for hashing. Friet also shows competitive results for encryption when compared with Gimli, therefore the fault detection capabilities for this cipher is a big advantage. Subterranean 2.0 got the best results for encryption and area, thus it is a lightweight and high throughput encryption cipher. While it can perform hashing, it is not as fast as Gimli-24-Hash.

Possible future work related to Chapter 5 can be optimizing and improving the obtained results, evaluating more primitives, and studying the relationship between theoretical design and hardware results. The implementations in Chapter 5 with the lightweight NIST API does not have an optimal frequency result, when the round function is synthesized by itself the frequency is higher and thus it is possible to improve the circuit frequency or redesign the architecture around it. Instead of improving the results for the three ciphers, another path of future work is to increase the number of ciphers analyzed to already standardized ciphers such as SHA-2 or AES-GCM with a similar framework as used for the ones in Chapter 5. Finally, a more interesting work is to fill the gaps between theoretical cipher design and hardware results, for example, is it possible to have a good estimation of energy results for any cipher just by counting the number of Boolean operations? Given it is possible

to estimate energy results at a theoretical level, can we design more energy-efficient ciphers.

Other future work in hardware design and evaluations, would be to evaluate other public-key or symmetric key primitives, from the post-quantum [2] and lightweight [133] NIST competition and the newest NIST standard SP 800-208 [32]. The work could not only involve the design but also side-channel evaluation and countermeasure proposal and evaluation. In terms of side-channel attacks, one interesting aspect would be to evaluate a full implementation with the necessary countermeasures and the pseudo-random number generator, thus an almost ready production design. The same could be done for fault attacks, therefore an implementation that is complete in hardware and fully evaluated against some specific fault attack.

# Bibliography


[1] Mark D. Aagaard and Nusa Zidaric. *ASIC Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process*. Cryptology ePrint Archive, Report 2021/049. https://eprint.iacr.org/2021/049. 2021.

[2] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. https://doi.org/10.6028/NIST.IR.8309. 2020. DOI: 10.6028/NIST.IR.8309.

[3] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-Quantum Key Exchange: A New Hope". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 327–343. DOI: 10.5555/3241094.3241120. URL: https://dl.acm.org/doi/10.5555/3241094.3241120.

[4] Hamad Alrimeih and Daler Rakhmatov. "Fast and Flexible Hardware Support for ECC Over Multiple Standard Prime Fields". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 22.12 (Dec. 2014), pp. 2661–2674. DOI: 10.1109/TVLSI.2013.2294649. URL: http://dx.doi.org/10.1109/TVLSI.2013.2294649.

[5] D. Amiet, A. Curiger, and P. Zbinden. "Flexible FPGA-Based Architectures for Curve Point Multiplication over GF(p)". In: *2016 Euromicro Conference on Digital System Design (DSD)*. Aug. 2016, pp. 107–114. DOI: 10.1109/DSD.2016.70. URL: http://dx.doi.org/10.1109/DSD.2016.70.

[6] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zémor. *Bit Flipping Key Encapsulation – Submission to Round 3 of NIST's Post-Quantum Cryptography Standardization Process*. 2020. URL: https://bikesuite.org.

[7] Avnet. *Zedboard*. http://zedboard.org/product/zedboard. 2019.

[8] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. *Supersingular Isogeny Key Encapsulation – Submission to Round 2 of NIST's Post-Quantum Cryptography Standardization Process*. 2019. URL: https://sike.org.

[9] Steve Babbage, Daniel J. Bernstein, Alex Biryukov, Anne Canteaut, Carlos Cid, Joan Daemen, Orr Dunkelman, Henri Gilbert, Tetsu Iwata, Stefan Lucks, Willi Meier, Bart Preneel, Vincent Rijmen, Matt Robshaw, Phillip Rogaway, Greg Rose, Serge Vaudenay, and Hongjun Wu. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. https://competitions.cr.yp.to/caesar.html. 2014.

[10] Brian Baldwin, Richard Moloney, Andrew Byrne, Gary McGuire, and William P. Marnane. "A Hardware Analysis of Twisted Edwards Curves for an Elliptic Curve Cryptosystem". In: *Reconfigurable Computing: Architectures, Tools and Applications*. Vol. 5453. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 355–361. DOI: 10.1007/978-3-642-00641-8_41. URL: http://dx.doi.org/10.1007/978-3-642-00641-8_41.

[11] Elaine Barker. *Recommendation for Key Management : Part 1 - General*. Tech. rep. NIST, May 2020. DOI: 10.6028/NIST.SP.800-57pt1r5. URL: https://doi.org/10.6028/NIST.SP.800-57pt1r5.

[12] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". In: *Advances in Cryptology - CRYPTO' 86*. Vol. 263. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1987, pp. 311–323. DOI: 10.1007/3-540-47721-7_24. URL: http://dx.doi.org/10.1007/3-540-47721-7_24.

[13] Lejla Batina, Geeke Bruin-Muurling, and Sıddıka Berna Örs. "Flexible Hardware Design for RSA and Elliptic Curve Cryptosystems". In: *Topics in Cryptology – CT-RSA 2004*. Vol. 2964. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 250–263. DOI: 10.1007/978-3-540-24660-2_20. URL: http://dx.doi.org/10.1007/978-3-540-24660-2_20.

[14] Lejla Batina, Łukasz Chmielewski, Louiza Papachristodoulou, Peter Schwabe, and Michael Tunstall. "Online Template Attacks". In: *Progress in Cryptology – INDOCRYPT 2014: 15th International Conference on Cryptology in India, New Delhi, India, December 14-17, 2014, Proceedings*. Springer International Publishing, 2014, pp. 21–36. DOI: 10.1007/s13389-017-0171-8. URL: https://doi.org/10.1007/s13389-017-0171-8.

[15] Daniel J. Bernstein. "The Salsa20 Family of Stream Ciphers". In: *New Stream Cipher Designs: The eSTREAM Finalists*. Ed. by Matthew Robshaw and Olivier Billet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 84–97. DOI: 10.1007/978-3-540-68351-3_8. URL: https://doi.org/10.1007/978-3-540-68351-3_8.

[16] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. *Gimli*. NIST Lightweight Cryptography round 2 candidate. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/gimli-spec-round2.pdf.

[17] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. "Gimli : A Cross-Platform Permutation". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Cham: Springer International Publishing, 2017, pp. 299–320. DOI: 10.1007/978-3-319-66787-4_15. URL: https://dx.doi.org/10.1007/978-3-319-66787-4_15.

[18] Daniel J. Bernstein and Bo-Yin Yang. "Fast constant-time gcd computation and modular inversion". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.3 (May 2019), pp. 340–398. DOI: 10.13154/tches.v2019.i3.340-398. URL: https://tches.iacr.org/index.php/TCHES/article/view/8298.

