A Panorama on Classical Cryptography

Designing, Implementing, Breaking, Verifying, and Standardizing Cryptography

Benoît Viguier

A Panorama on Classical Cryptography

Designing, Implementing, Breaking, Verifying, and Standardizing Cryptography

Benoît Viguier

Invitation

to attend the public defense
of my doctoral thesis

A Panorama on Classical Cryptography

Designing, Implementing, Breaking, Verifying, and Standardizing Cryptography

on Monday
December 13th, 2021
at 16:30

in the Aula of the
Radboud University,
Comeniuslaan 2,
in Nijmegen

Afterwards there will be
a reception to which you are
also cordially invited.

Benoît Viguier.

Intentionally left blank.

A PANORAMA ON CLASSICAL CRYPTOGRAPHY

BENOÎT VIGUIER

A PANORAMA ON CLASSICAL CRYPTOGRAPHY

Designing, Implementing, Breaking, Verifying, and Standardizing Cryptography

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 13 december 2021
om 16:30 uur precies

door

Benoît Georges Pierre Viguier

geboren op 29 maart 1988
te Angers – Frankrijk

PROMOTOREN:

prof. dr. Peter Schwabe

prof. dr. Joan Daemen

COPROMOTOR:

dr. Freek Wiedijk

MANUSCRIPTCOMMISSIE:

dr. Diego F. Aranha
Aarhus Universitet, Denemarken

prof. dr. Gilles Barthe
Instituto IMDEA Software, Spanje

prof. dr. Lejla Batina — *voorzitter*

prof. dr. María Naya-Plasencia
Inria Paris, Frankrijk

dr. Pierre-Yves Strub
Laboratoire d'Informatique de l'École Polytechnique (LIX), Frankrijk

C’est par le principe des bon repas,
que l’on devient gros et gras.

— Grand Père (1929 – 2002)

At the beginning of your PhD you have to make a choice:
SkyTeam or Star Alliance?

— Peter Schwabe, 2016

ACKNOWLEDGMENTS

This thesis concludes what I consider without doubts as the best four and a half years of my life, and for this I am extremely grateful for all the support I was provided with over this period.

First and foremost, I would like to thank my supervisors Peter, Freek, Joan, and Herman. Their invisible contribution cannot be overstated as their doors were always open to my questions and doubts; I really appreciated their openness and our enlightening discussions.

Over the course of those years, I have had the daily company of the brilliant minds of my PhD colleagues and friends. I had the pleasure of not only sharing offices —the corner room and PQHQ— with them, but also those moments at the multiple conference venues, hotels & airbnb. For this I thank Daan, Denisa, Joost & Joost, Ko, Matthias, Niels, Pedro, and Thom. I would also like to extend my gratitude to my other colleagues Marc, Dan, Jon, John, Bas, Anna G., Anna K., Kostas, Louiza, Lukasz, Paulus, Lejla, Hugo, Fabian, Engelbert, Ronny, Simona, Bart, Eric, Mireille, Eelco, Shanley, and Irma, with whom I had enjoyable discussions during the 10:30am coffee break. I also would like to further thank Pedro, not only a colleague but also an amazing roommate, it was a pleasure having you around and your baking skills will be missed greatly. Finally, a special mention to Jon and Marc which whom I share the passion of photography; discussing the craft, the tools, the technique and the art has always been delightful.

While this thesis is under my sole name, the work presented here would not have been possible without the numerous collaborations with my coauthors, especially at the Lorentz Center in Leiden. For this I thank Tomer Ashur, Daniel J. Bernstein, Guido Bertoni, Fabio Campos, Joan Daemen, Maria Eichlseder, Lars Jellema, Stefan Kölbl, Martin M. Lauridsen, Mauk Lemmen, Gaëtan Leurent, Pedro Maat Costa Massolino, Florian Mendel, Brice Minaud, Lars Müller, Kashif Nawaz, Michaël Peeters, Yann Rotella, Yu Sasaki, Tobias Schneider, Peter Schwabe, Daan Sprenkels, François-Xavier Standaert, Yosuke Todo, Gilles Van Assche, Ronny Van Keer, Freek Wiedijk. They openly shared their knowledge and provided me with insightful discussions.

I would like to thank Lennart Beringer and Andrew Appel for their hospitality during my visit at Princeton University. Their insight and help resolved some of the issues I was facing with the VST, allowing me to further improve and speed-up my proofs. Similarly, thanks to John Wiegley for taking the time to explain to me proofs by reflection

during the DeepSpec Summer School, those revealed to be extremely useful later. In the same field, I would like to thank Jade Philipoom, Andres Erbsen, and Jason Gross for their inputs on how to improve my proofs techniques during the HACS workshops. I thank Benjamin Grégoire and Damien Rouhling for their advices during my short stay at Sophia-Antipolis.

I am glad to have been able to meet brilliant minds and now friends at the DeepSpec Summer School 2017 in Philly. Alyssa, Noah, Adam, Olivier, Matthew, Ekatarina, and Willy are amazing; I am looking forward to meeting you all again.

All this moving around highlights two things: first, that research blossoms in collaborations between individuals, and secondly, that I was extremely lucky to have Peter, Freek and Joan as supervisors. They provided me with numerous opportunities to visit so many Institutes, and expand my area of research; for all of this, thank you again.