[19] Guido Bertoni. *Ketje Keyak VHDL*. GitHub repository. https://github.com/guidobertoni/KetjeKeyakVHDL. 2017.

[20] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. "Farfalle: parallel permutation-based cryptography". In: *IACR Transactions on Symmetric Cryptology* 2017.4 (Dec. 2017), pp. 1–38. DOI: 10.13154/tosc.v2017.i4.1-38. URL: https://tosc.iacr.org/index.php/ToSC/article/view/855.

[21] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications". In: *Selected Areas in Cryptography*. Ed. by Ali Miri and Serge Vaudenay. Berlin, Heidelberg: Springer, 2012, pp. 320–337. DOI: 10.1007/978-3-642-28496-0_19. URL: http://dx.doi.org/10.1007/978-3-642-28496-0_19.

[22] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Keccak in VHDL*. https://keccak.team/hardware.html. 2012.

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The Keccak reference*. SHA-3 Submission. Version 3. https://keccak.team/files/Keccak-reference-3.0.pdf. 2011.

[24] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *CAESAR submission: Ketje v2*. https://keccak.team/ketje.html. 2016.

[25] Ingrid Biehl, Bernd Meyer, and Volker Müller. "Differential Fault Attacks on Elliptic Curve Cryptosystems". In: *Advances in Cryptology — CRYPTO 2000*. Ed. by Mihir Bellare. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 131–146. DOI: 10.1007/3-540-44598-6_8. URL: http://dx.doi.org/10.1007/3-540-44598-6_8.

[26] W. Bosma, J. J. Cannon, and C. Playoust. "The Magma Algebra System I: The User Language". In: *J. Symb. Comput.* 24.3/4 (1997), pp. 235–265. DOI: 10.1006/jsco.1996.0125. URL: http://dx.doi.org/10.1006/jsco.1996.0125.

[27] David Cash, Eike Kiltz, and Victor Shoup. "The Twin Diffie-Hellman Problem and Applications". In: *Advances in Cryptology – EUROCRYPT 2008*. Ed. by Nigel Smart. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 127–145. DOI: 10.1007/978-3-540-78967-3_8. URL: https://doi.org/10.1007/978-3-540-78967-3_8.

[28] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. "CSIDH: An Efficient Post-Quantum Commutative Group Action". In: *Advances in Cryptology – ASIACRYPT 2018*. Ed. by Thomas Peyrin and Steven Galbraith. Cham: Springer International Publishing, 2018, pp. 395–427. DOI: 10.1007/978-3-030-03332-3_15. URL: https://doi.org/10.1007/978-3-030-03332-3_15.

[29] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on Post-Quantum Cryptography*. Tech. rep. NIST, 2016. DOI: 10.6028/NIST.IR.8105. URL: https://doi.org/10.6028/NIST.IR.8105.

[30] Łukasz Chmielewski, Pedro Maat Costa Massolino, Jo Vliegen, Lejla Batina, and Nele Mentens. "Completing the Complete ECC Formulae with Countermeasures". In: *Journal of Low Power Electronics and Applications* 7.1 (2017). DOI: 10.3390/jlpea7010003. URL: http://www.mdpi.com/2079-9268/7/1/3.

[31] Christophe Clavier and Marc Joye. "Universal Exponentiation Algorithm A First Step towards Provable SPA-Resistance". In: *Cryptographic Hardware and Embedded Systems — CHES 2001*. Ed. by Çetin K. Koç, David Naccache, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 300–308. DOI: 10.1007/3-540-44709-1_25. URL: https://doi.org/10.1007/3-540-44709-1_25.

[32] David A. Cooper, Daniel C. Apon, Quynh H. Dang, Michael S. Davidson, Morris J. Dworkin, and Carl A. Miller. *Recommendation for Stateful Hash-Based Signature Schemes*. Tech. rep. NIST, Oct. 2020. DOI: 10.6028/NIST.SP.800-208. URL: https://doi.org/10.6028/NIST.SP.800-208.

[33] Jean-Sébastien Coron. "Resistance Against Differential Power Analysis For Elliptic Curve Cryptosystems". In: *Cryptographic Hardware and Embedded Systems*. Ed. by Çetin K. Koç and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 292–302. DOI: 10.1007/3-540-48059-5_25. URL: https://doi.org/10.1007/3-540-48059-5_25.

[34] Craig Costello. "B-SIDH: Supersingular Isogeny Diffie-Hellman Using Twisted Torsion". In: *Advances in Cryptology – ASIACRYPT 2020*. Ed. by Shiho Moriai and Huaxiong Wang. Cham: Springer International Publishing, 2020, pp. 440–463. DOI: 10.1007/978-3-030-64834-3_15. URL: https://doi.org/10.1007/978-3-030-64834-3_15.

[35] Craig Costello and Hüseyin Hisil. "A Simple and Compact Algorithm for SIDH with Arbitrary Degree Isogenies". In: *Advances in Cryptology - ASIACRYPT 2017*. Vol. 10625. Lecture Notes in Computer Science. Springer, 2017, pp. 303–329. DOI: 10.1007/978-3-319-70697-9_11. URL: https://doi.org/10.1007/978-3-319-70697-9_11.

[36] Craig Costello and Patrick Longa. "Four$\mathbb{Q}$: Four-Dimensional Decompositions on a $\mathbb{Q}$-curve over the Mersenne Prime". In: *Advances in Cryptology – ASIACRYPT 2015*. Ed. by Tetsu Iwata and Jung Hee Cheon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 214–235. DOI: 10.1007/978-3-662-48797-6_10. URL: https://doi.org/10.1007/978-3-662-48797-6_10.

[37] Craig Costello, Patrick Longa, and Michael Naehrig. "Efficient Algorithms for Supersingular Isogeny Diffie-Hellman". In: *Advances in Cryptology – CRYPTO 2016*. Ed. by Matthew Robshaw and Jonathan Katz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 572–601. DOI: 10.1007/978-3-662-53018-4_21. URL: https://dx.doi.org/10.1007/978-3-662-53018-4_21.

[38] Craig Costello, Patrick Longa, and Michael Naehrig. *SIDH Library. https://github.com/Microsoft/PQCrypto-SIDH*. 2016.

[39] Joan Daemen. "Cipher and hash function design, strategies based on linear and differential cryptanalysis, PhD Thesis". https://cs.ru.nl/~joan/. PhD thesis. 1995.

[40] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Xoodoo cookbook*. https://eprint.iacr.org/2018/767. Mar. 2019.

[41] Joan Daemen, Pedro Maat Costa Massolino, Alireza Mehrdad, and Yann Rotella. "The Subterranean 2.0 Cipher Suite". In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (June 2020), pp. 262–294. DOI: 10.13154/tosc.v2020.iS1.262-294. URL: https://tosc.iacr.org/index.php/ToSC/article/view/8622.

[42] Joan Daemen, Pedro Maat Costa Massolino, and Yann Rotella. *The Subterranean 2.0 cipher suite*. NIST Lightweight Cryptography round 2 candidate. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/subterranean-spec-round2.pdf.

[43] Alan Daly, William Marnane, Tim Kerins, and Emanuel Popovici. "An FPGA implementation of a GF(p) ALU for encryption processors". In: *Microprocessors and Microsystems* 28.5–6 (2004). Special Issue on FPGAs: Applications and Designs, pp. 253–260. DOI: 10.1016/j.micpro.2004.03.006. URL: https://doi.org/10.1016/j.micpro.2004.03.006.

[44] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. "A Practical Implementation of the Timing Attack". In: *Smart Card Research and Applications*. Ed. by Jean-Jacques Quisquater and Bruce Schneier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 167–182. DOI: 10.1007/10721064_15. URL: http://dx.doi.org/10.1007/10721064_15.

[45] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography". In: *Information Theory, IEEE Transactions on* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638. URL: https://doi.org/10.1109/TIT.1976.1055638.

[46] Digilent. *Arty A7: Artix-7 FPGA Development Board for Makers and Hobbyists*. https://store.digilentinc.com/arty-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/. 2019.

[47] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2*. Submission to the NIST LWC Competition, http://ascon.iaik.tugraz.at. 2019.

[48] Junfeng Fan, Kazuo Sakiyama, and Ingrid Verbauwhede. "Elliptic curve cryptography on embedded multicore systems". In: *Design Automation for Embedded Systems* 12.3 (2008), pp. 231–242. DOI: 10.1007/s10617-008-9021-3. URL: https://doi.org/10.1007/s10617-008-9021-3.

[49] Mohammad-Hossein Farzam, Siavash Bayat-Sarmadi, Hatameh Mosanaei-Boorani, and Armin Alivand. "Hardware Architecture for Supersingular Isogeny Diffie-Hellman and Key Encapsulation Using a Fast Montgomery Multiplier". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.5 (2021), pp. 2042–2050. DOI: 10.1109/TCSI.2021.3062871. URL: https://doi.org/10.1109/TCSI.2021.3062871.

[50] Luca De Feo, David Jao, and Jérôme Plût. "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies". In: *J. Mathematical Cryptology* 8.3 (2014), pp. 209–247. DOI: 10.1515/jmc-2012-0015. URL: http://www.degruyter.com/view/j/jmc.2014.8.issue-3/jmc-2012-0015/jmc-2012-0015.xml.

[51] J. Fuegi and J. Francis. "Lovelace & Babbage and the creation of the 1843 'notes'". In: *IEEE Annals of the History of Computing* 25.4 (2003), pp. 16–26. DOI: 10.1109/MAHC.2003.1253887. URL: https://www.scss.tcd.ie/coghlan/repository/J_Byrne/A_Lovelace/J_Fuegi_&_J_Francis_2003.pdf.

[52] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *J. Cryptology* 26.1 (2013), pp. 80–101. DOI: 10.1007/s00145-011-9114-1. URL: https://doi.org/10.1007/s00145-011-9114-1.

[53] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. "On the Security of Supersingular Isogeny Cryptosystems". In: *Advances in Cryptology - ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. 2016, pp. 63–91. DOI: 10.1007/978-3-662-53887-6_3. URL: https://doi.org/10.1007/978-3-662-53887-6_3.

[54] Santosh Ghosh, Monjur Alam, Dipanwita Roy Chowdhury, and Indranil Sen Gupta. "Parallel crypto-devices for GF(p) elliptic curve multiplication resistant against side channel attacks". In: *Computers & Electrical Engineering* 35.2 (2009). Circuits and Systems for Real-Time Security and Copyright Protec-

tion of Multimedia, pp. 329–338. DOI: 10.1016/j.compeleceng.2008.06.009. URL: https://doi.org/10.1016/j.compeleceng.2008.06.009.

[55] Johann Großschädl, Roberto M. Avanzi, Erkay Savaş, and Stefan Tillich. "Energy-Efficient Software Implementation of Long Integer Modular Arithmetic". In: *Cryptographic Hardware and Embedded Systems – CHES 2005*. Ed. by Josyula R. Rao and Berk Sunar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 75–90. DOI: 10.1007/11545262_6. URL: http://dx.doi.org/10.1007/11545262_6.

[56] Shay Gueron. "Enhanced Montgomery Multiplication". In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 46–56. DOI: 10.1007/3-540-36400-5_5. URL: http://dx.doi.org/10.1007/3-540-36400-5_5.

[57] Nicolas Guillermin. "A High Speed Coprocessor for Elliptic Curve Scalar Multiplications over $\mathbb{F}_p$". In: *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 48–64. DOI: 10.1007/978-3-642-15031-9_4. URL: https://doi.org/10.1007/978-3-642-15031-9_4.

[58] Tim Güneysu and Christof Paar. "Ultra High Performance ECC over NIST Primes on Commercial FPGAs". In: *Cryptographic Hardware and Embedded Systems – CHES 2008*. Vol. 5154. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 62–78. DOI: 10.1007/978-3-540-85053-3_5. URL: http://dx.doi.org/10.1007/978-3-540-85053-3_5.

[59] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs". In: *Cryptographic Hardware and Embedded Systems - CHES 2004*. Ed. by Marc Joye and Jean-Jacques Quisquater. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 119–132. DOI: 10.1007/978-3-540-28632-5_9. URL: http://dx.doi.org/10.1007/978-3-540-28632-5_9.

[60] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *Theory of Cryptography (TCC 2017)*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. Lecture Notes in Computer Science. Springer, 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12. URL: https://doi.org/10.1007/978-3-319-70500-2_12.

[61] 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. https://eprint.iacr.org/2016/626. 2016.

[62] Jin-Hua Hong and Cheng-Wen Wu. "Radix-4 modular multiplication and exponentiation algorithms for the RSA public-key cryptosystem". In: *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*. June 2000, pp. 565–570. DOI: 10.1109/ASPDAC.2000.835164. URL: https://doi.org/10.1109/ASPDAC.2000.835164.

[63] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. "Standard Lattice-Based Key Encapsulation on Embedded Devices". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018), pp. 372–393. DOI: 10.13154/tches.v2018.i3.372-393. URL: https://tches.iacr.org/index.php/TCHES/article/view/7279.