I would like to give my regards and thanks to Gilles Van Assche and Michaël Peeters for their support at STMicroelectronics, to Maxime Van Assche and Michel Vanden Bossche at Mission Critical IT, and to Ivan Leplumey and Mireille Ducassé at the National Institute of Applied Sciences (INSA Rennes).

Writing a PhD requires not only a good work environment, but also the support of friends. For this I am glad to have my dance trainers Roel and Claudia. Thank you for your motivation and drive which pushed me to succeed in sports as well as academically. I am extremely grateful to have Laura as my dance partner, her support during this last year cannot be highlighted enough, especially through my multiple and unfortunate injuries. For this, thank you, Laura.

Last but not least, I thank my close friends. Linde, for all those delightful conversations and precious advices you provided me through those past years; Denisa, for your endless energy while at work; Lily, for your happiness in our passion for photography; Inga, for your precious time while you where in the Netherlands.

Et enfin, merci Papa et Maman pour votre soutien alors que je ne vous vois qu'en rares occassions; merci de me soutenir et de m'encourager dans mes choix malgré la distance qui nous sépare.

Nijmegen, June 2021
Benoît Viguier

CONTENTS AT A GLANCE

TABLE OF CONTENTS

Part I

INTRODUCTION & PRELIMINARIES

"Take the first step in faith. You don't have to see the whole staircase, just take the first step." — Martin Luther King Jr.

1 INTRODUCTION

The main subject of this thesis is classical cryptography. As the subtitle *"Designing, Implementing, Breaking, Verifying, and Standardizing Cryptography"* suggests, we will be covering a wide field of the topic with a large part of this panorama focused on symmetric cryptography.

1.1 A BIT OF HISTORY

For thousands of years secrecy in communication has been needed. A general communicating with his lieutenant, a king with his armies, a spy with his country, a queen trying to overthrow her cousin etc. Examples are numerous. In the following we explore briefly the foundations of modern cryptography.

1883 – KERCKHOFFS'S PRINCIPLE In January and February 1883, Auguste Kerckhoffs published two articles under the title *La Cryptographie Militaire (Military Cryptography)* [Ker83]. These papers surveyed in depth the cryptographic techniques available at that time, and some of the cryptanalysis approaches used to break them. In addition to being one of the first Systematization-of-Knowledge (SoK) paper in the subject, Kerckhoffs also included practical rules to observe in regard to the future development of cipher designs. Those rules also known as the six principles, are the following:

1. The system must be practically, if not mathematically, indecipherable;
2. It should not require secrecy, and it should not be a problem if it falls into enemy hands;
3. It must be possible to communicate and remember the key without using written notes, and correspondents must be able to change or modify it at will;
4. It must be applicable to telegraph communications;
5. It must be portable, and should not require several persons to handle or operate;
6. Lastly, given the circumstances in which it is to be used, the system must be easy to use and should not be stressful to use or require its users to know and comply with a long list of rules.

It becomes quickly apparent that some of those propositions are now outdated, e. g., number 4. With regard to modern times, aside from

the obvious rule number 1, the most important is the second one. This has become the de facto standard for any newly submitted cipher as it allows for significantly more cryptanalysis and stronger attack scenarios.

1917 – VERNAM CIPHER Gilbert S. Vernam was working at AT&T Bell Labs as engineer when he invented in 1917 what is known today as the Vernam Cipher [Ver26]. It worked on a teleprinter where the user would use two paper tapes, one was the *message* while the other was the *key*, both encoding a string of characters in Baudot code, a 5-bit code ancestor of ASCII and UTF-8. Each parallel bit of the tapes was added modulo 2, effectively doing an *exclusive or* operation and generating the *ciphertext* which was subsequently transmitted over the cable. Upon reception, the recipient uses then the same key tape and reproduces the message.

Few years later, Captain of the US Army Joseph Mauborgne suggested to use random information on the key tapes such that they were as long as the message. By combining this idea with the Vernam cipher, this implemented what is known today as the One-Time Pad (OTP)[1].

1970-TODAY – ERA OF MODERN SYMMETRIC CRYPTOGRAPHY In 1972, the National Bureau of Standards (NBS) —later renamed as National Institute of Standards and Technology (NIST)— with consultation of the National Security Agency (NSA) was looking for a cryptographic algorithm for commercial use, e. g., in the banking sector. The idea would be to have a unified standard —Federal Information Processing Standard (FIPS)— that would be secure enough for the American companies. As none of the first candidates were deemed suitable, a second call for proposals was made in 1974. That is when International Business Machines Corporation (IBM) submitted a candidate cipher similar to LUCIFER [Fei73] designed by Horst Feistel in 1971. After a few modifications suggested by the NSA in 1975, the cipher was standardized as the Data Encryption Standard (DES) in FIPS 46 [Sta77] in 1977 and in FIPS 74 [Sta81] in 1981.

It took thirteen years of research to find an attack breaking the security claim of DES. The first one discovered by Biham and Shamir is the differential cryptanalysis (see more in Section 2.5). The second attack was proposed by Mitsuru Matsui in 1993 and is known as linear cryptanalysis.

Furthermore, with the computing power growing over time, the 56-bit key size became vulnerable to brute force attacks (an attacker would try all the possible keys) and DES was effectively broken in less than a day at the beginning of 1999 [FLG98].