[64] Kimmo Järvinen, Andrea Miele, Reza Azarderakhsh, and Patrick Longa. "Four$\mathbb{Q}$ on FPGA: New Hardware Speed Records for Elliptic Curve Cryptography over Large Prime Characteristic Fields". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 517–537. DOI: 10.1007/978-3-662-53140-2_25. URL: https://doi.org/10.1007/978-3-662-53140-2_25.

[65] Khalid Javeed and Xiaojun Wang. "Efficient Montgomery Multiplier for Pairing and Elliptic Curve Based Cryptography". In: *Communication Systems, Networks Digital Signal Processing (CSNDSP), 2014 9th International Symposium on*. July 2014, pp. 255–260. DOI: 10.1109/CSNDSP.2014.6923835. URL: http://dx.doi.org/10.1109/CSNDSP.2014.6923835.

[66] Marc Joye and Sung-Ming Yen. "The Montgomery Powering Ladder". In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 291–302. DOI: 10.1007/3-540-36400-5_22. URL: https://doi.org/10.1007/3-540-36400-5_22.

[67] Jens-Peter Kaps, William Diehl, Michael Tempelmeier, Ekawat Homsirikamol, and Kris Gaj. *Hardware API for Lightweight Cryptography*. Tech. rep. https://cryptography.gmu.edu/athena/LWC/LWC_HW_API.pdf. 2020, pp. 1–26.

[68] A. Karatsuba and Yu. Ofman. "Multiplication of Many-Digital Numbers by Automatic Computers". In: *Proceedings of the USSR Academy of Sciences*. 1962, pp. 293–294.

[69] Auguste Kerckhoffs. "La cryptographie militaire". In: *Journal des sciences militaires* 9 (Jan. 1883), pp. 5–38. URL: https://www.petitcolas.net/kerckhoffs/crypto_militaire_1_b.pdf.

[70] Mustafa Khairallah, Thomas Peyrin, and Anupam Chattopadhyay. *Preliminary Hardware Benchmarking of a Group of Round 2 NIST Lightweight AEAD Candidates*. Cryptology ePrint Archive, Report 2020/1459. https://eprint.iacr.org/2020/1459. 2020.

[71] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of Computation* 48 (1987), pp. 203–209. DOI: 10.1090/S0025-5718-1987-0866109-5. URL: https://doi.org/10.1090/S0025-5718-1987-0866109-5.

[72] Çetin K. Koç, Tolga Acar, and Burton S. Kaliski Jr. "Analyzing and comparing Montgomery multiplication algorithms". In: *Micro, IEEE* 16.3 (June 1996), pp. 26–33. DOI: 10.1109/40.502403. URL: http://dx.doi.org/10.1109/40.502403.

[73] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology – CRYPTO' 99*. Vol. 1666. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25. URL: http://dx.doi.org/10.1007/3-540-48405-1_25.

[74] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9. URL: http://dx.doi.org/10.1007/3-540-68697-5_9.

[75] B. Koziel, A. Ackie, R. El Khatib, R. Azarderakhsh, and M. M. Kermani. "SIKE'd Up: Fast Hardware Architectures for Supersingular Isogeny Key Encapsulation". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2020), pp. 1–13. DOI: 10.1109/TCSI.2020.2992747. URL: http://dx.doi.org/10.1109/TCSI.2020.2992747.

[76] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. "A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography". In: *IEEE Transactions on Computers* (2018), pp. 1594–1609. DOI: 10.1109/TC.2018.2815605. URL: https://doi.org/10.1109/TC.2018.2815605.

[77] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. "Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA". In: *Progress in Cryptology - INDOCRYPT 2016*. Ed. by Orr Dunkelman and Somitra Kumar Sanadhya. Vol. 10095. Lecture Notes in Computer Science. Springer, 2016, pp. 191–206. DOI: 10.1007/978-3-319-49890-4_11. URL: https://doi.org/10.1007/978-3-319-49890-4_11.

[78] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. RFC 7748. RFC Editor, 2016, pp. 1–22. URL: http://www.rfc-editor.org/rfc/rfc7748.txt.

[79] Weiqiang Liu, Ziying Ni, Jian Ni, Ciara Rafferty, and Máire O'Neill. "High Performance Modular Multiplication for SIDH". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2020), pp. 3118–3122. DOI: 10.1109/TCAD.2019.2960330. URL: https://doi.org/10.1109/TCAD.2019.2960330.

[80] K. C. C. Loi and S. B. Ko. "Scalable Elliptic Curve Cryptosystem FPGA Processor for NIST Prime Curves". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.11 (Nov. 2015), pp. 2753–2756. DOI: 10.1109/TVLSI.2014.2375640. URL: http://dx.doi.org/10.1109/TVLSI.2014.2375640.

[81] Patrick Longa, Wen Wang, and Jakub Szefer. *The Cost to Break SIKE: A Comparative Hardware-Based Analysis with AES and SHA-3*. Cryptology ePrint Archive, Report 2020/1457. https://eprint.iacr.org/2020/1457. 2020.

[82] A. A. Lovelace. "Notes by A.A.L. [August Ada Lovelace]". In: *Taylor's Scientific Memoirs* (1843), pp. 666–731.

[83] Yuan Ma, Zongbin Liu, Wuqiong Pan, and Jiwu" Jing. "A High-Speed Elliptic Curve Cryptographic Processor for Generic Curves over GF(p)". In: *Selected Areas in Cryptography – SAC 2013: 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 421–437. DOI: 10.1007/978-3-662-43414-7_21. URL: http://dx.doi.org/10.1007/978-3-662-43414-7_21.

[84] Pedro Maat C. Massolino, Lejla Batina, Ricardo Chaves, and Nele Mentens. "Area-optimized Montgomery multiplication on IGLOO 2 FPGAs". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056762. URL: https://dx.doi.org/10.23919/FPL.2017.8056762.

[85] Pedro Maat C. Massolino, Lejla Batina, Ricardo Chaves, and Nele Mentens. *Low Power Montgomery Modular Multiplication on Reconfigurable Systems*. Cryptology ePrint Archive, Report 2016/280. 2016. URL: https://eprint.iacr.org/2016/280.

[86] Pedro Maat C. Massolino, Barış Ege, and Lejla Batina. "Smart Card Fault Injections with High Temperatures". In: *6th Conference on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE 2016)*. Nov. 2016. URL: https://upcommons.upc.edu/handle/2117/99293.

[87] Pedro Maat C. Massolino, Patrick Longa, Joost Renes, and Lejla Batina. "A Compact and Scalable Hardware/Software Co-design of SIKE". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.2 (Mar. 2020), pp. 245–271. DOI: 10.13154/tches.v2020.i2.245-271. URL: https://tches.iacr.org/index.php/TCHES/article/view/8551.

[88] Pedro Maat C. Massolino, Joost Renes, and Lejla Batina. "Implementing Complete Formulas on Weierstrass Curves in Hardware". In: *Security, Privacy, and Applied Cryptography Engineering. SPACE 2016*. Vol. 10076. Cham: Springer International Publishing, 2016, pp. 89–108. DOI: 10.1007/978-3-319-49445-6_5. URL: https://doi.org/10.1007/978-3-319-49445-6_5.

[89] R. J. McEliece. *The Theory of Information and Coding: Student Edition*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511819896.

[90] Ciaran McIvor, Máire McLoone, and John V. McCanny. "FPGA Montgomery multiplier architectures - a comparison". In: *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*. Apr. 2004, pp. 279–282. DOI: 10.1109/FCCM.2004.36. URL: http://dx.doi.org/10.1109/FCCM.2004.36.

[91] Ciaran McIvor, Máire McLoone, and John V. McCanny. "Hardware Elliptic Curve Cryptographic Processor Over GF(p)". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 53.9 (Sept. 2006), pp. 1946–1957. DOI: 10.1109/TCSI.2006.880184. URL: http://dx.doi.org/10.1109/TCSI.2006.880184.

[92] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Boca Raton, Florida, USA: CRC Press, 1996.

[93] Microsemi. *IGLOO2 Product Information Brochure*. Tech. rep. Microsemi, 2014. URL: http://www.microsemi.com/document-portal/doc_download/132013-igloo2-product-information-brochure.

[94] Victor Miller. "Use of Elliptic Curves in Cryptography". In: *Advances in Cryptology - CRYPTO 85 Proceedings*. Vol. 218. Lecture Notes in Computer Science. Berlin, Germany: Springer Berlin / Heidelberg, 1986, pp. 417–426. DOI: 10.1007/3-540-39799-X_31. URL: http://dx.doi.org/10.1007/3-540-39799-X_31.

[95] Kamyar Mohajerani. *Hardware implementation of Lightweight Cryptography candidates in Bluespec SystemVerilog*. GitHub repository. https://github.com/kammoh/bluelight. 2021.

[96] Kamyar Mohajerani, Richard Haeussler, Rishub Nagpal, Farnoud Farahmand, Abubakr Abdulgadir, Jens-Peter Kaps, and Kris Gaj. *FPGA Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process: Methodology, Metrics, Tools, and Results*. Cryptology ePrint Archive, Report 2020/1207. https://eprint.iacr.org/2020/1207. 2020.

[97] Peter L. Montgomery. "Modular Multiplication without Trial Division". In: *Mathematics of Computation* 44.170 (1985), pp. 519–521. DOI: 10.1090/S0025-5718-1985-0777282-X. URL: https://doi.org/10.1090/S0025-5718-1985-0777282-X.

[98] Dustin Moody. *Round 2 of the NIST PQC "Competition" – What was NIST thinking?* Presentation PQCrypto 2019, https://csrc.nist.gov/CSRC/media/Presentations/Round-2-of-the-NIST-PQC-Competition-What-was-NIST/images-media/pqcrypto-may2019-moody.pdf. 2019.

[99] Amine Mrabet, Nadia El-Mrabet, Ronan Lashermes, Jean-Baptiste Rigaud, Belgacem Bouallegue, Sihem Mesnager, and Mohsen Machhout. *A Systolic Hardware Architectures of Montgomery Modular Multiplication for Public Key Cryptosystems*. Cryptology ePrint Archive, Report 2016/487. 2016. URL: http://eprint.iacr.org/2016/487.

[100] National Institute for Standards and Technology. *Federal Information Processing Standards Publication 186-4. Digital signature standard*. Tech. rep. NIST, 2013. DOI: 10.6028/NIST.FIPS.186-4. URL: http://dx.doi.org/10.6028/NIST.FIPS.186-4.

[101] National Institute for Standards and Technology. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Tech. rep. NIST, 2015. DOI: 10.6028/NIST.FIPS.202. URL: http://dx.doi.org/10.6028/NIST.FIPS.202.

[102] H. D. Nguyen, B. Pasca, and T. B. Preußer. "FPGA-Specific Arithmetic Optimizations of Short-Latency Adders". In: *2011 21st International Conference on Field Programmable Logic and Applications*. Sept. 2011, pp. 232–237. DOI: 10.1109/FPL.2011.49. URL: http://dx.doi.org/10.1109/FPL.2011.49.

[103] Tobias Oder and Tim Güneysu. "Implementing the NewHope-Simple Key Exchange on Low-Cost FPGAs". In: *Progress in Cryptology - LATINCRYPT 2017*. Ed. by Tanja Lange and Orr Dunkelman. Vol. 11368. Lecture Notes in Computer Science. Springer, 2019, pp. 128–142. DOI: 10.1007/978-3-030-25283-0_7. URL: https://doi.org/10.1007/978-3-030-25283-0_7.

[104] Gerardo Orlando and Christof Paar. "A Scalable GF(p) Elliptic Curve Processor Architecture for Programmable Hardware". In: *Cryptographic Hardware and Embedded Systems — CHES 2001*. Vol. 2162. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 348–363. DOI: 10.1007/3-540-44709-1_29. URL: http://dx.doi.org/10.1007/3-540-44709-1_29.

[105] Sıddıka Berna Örs, Lejla Batina, Bart Preneel, and Joos Vandewalle. "Hardware implementation of a Montgomery modular multiplier in a systolic array". In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. Apr. 2003, p. 8. DOI: 10.1109/IPDPS.2003.1213341. URL: http://dx.doi.org/10.1109/IPDPS.2003.1213341.

[106] H. Orup. "Simplifying quotient determination in high-radix modular multiplication". In: *Proceedings of the 12th Symposium on Computer Arithmetic*. 1995, pp. 193–199. DOI: 10.1109/ARITH.1995.465359. URL: http://dx.doi.org/10.1109/ARITH.1995.465359.

[107] Matthew R. Pillmeier, Michael J. Schulte, and E. George Walters. "Design alternatives for barrel shifters". In: *Proceedings of SPIE - The International Society for Optical Engineering* 4791 (Dec. 2002), pp. 436–447. DOI: 10.1117/12.452034. URL: https://doi.org/10.1117/12.452034.

[108] Christopher Pöpper, Oliver Mischke, and Tim Güneysu. "MicroACP - A Fast and Secure Reconfigurable Asymmetric Crypto-Processor". In: *Reconfigurable Computing: Architectures, Tools, and Applications*. Vol. 8405. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 240–247. DOI: 10.1007/978-3-319-05960-0_24. URL: http://dx.doi.org/10.1007/978-3-319-05960-0_24.