In light of such attacks, NIST issued in 1997 a call for proposals [SN97] aimed to replace DES. The 3-year process took place with

1 As matter of fact, it was first described in 1882 by the American cryptographer Frank Miller [Bel11]

3 rounds and 15 candidates. Of the 5 finalists in October 2002, Rijndael [DR02], designed by Joan Daemen and Vincent Rijmen, got selected as the Advanced Encryption Standard (AES) [NIS01] for its *"elegance, efficiency, security and principled design"* – Rivest [Riv02].

1976-TODAY – ASYMMETRIC CRYPTOGRAPHY While symmetric cryptography makes uses of a shared secret key, this key still needs to be transported or initially established in a secure manner which is sometimes hard to ensure. In order to solve the problem, Diffie and Hellman proposed a protocol [DH76] making use of number theory. The algorithm is now known as the Diffie-Hellman-Merkle key-exchange in recognition for Merkle's contribution to public-key cryptography [Hel02].

Public-key cryptography or asymmetric cryptography makes use of two keys. The first one—called the *secret key* or *private key*—is kept by the recipient while the second one—called the *public key*—is widely distributed for anyone to use to encrypt a message. The most well-known example of such system was designed in 1976 by Rivest, Shamir, and Adleman: the RSA cryptosystem [RSA78] which is used in GNU Privacy Guard (GPG) to encrypt for example mails.

However, one of the weaknesses of the RSA cryptosystem is the size of the keys and numbers on which operations have to be computed. To solve this problem Koblitz [Kob87] and Miller [Mil86] propose to use Elliptic-Curve Cryptography (ECC), working with significantly smaller numbers while keeping the same security level.

In 2006, Bernstein proposes Curve25519 [Ber06a] and X25519, the Diffie-Hellman-Merkle key-exchange protocol associated to it. This protocol is standardized in RFC 7748 [LHT] and used by a wide variety of applications [Thi] such as SSH, Signal Protocol, Tor, Zcash, and TLS to establish a shared secret over an insecure channel.

CLASSICAL CRYPTOGRAPHY & QUANTUM ATTACKS The security of asymmetric cryptography relies mainly on two problems whose complexity sharply increases with the size of the prime numbers chosen. On one hand for RSA, it is the difficulty of factoring a large integer:

$$\textit{given } N\textit{, find the primes } p \textit{ and } q \textit{ such that } p \times q = N.$$

For example, 143 can be easily factorized in 11×13, however it is significantly more difficult to do by hand for 5293. On the other hand the Diffie-Hellman-Merkle protocol relies on the discrete-logarithm problem, one of its variation being (here simplified):

$$\textit{given } z, g\textit{, and } N \textit{ prime such that } z \equiv g^x \mod N\textit{, find } x.$$

Unfortunately Shor discovered in 1994 a quantum algorithm —an algorithm which requires a quantum computer— for integer factorization [Sho94] that solves this problem in polynomial time. In other words, increasing the size of the number used does not provide significant improvement in the security of the schemes. Additionally, Shor's algorithm also breaks the discrete logarithm problem, putting basically all the current asymmetric cryptography at risk. For this reason, NIST started the Post Quantum Cryptography project [SN17].

While asymmetric cryptography is at risk one could think that symmetric cryptography is safe from such attacks, but that would be without Grover's research. In 1996, he discovered another quantum algorithm which significantly shorten the time to search the correct key for a cipher [Gro96; Gro97]. For example, assuming a lock with 4 digits, there exists 10 000 possible combinations for the correct sequence, meaning an average of $10000/2 = 5000$ tries for a classical algorithm. Grover's algorithm would need only $\sqrt{10000} = 100$ tries to find the correct combination. Fortunately this threat is solved by using longer keys.

Nevertheless, the research into new cryptographic algorithms which are "quantum-proof" is part of the *Post-Quantum Cryptography* field as opposed to *Classical Cryptography*. This thesis being titled A PANORAMA ON CLASSICAL CRYPTOGRAPHY, it will not cover them in the following chapters.

LIGHTWEIGHT CRYPTOGRAPHY In the recent years, an enormous growth of the "Internet of Things" (IOT) is changing the world. Forecasts [Liu19] project the number of interconnected embedded devices to around 50 billion worldwide by 2030, a five-fold increase in the next ten years. Driven by the lack of standardized cryptographic algorithms which are suitable for such constrained environments, NIST started in 2015 a project (NIST-LWC) [SN15] to solicit, evaluate, and eventually standardize lightweight authenticated-encryption algorithms with associated data (AEAD) and hashing.

In light of this project we propose the new cipher GIMLI which we present in Chapter 4.

Lightweight Cryptography (LWC), a sub-field of cryptography, covers cryptographic algorithms intended for use in constrained hardware and software environments. The main goal of NIST's project is to provide algorithms that are more suitable for use on constrained devices where the performance of current NIST cryptographic standards is not acceptable. Thereby, performance figures should be considered on a wide range of 8-bit, 16-bit and 32-bit microcontroller architectures.

For this reason, in Chapter 5 we chose one of such architecture and proposed optimized implementations of a selection of the 32 candidates.

THE NEED FOR FORMAL VERIFICATION As large cryptographic systems being composed of thousands lines of code and multiple complex algorithms, it is not surprising to discover bugs and vulnerabilities in them. The bigger the code size, the larger the risk. For example, during the period 2016-2020, the cryptographic library OpenSSL [Opea] has been subject to many bugs leading to Common Vulnerabilities and Exposures (CVE) [Opeb], including 8 null-pointer arithmetic/dereferencing, 4 overflow bugs, 13 memory safety errors, 8 side-channel leaks, and 4 large-number arithmetic bugs.