[109] Thomas B. Preußer and Markus Krause. "Survey on and re-evaluation of wide adder architectures on FPGAs". In: *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2016, Cancun, Mexico, November 30 - Dec. 2, 2016*. 2016, pp. 1–6. DOI: 10.1109/ReConFig.2016.7857189. URL: https://doi.org/10.1109/ReConFig.2016.7857189.

[110] Joost Renes. "Security on the Line: Modern Curve-based Cryptography". https://joostrenes.nl/pubs.html. PhD thesis. Nijmegen: Radboud University, 2019.

[111] Joost Renes, Craig Costello, and Lejla Batina. "Complete Addition Formulas for Prime Order Elliptic Curves". In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 403–428. DOI: 10.1007/978-3-662-49890-3_16. URL: https://doi.org/10.1007/978-3-662-49890-3_16.

[112] Ronald L. Rivest, Adi Shamir, and Len Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[113] D. B. Roy, D. Mukhopadhyay, M. Izumi, and J. Takahashi. "Tile before multiplication: An efficient strategy to optimize DSP multiplier for accelerating prime field ECC for NIST curves". In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–6. DOI: 10.1145/2593069.2593234. URL: https://doi.org/10.1145/2593069.2593234.

[114] Debapriya Basu Roy, Poulami Das, and Debdeep Mukhopadhyay. "ECC on Your Fingertips: A Single Instruction Approach for Lightweight ECC Design in GF(p)". In: *Selected Areas in Cryptography – SAC 2015: 22nd International Conference, Sackville, NB, Canada, August 12–14, 2015, Revised Selected Papers*. Cham: Springer International Publishing, 2016, pp. 161–177. DOI: 10.1007/978-3-319-31301-6_9. URL: http://dx.doi.org/10.1007/978-3-319-31301-6_9.

[115] Debapriya Basu Roy, Tim Fritzmann, and Georg Sigl. "Efficient Hardware/Software Co-Design for Post-Quantum Crypto Algorithm SIKE on ARM and RISC-V Based Microcontrollers". In: ICCAD '20. Virtual Event, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3400302.3415728. URL: https://doi.org/10.1145/3400302.3415728.

[116] Debapriya Basu Roy and Debdeep Mukhopadhyay. *Post Quantum ECC on FPGA Platform*. Cryptology ePrint Archive, Report 2019/568. https://eprint.iacr.org/2019/568. 2019.

[117] W. A. Stein et al. *Sage Mathematics Software (Version 7.0)*. http://www.sagemath.org. The Sage Development Team. 2016.

[118] Kazuo Sakiyama, Nele Mentens, Lejla Batina, Baart Preneel, and Ingrid Verbauwhede. "Reconfigurable Modular Arithmetic Logic Unit for High-Performance Public-Key Cryptosystems". In: *Reconfigurable Computing: Architectures and Applications*. Vol. 3985. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 347–357. DOI: 10.1007/11802839_43. URL: http://dx.doi.org/10.1007/11802839_43.

[119] Miguel Morales Sandoval and Arturo Diaz Perez. *Novel algorithms and hardware architectures for Montgomery Multiplication over GF(p)*. Cryptology ePrint Archive, Report 2015/696. 2015. URL: http://eprint.iacr.org/2015/696.

[120] Pascal Sasdrich and Tim Güneysu. "Cryptography for next generation TLS: Implementing the RFC 7748 elliptic Curve448 cryptosystem in hardware". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–6. DOI: 10.1145/3061639.3062222. URL: http://dx.doi.org/10.1145/3061639.3062222.

[121] Pascal Sasdrich and Tim Güneysu. "Efficient Elliptic-Curve Cryptography Using Curve25519 on Reconfigurable Devices". In: *Reconfigurable Computing: Architectures, Tools, and Applications*. Vol. 8405. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 25–36. DOI: 10.1007/978-3-319-05960-0_3. URL: http://dx.doi.org/10.1007/978-3-319-05960-0_3.

[122] C.P. Schnorr. "Efficient signature generation by smart cards". In: *Journal of Cryptology* 4 (1991), pp. 161–174. DOI: 10.1007/BF00196725. URL: https://doi.org/10.1007/BF00196725.

[123] Michael Scott. "Implementing Cryptographic Pairings". In: *Proceedings of the First International Conference on Pairing-Based Cryptography*. Pairing'07. Tokyo, Japan: Springer-Verlag, 2007, pp. 177–196. DOI: 10.5555/2394336.2394352. URL: http://dx.doi.org/10.5555/2394336.2394352.

[124] Michael Scott. *Slothful reduction*. Cryptology ePrint Archive, Report 2017/437. 2017. URL: https://eprint.iacr.org/2017/437.

[125] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Review* 41.2 (1999), pp. 303–332. DOI: 10.1137/S0036144598347011. URL: https://doi.org/10.1137/S0036144598347011.

[126] Victor Shoup. *A Proposal for an ISO Standard for Public Key Encryption (version 2.1)*. https://www.shoup.net/papers/iso-2_1.pdf. Dec. 2001.

[127] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer-Verlag New York, 2009. DOI: 10.1007/978-0-387-09494-6. URL: https://doi.org/10.1007/978-0-387-09494-6.

[128] Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat C. Massolino, Kostas Papagiannopoulos, Francesco Regazzoni, and Niels Samwel. "Friet: an Authenticated Encryption Scheme with Built-in Fault Detection". In: *Advances in Cryptology – EUROCRYPT 2020*. Vol. 12105. Cham: Springer International Publishing, 2020, pp. 581–611. DOI: 10.1007/978-3-030-45721-1_21. URL: https://doi.org/10.1007/978-3-030-45721-1_21.

[129] M. Taha and P. Schaumont. "Side-channel countermeasure for SHA-3 at almost-zero area overhead". In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2014, pp. 93–96. DOI: 10.1109/HST.2014.6855576. URL: http://dx.doi.org/10.1109/HST.2014.6855576.

[130] GCC Team. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/. 2021.

[131] LLVM Team. *The LLVM Compiler Infrastructure*. https://www.llvm.org/. 2021.

[132] A. F. Tenca and L. A. Tawalbeh. "An efficient and scalable radix-4 modular multiplier design using recoding techniques". In: *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*. Vol. 2. Nov. 2003, pp. 1445–1450. DOI: 10.1109/ACSSC.2003.1292225. URL: http://dx.doi.org/10.1109/ACSSC.2003.1292225.

[133] Meltem Sönmez Turan, Kerry A. McKay, Çağdaş Çalık, Donghoon Chang, and Larry Bassham. *Status Report on the First Round of the NIST Lightweight Cryptography Standardization Process*. Tech. rep. NIST, Oct. 2019. DOI: 10.6028/NIST.IR.8268. URL: https://doi.org/10.6028/NIST.IR.8268.

[134] Michal Varchola, Tim Güneysu, and Oliver Mischke. "MicroECC: A Lightweight Reconfigurable Elliptic Curve Crypto-processor". In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. Nov. 2011, pp. 204–210. DOI: 10.1109/ReConFig.2011.61. URL: http://dx.doi.org/10.1109/ReConFig.2011.61.

[135] Jo Vliegen, Nele Mentens, Jan Genoe, An Braeken, Serge Kubera, Abdellah Touhafi, and Ingrid Verbauwhede. "A compact FPGA-based architecture for elliptic curve cryptography over prime fields". In: *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*. July 2010, pp. 313–316. DOI: 10.1109/ASAP.2010.5540977. URL: http://dx.doi.org/10.1109/ASAP.2010.5540977.

[136] C.D. Walter. "Montgomery exponentiation needs no final subtractions". In: *Electronics Letters* 35.21 (Oct. 1999), pp. 1831–1832. DOI: 10.1049/el:19991230. URL: http://dx.doi.org/10.1049/el:19991230.

[137] Colin D. Walter. "Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli". In: *Topics in Cryptology — CT-RSA 2002*. Ed. by Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 30–39. DOI: 10.1007/3-540-45760-7_3. URL: https://doi.org/10.1007/3-540-45760-7_3.

[138] Gavin Xiaoxu Yao, Junfeng Fan, Ray C. C. Cheung, and Ingrid Verbauwhede. "Faster Pairing Coprocessor Architecture". In: *Pairing-Based Cryptography – Pairing 2012: 5th International Conference, Cologne, Germany, May 16-18, 2012, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–176. DOI: 10.1007/978-3-642-36334-4_10. URL: https://doi.org/10.1007/978-3-642-36334-4_10.

# Summary

This thesis describes hardware designs and evaluation for distinct asymmetric and symmetric cryptographic building blocks. Those evaluations can later be applied or improved for future hardware designs of the same cryptosystem. The thesis is divided into 5 chapters. Chapter 1 gives common background information for the hardware designs of Chapters 2, 3 and 4. Chapters 2, 3 and 4 present hardware implementations of asymmetric cryptography building blocks. Chapter 5 consists of 3 different contributions proposing new symmetric key components, they were bundled together into only the hardware architecture and evaluation of each one. Here are more details of Chapters 2, 3, 4 and 5.

**Chapter 2**

Following the studies on Montgomery multiplication, I designed and evaluated two Montgomery multiplication word-based algorithms called CIOS and FIOS by Koç et al. [72]. Each algorithm was implemented with its own architecture, while exploiting the IGLOO2 FPGAs DSPs and memory modules. Because of the design choices, the results show a very compact and high frequency architecture for Montgomery multiplication in the IGLOO2 FPGA.

**Chapter 3**

I used the Montgomery multipliers from Chapter 2 as the main building block for a full elliptic curve scalar multiplication co-processor. The elliptic curve scalar multiplication co-processor was made in order to evaluate Renes et al. [111] Weierstrass complete addition formulas and to showcase possible parallelizations. Even though the presented elliptic curve formulas have been parallellized from 2 up to 6 multiplier units, the hardware proof of concept only employed 3 units.

**Chapter 4**

I implemented a key encapsulation scheme based on isogenies between elliptic curves called Supersingular Isogeny Key Encapsulation Mechanism (SIKE) [8]. Because of the size of the project and SIKE's complexity, it was implemented as a hardware/software co-design. However, unlike a more traditional hardware/software co-design where a generic CPU is used together with one or more co-processors, in this project the CPU was custom made to be as simple as possible and enough to run the SIKE algorithm. The co-processor used for this work is not the one in Chapter 2,

but a new one called `Carmela`. The `Carmela` co-processor applies a different Montgomery multiplication algorithm than the one from Chapters 2 and 3, and also an optimization for Montgomery-friendly primes.

**Chapter 5**

I compared and implemented 3 lightweight proposals called: Gimli, Friet and Subterranean 2.0.

Gimli is a 384 bits permutation that was conceived as a small permutation, when compared to SHA-3, with reasonable performance in software and hardware. Gimli-24-Cipher and Gimli-24-Hash are AE and hash functions based on the Gimli permutation and the sponge construction. Just like Gimli, Friet-P is also a small permutation, but it is engineered for fault detection through error correction codes. This is achieved with 384 bits permutation split into 3 128-bit parts, called $a$, $b$ and $c$, and the parity part of 128 bits $d$, thus making it 512 bits. The 4 parts have the following relationship $a_i \oplus b_i \oplus c_i = d_i$ for $i \in [0, 127]$, thus if one bit $a_i$, $b_i$, $c_i$ or $d_i$ is flipped, then it is possible to detect, while two are undetectable. For the SAE mode Friet also combines the sponge construction alongside the Friet-P permutation. Finally, Subterranean 2.0 is a cipher suite with a 257-bit permutation based on the original primitive called Subterranean by Daemen [39]. In the new version, the permutation was kept from the original, new input and output locations were chosen and a sponge duplex mode was built on top to offer an XOF, SAE and MAC.

# Samenvatting (Dutch summary)

Dit proefschrift beschrijft en evalueert hardwareontwerpen voor verschillende asymmetrische en symmetrische cryptografische primitieven. Deze evaluaties kunnen gebruikt worden bij toekomstige ontwerpen van dezelfde cryptosystemen. Het proefschrift bestaat uit vijf hoofdstukken. Hoofdstuk 1 bevat de basiskennis en achtergrondinformatie voor de hardwareontwerpen gepresenteerd in Hoofdstukken 2, 3 en 4. Dit gaat om ontwerpen van bouwstenen voor asymmetrische cryptografie. In Hoofdstuk 5 worden drie bijdragen beschreven voor de symmetrische cryptografie. We beschrijven nu Hoofdstukken 2 – 5 in meer detail.

### Hoofdstuk 2

Na Montgomeryvermenigvulding bestudeerd te hebben, heb ik twee *word-based* algoritmen, te weten CIOS en FIOS door Koç et al. [72], voor hardware geschikt gemaakt en geëvalueerd. Beide hebben een verschillende architectuur gekregen die de DSPs en geheugenmodulen van de IGLOO2 FPGAs in hun voordeel gebruiken. Dankzij deze ontwerpkeuzes is het resultaat twee erg compacte ontwerpen, die op hoge frequentie kunnen draaien, voor de IGLOO2 FPGA.

### Hoofdstuk 3

Met de Montgomery vermenigvuldigingscomponenten uit het vorige hoofdstuk, ontwerp ik een co-processor voor scalaire vermenigvuldiging in elliptische krommen met als doel om de Weierstrass volledige optellingsformules van Renes et al. [111] te beoordelen en mogelijke parallellisatie aan te tonen. Hoewel de formules 6 vermenigvuldigingscomponententen tegelijkertijd kunnen gebruiken, gebruikt het *proof of concept* er slechts 3.