Such vulnerabilities usually lead to disastrous consequences such as the disclosure of private information. For example, Figure 1.1[2] briefly illustrates how the 2014 Heartbleed [Meh14] vulnerability, a missing bound check, can be exploited to recover private information from a server.

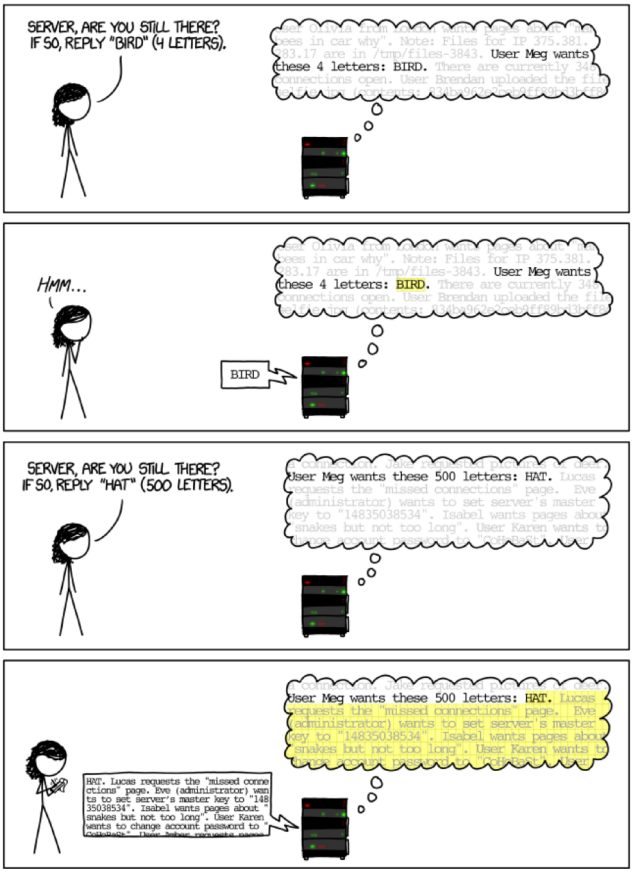


Figure 1.1: How the Heartbleed bug works. XKCD/1354

2 https://xkcd.com/1354/

Finding such bugs usually relies heavily on manual code inspection, testing, mutating, and fuzzing. Additionally, while some vulnerabilities may look difficult to exploit, Brumley et al. [Bru+12] have shown that even a simple carry bug inside an elliptic-curve implementation may lead to the retrieval of a long-term private key.

This illustrates the need of formal verification in cryptographic software, in other words mathematically proving the absence of entire classes of potential bugs and vulnerabilities. However, such techniques are seen as tedious and time-consuming; for this reason, the most notable formal approach applied to cryptographic have focused in generating proven by construction software. This concept has been used to create HACL* [Zin+17; ZBB16; Pro+17] in the NSS libraries by Mozilla or the implementation of ECC generated via Coq [Erb+19; Erb+16; Erb17] in BoringSSL [Bor].

However, while generating correct software improves the global quality and security of cryptographic software, it does not remove the need to prove the correctness of already existing codebases. For this reason, in Section 4.2 of [Bru+12], Brumley et al. describe from a high-level point of view a method to formally verify the correctness of cryptographic implementations. This is the approach that we will use in Chapter 7 to prove a specific implementation of X25519.

1.2 ORGANIZATION OF THIS THESIS

This thesis is composed of nine chapters divided into four parts aimed at covering different areas of classical cryptography. Preliminaries aside, there is little dependence between chapters, allowing the reader to quickly skip through some of the material presented here.

PART I: INTRODUCTION & PRELIMINARIES is composed of this chapter followed by two preliminaries (Chapters 2 and 3). In those we first cover the basics of symmetric cryptography before providing the foundations for formal reasoning and software verification.

PART II: DESIGNING, IMPLEMENTING, BREAKING spreads over three chapters. First Chapter 4 covers the design of the cryptographic primitive GIMLI, then Chapter 5 looks at the implementation of cryptographic algorithms, taking as targeted platform the RISC-V architecture. Finally, Chapter 6 provides the cryptanalysis of MORUS with which we broke the security claim.

PART III: VERIFYING is composed of the single Chapter 7 and takes a look at the verification of the asymmetric primitive X25519. We verify that a C implementation matches the RFC specification. Additionally, we prove that the RFC is correct with regard to the original paper before going one step further, verifying the correctness of the operations described in the protocol with respect to the theory of elliptic curves.

PART IV: STANDARDIZING is split into two chapters. First, in Chapter 8, we introduce KANGAROOTWELVE a tree-based hash function using the SHA-3 permutation; then we describe the standardization process used by CFRG in Chapter 9.

Figure 1.2 describes the dependence between the chapters, providing multiple possible reading orders. The colors of the nodes highlight which section of this thesis title is illustrated by the chapter.

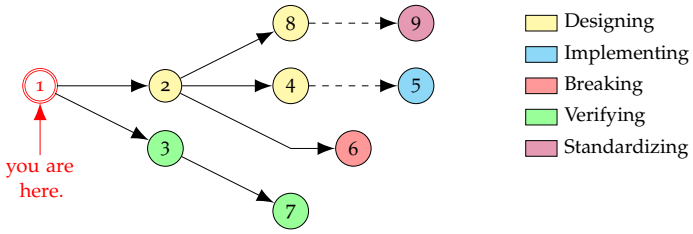


Figure 1.2: Logical dependence of the chapters.

1.3 CONTRIBUTIONS

The main chapters of this thesis are based on five published papers at different venues and can also be found on the International Association for Cryptologic Research (IACR) ePrint archive. While their main subject remains unchanged, minor modifications have been applied to align notations, unify style, combine overlaps, or include additional pieces of information not available in the original publications due to the limitation in length of the submissions.

Because they are results of collaborations with a large number of coauthors, it is often not an easy task to determine the individual contribution of a specific author. As a result in all the papers named below, authors are listed in alphabetical order with respect to the 2004 culture statement from the American Mathematical Society (AMS) [Soc].

SOFTWARE AVAILABILITY. Most of the material presented here comes with associated software which we place in public domain and make available at:

`https://doi.org/10.5281/zenodo.4534692`

Additionally, in order to optimally reproduce our results we advise the use of Linux environments.

THE READER, THE AUTHORS & I. It is often difficult to decide which narrative to use while writing a thesis. In a general setting, it is easily apparent that the use of the singular first-person is ill-advised as it overshadows the work of the coauthors. The ambiguity is more visible in the use of *we*. Does it refer to the authors of a specific paper or does it refer to *us* —the reader and *I*— going together through the material presented in this thesis?

The core of this manuscript is composed of published papers, of which none with a single author; therefore the first-person will naturally refer to my coauthors and *I*. There will be few exceptions but the context should help *us* to resolve this duality.

The rest of this section describes the content of the principal chapters of this thesis and highlights my individual contribution to the original related paper.

CHAPTER 4: GIMLI

In this chapter we present GIMLI, a 384-bit permutation designed to achieve high security with high performance across a broad range of platforms, including 64-bit Intel/AMD server CPUs, 64-bit and 32-bit ARM smartphone CPUs, 32-bit ARM microcontrollers, 8-bit AVR microcontrollers, FPGAs, ASICs without side-channel protection, and ASICs with side-channel protection. We further discuss our submission at the NIST's lightweight cryptography competition.

This chapter is based on the following publication:

> 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*. LNCS 10529. Springer, 2017.

CONTRIBUTION: I participated in the initial design of the permutation and helped in the choice of the constants by implementing simple prototypes with different parameters, allowing to quickly test and discard some of the shift and rotational constants in the design. I also helped in benchmarking the execution speed of the different linear layer options. Additionally, I am responsible for the implementations in Python, C and in assembly on the following platforms: AVR, ARM Cortex-M0, ARM Cortex-M3, and ARM Cortex-M4. In terms of cryptanalysis, I provided the propagation analysis and checked the avalanche criterion with the method of Monte-Carlo.

CHAPTER 5: ASSEMBLY OR OPTIMIZED C FOR LIGHTWEIGHT CRYPTOGRAPHY ON RISC-V?

In this chapter we provide different optimization strategies for several candidates of NIST's lightweight cryptography standardization project

on a RISC-V architecture. We studied the general impact of optimizing symmetric-key algorithms in assembly and in plain C. Additionally, we present optimized implementations, achieving a speed-up of up to 81% over available implementations at that time, and discuss general implementation strategies.

This work led to the following publication:

> Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoît Viguier. "Assembly or Optimized C for Lightweight Cryptography on RISC-V?" In: *CANS 2020: Cryptology and Network Security*. LNCS 12579. Springer, 2020.

CONTRIBUTION: I provided the benchmark and compilation framework for our optimized implementations, allowing us to easily switch between platforms (simulators or board) and algorithms. Additionally, I am responsible for the code of GIMLI, I fixed the implementation of DELIRIUM and I ported the assembly implementations of KECCAK and AES from Stoffelen [Sto19] into optimized C. Using our framework I also automated the benchmark of Weatherley's optimized implementations [Wea20].

CHAPTER 6: CRYPTANALYSIS OF MORUS

In this chapter we have a look at MORUS, a high-performance authenticated encryption algorithm submitted to the CAESAR competition, and selected as a finalist. There are three versions of MORUS: MORUS-640 with a 128-bit key, and MORUS-1280 with 128-bit or 256-bit keys. For all versions the security claim for confidentiality matches the key size. We analyze the components of this algorithm (initialization, state update and tag generation), and report several results.

As our main result, we present a linear correlation in the keystream of full MORUS, which can be used to distinguish its output from random and to recover some plaintext bits in the broadcast setting. For MORUS-1280, the correlation is 2^{-76} and can be exploited after around 2^{152} encryptions, which is less than what would be expected for a 256-bit secure cipher. For MORUS-640, the same attack results in a correlation of 2^{-73}, which does not violate the security claims of the cipher.

To identify this correlation, we make use of rotational symmetries in MORUS using linear masks that are invariant by word-rotations of the state. This motivates us to introduce single-word versions of MORUS called MINIMORUS, simplifying the analysis. The attack has been implemented and verified on MINIMORUS, where it yields a correlation of 2^{-16}.