### Hoofdstuk 4

In dit hoofdstuk implementeer ik *Supersingular Isogeny Key Encapsulation* (SIKE) [8], een *key encapsulation* schema gebasseerd op isogenieën. Door de complexiteit en omvang van het project, heb ik gekozen voor een hardware/software co-design. In tegenstelling tot een traditioneel co-design, waar een normale CPU wordt gekoppeld aan een of meerdere co-processors, heb ik voor dit ontwerp een zo simpel mogelijke CPU ontworpen die het SIKE algoritme kan uitvoeren. Voor deze CPU wordt "`Carmela`", een nieuwe vermenigvuldigings component gebruikt, die een ander Montgomery algoritme

gebruikt dan de componenten uit Hoofdstukken 2 en 3, dat bovendien geoptimaliseerd kan worden voor Montgomery-vriendelijke priemgetallen.

**Hoofdstuk 5**

Ik vergelijk en implementeer drie verschillende lichtgewicht versleutelingsprimitieven: Gimli, Friet en Subterranean 2.0.

Gimli is een 384-bits permutatie ontworpen als een klein alternatief op SHA-3 met schappelijke prestaties in software en hardware. Gimli-24-Cipher en Gimli-24-Hash zijn een AE en resp. een hash functie geconstrueerd met behulp van Gimli en de spons-constructie. Friet-P is, net als Gimli, een kleine permutatie, maar ontworpen om fouttolerant te zijn met behulp van foutverbeterende codes. Dit wordt gedaan door de 384-bits van de permutatie in drie 128-bits delen te splitsen: $a$, $b$ en $c$. Een vierde pariteitsdeel $d$ wordt toegevoegd met de volgende vergelijking: $a_i \oplus b_i \oplus c_i = d_i$ waar $i \in [0, 127]$ wat resulteert in 512 bits totaal. Een enkele fout binnen $a_i$, $b_i$, $c_i$ of $d_i$ kan correct herkend en gecorrigeerd worden, maar twee of meer niet. Voor de SAE-modus combineert Friet een spons-constructie met Friet-P. Tot slot is Subterranean 2.0 een versleuteling met een 257-bits permutatie gebaseerd op de originele primitieve Subterranean van Daemen [39]. De onderliggende permutatie is hetzelfde gebleven, maar andere in- en uitvoerslocaties zijn gekozen naast het specificeren van een nieuwe duplex spons-modus om een XOF, SAE en MAC aan te kunnen bieden.

# List of Publications

### Conference/Journal proceedings

- Pedro Maat C. Massolino, Joost Renes, and Lejla Batina. "Implementing Complete Formulas on Weierstrass Curves in Hardware". In: *Security, Privacy, and Applied Cryptography Engineering. SPACE 2016*. Vol. 10076. Cham: Springer International Publishing, 2016, pp. 89–108. DOI: 10.1007/978-3-319-49445-6_5. URL: https://doi.org/10.1007/978-3-319-49445-6_5

- Pedro Maat C. Massolino, Barış Ege, and Lejla Batina. "Smart Card Fault Injections with High Temperatures". In: *6th Conference on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE 2016)*. Nov. 2016. URL: https://upcommons.upc.edu/handle/2117/99293

- Łukasz Chmielewski, Pedro Maat Costa Massolino, Jo Vliegen, Lejla Batina, and Nele Mentens. "Completing the Complete ECC Formulae with Countermeasures". In: *Journal of Low Power Electronics and Applications* 7.1 (2017). DOI: 10.3390/jlpea7010003. URL: http://www.mdpi.com/2079-9268/7/1/3

- Pedro Maat C. Massolino, Lejla Batina, Ricardo Chaves, and Nele Mentens. "Area-optimized Montgomery multiplication on IGLOO 2 FPGAs". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056762. URL: https://dx.doi.org/10.23919/FPL.2017.8056762

- Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. "Gimli : A Cross-Platform Permutation". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Cham: Springer International Publishing, 2017, pp. 299–320. DOI: 10.1007/978-3-319-66787-4_15. URL: https://dx.doi.org/10.1007/978-3-319-66787-4_15

- Pedro Maat C. Massolino, Patrick Longa, Joost Renes, and Lejla Batina. "A Compact and Scalable Hardware/Software Co-design of SIKE". in: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.2 (Mar. 2020), pp. 245–271. DOI: 10.13154/tches.v2020.i2.245-271. URL: https://tches.iacr.org/index.php/TCHES/article/view/8551

- Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat C. Massolino, Kostas Papagiannopoulos, Francesco Regazzoni, and Niels Samwel. "Friet: an Authenticated Encryption Scheme with Built-in Fault Detection". In: *Advances in Cryptology – EUROCRYPT 2020*. Vol. 12105. Cham: Springer International Publishing, 2020, pp. 581–611. DOI: 10.1007/978-3-030-45721-1_21. URL: https://doi.org/10.1007/978-3-030-45721-1_21

- Joan Daemen, Pedro Maat Costa Massolino, Alireza Mehrdad, and Yann Rotella. "The Subterranean 2.0 Cipher Suite". In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (June 2020), pp. 262–294. DOI: 10.13154/tosc.v2020.iS1.262-294. URL: https://tosc.iacr.org/index.php/ToSC/article/view/8622

## Preprints

- Pedro Maat C. Massolino, Lejla Batina, Ricardo Chaves, and Nele Mentens. *Low Power Montgomery Modular Multiplication on Reconfigurable Systems*. Cryptology ePrint Archive, Report 2016/280. 2016. URL: https://eprint.iacr.org/2016/280

## Other

- Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. *Gimli*. NIST Lightweight Cryptography round 2 candidate. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/gimli-spec-round2.pdf

- Joan Daemen, Pedro Maat Costa Massolino, and Yann Rotella. *The Subterranean 2.0 cipher suite*. NIST Lightweight Cryptography round 2 candidate. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/subterranean-spec-round2.pdf

# Curriculum Vitae

**August 2020 –**

Cryptographic Engineer
PQShield, Oxford, UK

**July 2018 – October 2018**

Intern
Riscure, the Netherlands

**January 2015 – July 2020**

PhD Candidate
Digital Security Group (DiS)
Radboud University, the Netherlands

**January 2011 – July 2014**

Master of Science
Departamento de Engenharia de Computação e Sistemas Digitais (PCS)
University of São Paulo (USP) – Escola Politécnica, Brazil

**January 2006 – December 2010**

Bachelor in electrical engineering with computer emphasis
Departamento de Engenharia de Computação e Sistemas Digitais (PCS)
University of São Paulo (USP) – Escola Politécnica, Brazil