We also study reduced versions of the initialization and finalization of MORUS, aiming to evaluate the security margin of these components. We show a forgery attack when finalization is reduced from 10 steps to 3, and a key-recovery attack in the nonce-misuse setting when

initialization is reduced from 16 steps to 10. These additional results do not threaten the full MORUS, but studying all aspects of the design is useful to understand its strengths and weaknesses.

This work led to the following publication:

> Tomer Ashur, Maria Eichlseder, Martin M. Lauridsen, Gaëtan Leurent, Brice Minaud, Yann Rotella, Yu Sasaki, and Benoît Viguier. "Cryptanalysis of MORUS." In: *Advances in Cryptology – ASIACRYPT 2018*. LNCS 11273. Springer, 2018.

CONTRIBUTION: In this collaborative work at the Lorentz center I provided the team the implementations of MINIMORUS, which we used to verify in practice the biases of the fragments and subsequently of the trails. By applying simple parallelism optimizations and using vector instructions, I further verified the complexity of the trail fragments for MORUS-640 and MORUS-1280.

CHAPTER 7: A COQ PROOF OF THE CORRECTNESS OF X25519 IN TWEETNACL

In this chapter we formally prove that the C implementation of the X25519 key-exchange protocol in the TweetNaCl library is correct. We prove both that it correctly implements the protocol from Bernstein's 2006 paper, as standardized in RFC 7748. We also formally prove, based on the work of Bartzia and Strub, that X25519 is mathematically correct, i. e., that it correctly computes scalar multiplication on the elliptic curve Curve25519. The proofs are all computer-verified using the Coq theorem prover. To establish the link between C and Coq we use the Verifiable Software Toolchain (VST).

This work led to the following publication:

> Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. "A Coq proof of the correctness of X25519 in TweetNaCl." In *34th IEEE Computer Security Foundations Symposium - CSF 2021*.

CONTRIBUTION: In terms of time dedicated to my research, this paper outclasses all the other publications presented here. I used the Master's Thesis by Weerwag as a base, and extended his work to prove the correctness of Montgomery ladder for Curve25519 over the quadratic extension field $\mathbb{F}_{p^2}$, thus providing a computer-verified version of Bernstein's proof [Ber06a]. I wrote the specification of RFC 7748 [LHT] in Coq and proved correct with respect to the original paper. I verified that the crypto scalar multiplication in TweetNaCl and subsequently all the low level arithmetic matches the Request for Comments (RFC) specification with the VST.

CHAPTER 8: KANGAROOTWELVE

In this chapter we present KangarooTwelve, a fast and secure arbitrary output-length hash function aiming at a higher speed than the FIPS-202's Secure Hash Algorithm (SHA)-3 and SHAKE functions. While sharing many features with SHAKE128, like the cryptographic primitive, the sponge construction, the eXtendable Output Function (XOF) and the 128-bit security strength, KangarooTwelve offers two major improvements over its standardized counterpart. First it has a built-in parallel mode that efficiently exploits multi-core or single-instruction-multiple-data (SIMD) instruction parallelism for long messages, without impacting the performance for short messages. Second, relying on the cryptanalysis results on Keccak over the past ten years, we tuned its permutation to require twice less computation effort while still offering a comfortable safety margin. By combining these two changes KangarooTwelve consumes less than 0.55 cycles/byte for long messages on the latest Intel SkylakeX architectures. The generic security of KangarooTwelve is guaranteed by the use of Sakura encoding for the tree hashing and of the sponge construction for the compression function.

This work led to the following publication at Applied Cryptography and Network Security (ACNS):

> Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. "KangarooTwelve: Fast Hashing Based on Keccak-*p*." In: *Applied Cryptography and Network Security – 16th International Conference, ACNS 2018*. LNCS 10892. Springer, 2018.

CONTRIBUTION: My contribution to this paper will not be found in the design, nor in the security analysis or implementations. Under advises of the Keccak Team, I wrote the draft and submitted to the Crypto Forum Research Group (CFRG) to make it an RFC. My experience in standardizing such a function and the interactions with this working group are summarized in the next chapter.

CHAPTER 9: THE IETF-IRTF STANDARDIZATION PROCESS

In this last short chapter, we briefly describe the IETF, the IRTF and the CFRG. Then we have a quick look at the writing of an RFC before focusing on its standardization process, with an emphasis on the steps and phases to go through withing the CFRG. Finally, we provide the timeline of the KangarooTwelve draft.

CONTRIBUTION: I was responsible for the internet draft, more precisely its writing, attending the meetings to have it move forward, and ensuring communications with the CFRG.

2

A BRIEF INTRODUCTION TO SYMMETRIC CRYPTOGRAPHY

In this chapter we aim to provide the necessary background in symmetric cryptography for the main part of this manuscript, namely Chapters 4 to 6 and 8. This chapter is divided into five sections which we advise reading in order of appearance.

In the first section we review basic definitions and notations the reader will encounter through this thesis. We briefly cover the binary operations and the few possible representations used to work on data.

In a second section, we use a bottom-up approach. We first review the basic construction of *cryptographic primitives*, such as permutations, and block ciphers. With those notions at hand, we briefly describe hash functions and encryption schemes with authenticated data. We then study the associated security notions to those primitives by giving a brief intuition of how security games work.

The third section serves as foundation for Chapter 4; we provide a survey on the *sponge construction*, a structure which is now widely adopted and used in various new algorithms. We describe the main design idea and its various applications such as hash functions and encryption schemes before covering its security properties.

In the fourth section, we provide a direct illustration of the previously introduced construction: we describe the Keccak-p permutation and its associated integration in SHA-3. This algorithm is then reused and serves as groundwork for Chapter 8.

In the last section of those preliminaries, we conclude with a short introduction to *differential cryptanalysis*. This attack is used frequently as one of the first security analysis for any new symmetric cryptographic design. In particular, we use it in Chapter 4 to assert the security strength of our design.

Additionally, a second kind of attack similar to differential cryptanalysis and known as *linear cryptanalysis* will make a guest appearance in Chapter 6. As a result, terminologies and concepts described in the previous section will come handy as they can also be applied in that context.

With the map of this chapter in mind, we turn our focus to the basic notations and definitions used through this thesis.

2.1 DEFINITIONS AND NOTATIONS

The following notations are used throughout the document:

BITS.
A bit is an element of $\mathbb{F}_2$. We denote a string of bits with single quotes, e. g., '`0`' or '`1`'. To shorten some notations, the n-times repetition of a bit '`s`' is denoted '`s`^{n}', e. g., '`110`^{4}' = '`110000`'. We only consider two operations for bit-strings. The concatenation of a and b is denoted as $a \| b$; and the truncation of a bit-string s to its first n bits is denoted $\lfloor s \rfloor_n$. The length of a bit-string s is denoted $|s|$. The empty bit-string is denoted as $*$. Note that '`0`^{0}' $= *$.

We denote the set of bit-strings of n elements as $\mathbb{F}_2^n$, and for arbitrary length bit-strings $\mathbb{F}_2^*$.

BYTES.
A byte is a string of 8 bits '$b_0 b_1 \dots b_7$' which can also be represented by the integer value $\sum_i 2^i b_i$ written in hexadecimal. E. g., the bit-string '`11110010`' can be equivalently written as `0x4F` as depicted in Figure 2.1. The length of a byte-string s is denoted $\|s\|$. In a similar fashion as with the bits, (`0x00`)^n denotes the n times repetition of the byte `0x00`, for example, (`0x00`)^7 = `0x00 00 00 00 00 00 00`.

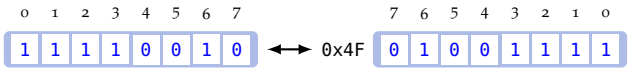


Figure 2.1: Example of byte representation.

WORDS.
A word is a sequence of bytes, often seen as the "native" data size of a CPU. For example, a 16-bit CPU will use 16-bit words. There are some exceptions; on modern x86 architecture a word is defined to be 16-bit wide for historical reasons. Extensions to 32 and 64 bits lead to the definition of DWORD —as double word— for 32-bit words and QWORD —as quad word— for 64-bit words.

In most cases a word is prefixed by its the width in specifications in order to avoid ambiguity.

ENDIANNESS.
The byte ordering inside a word is defined as endianness and primarily expressed as *big-endian* and *little-endian*. In a big-endian system, the most significant byte of a word is placed at the lowest index position. On the opposite, on a little-endian system, the least significant byte is at the lowest index position. This indexing becomes more obvious when we consider memory addresses.

As an example, in Figure 2.2 we show how the 32-bit decimal number 168496141 = `0x0A 0B 0C 0D` can be stored in memory on big-endian and little-endian systems.

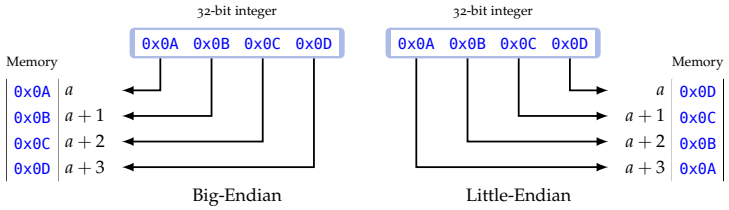


Figure 2.2: The Big-Endian and Little-Endian memory representation of 168496141 at an address a.

Big-endian ordering is also referred as the network order, sending the most significant byte first. Another common example is numbers, as they are written with their digits in big-endian order.

Little-endian ordering is used in most of recent CPU architectures such as Intel, ARM, and RISC-V etc.. Some architecture such as RISC-V offers the possibility to switch the endianness of data fetches and stores.

SIGNEDNESS.

So far we have only seen how bit-strings, and endianness are used to represent positive (or null) values. Note that those are also called *unsigned*.

A *signed* representation is necessary to allow the use of negative number. There exists mainly two ways to represent signed values: *one's complement*, and *two's complement*. In the *one's complement* representation, a positive value is transformed into its negative equivalent by inverting all the bits in the word. In the *two's complement* representation, a positive value is made into its negative by inverting all the bits in a word and adding 1 to the result.

Note that while the range of unsigned values of an 8-bit word is 0 to 255, the range of a signed value of an 8-bit word is -128 (or -127 depending on the representation) to 127 as the bit of highest index is used to determine the sign of the value.

The fact that *two's complement* can easily be implemented in hardware made this representation the most used today, covering virtually all processors. As a result, we will only consider this representation in this document.

BINARY OPERATIONS.

To provide consistency through this document, we use the following notation for bitwise operations:

- $\neg a$ to denote a bitwise negation (NEG) of the value a.

- $a \oplus b$ to denote a bitwise exclusive or (XOR) of the values a and b.
- $a \wedge b$ for a bitwise logical and (AND) of the values a and b.
- $a \vee b$ for a bitwise logical or (OR) of the values a and b,

While NEG, XOR, AND, and OR are bitwise operations, they are easily extended to bit-strings of equal length and words by being applied in parallel to each bit.

Those operations also have a mathematical equivalent over $\mathbb{F}_2$. XOR corresponds to the addition while AND is the multiplication. Similarly, NEG is equivalent to adding 1.

Furthermore, we use the following notations for word operations:

- $a \lll k$ for a cyclic left shift of the value a by a shift distance of k.
- $a \ll k$ for a non-cyclic left shift (i.e, a shift that is filling up with zero bits) of the value a by a shift distance of k.
- $a \gg k$ for a non-cyclic right shift of the value a by a shift distance of k. A logical right shift will always pad with zero bits. In an arithmetic one, the padding will be determined by the sign of the value, in other words, of the most significant bit.

Using the two's complement representation, $a \ll k$ is equivalent to the multiplication of a by 2^k (under the assumption that the resulting value is within the representable range of word). Similarly, $a \gg k$ is equivalent to the division of a by 2^k.

LINEAR OPERATIONS.

An important notion that is tightly tied to cryptanalysis is the *linearity* (See more in Section 2.5). Considering words as vectors of bit, a function $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ is a linear map if it preserves the addition over $\mathbb{F}_2$ (i. e., XOR). In other words, for all $x, y \in \mathbb{F}_2^n$:

$$f(x \oplus y) = f(x) \oplus f(y).$$

Such functions are equivalent to a matrix multiplication: $f : x \to A \cdot x$ with A am $m \times n$ matrix. Similarly, a function $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ is affine if it has an expression of the form:

$$f : x \to Ax \oplus B$$

with A an $m \times n$ matrix and B an m-bit vector. As a direct consequence; rotations, and shifts, and XOR are linear; NEG is affine; AND and OR are nor linear nor affine. By extension and abuse of language, we consider that NEG is a "linear" operation as it preserve the relevant kind of structure in the context considered.

2.2 PERMUTATIONS, BLOCK CIPHERS AND HASH FUNCTIONS

We now turn our focus on the basic components of a cryptographic system. We first describe the elements before discussing the security notions related to them.

2.2.1 *Elements*

STREAM CIPHER.
Vernam's approach in 1917 (see Chapter 1), i. e., XORing the plaintext with the key has the significant drawback of requiring the key and the message to be of the same length. This inconvenience is solved by using a stream cipher.

DEFINITION 2.2.1. *A k-bit stream cipher $\mathcal{SC}$ is a function which given a k-bit key returns a keystream of arbitrary length.*

$$\begin{aligned} \mathcal{SC} : \mathbb{F}_2^k &\to \mathbb{F}_2^* \\ K &\mapsto Z \end{aligned}$$

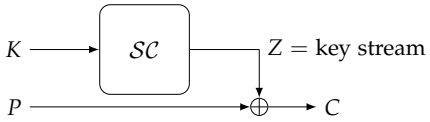


Figure 2.3: Encryption of a plaintext P into a ciphertext C with a stream cipher using a key K.

A well-known example of such design is CHACHA20, a 256-bit stream cipher designed by Bernstein in 2008 [Ber08b].

CRYPTOGRAPHIC PERMUTATION.
One of the most basic component of a cryptographic system is the permutation.

DEFINITION 2.2.2. *A b-bit cryptographic permutation $\mathcal{P}$ is a bijective function that is easy to evaluate.*

$$\begin{aligned} \mathcal{P} : \mathbb{F}_2^b &\to \mathbb{F}_2^b \\ x &\mapsto y \end{aligned}$$

To be considered for cryptographic operations, a permutation should not present a structural distinguisher; in other words, resist to attacks such as for example differential cryptanalysis, linear cryptanalysis, etc. An interesting property was formalized by Bertoni et al. [Ber+11a] as the hardness of constrained-input constrained-output (CICO) problems: Given two sets $X \subseteq \mathbb{F}_2^b$ and $Y \subseteq \mathbb{F}_2^b$, it should be no easier than for a random permutation to find an input-output pair (x, y) such that $y = \mathcal{P}(x)$ and $x \in X, y \in Y$.

BLOCK CIPHER.
With this definition at hand, we also define a block cipher.

DEFINITION 2.2.3. *A b-bit block cipher with k-bit key is defined by a couple of functions: an encryption function $\mathcal{E}$ and a decryption function $\mathcal{D}$.*

$$\begin{aligned}\mathcal{E} : \mathbb{F}_2^k \times \mathbb{F}_2^b &\to \mathbb{F}_2^b \\ K, P &\mapsto C\end{aligned}$$

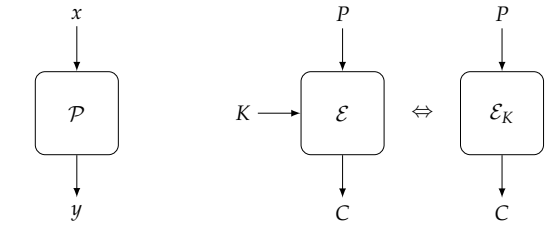
Given a k-bit key K and a b-bit plaintext block M, $\mathcal{E}$ produces a b-bit ciphertext C.

$$\begin{aligned}\mathcal{D} : \mathbb{F}_2^k \times \mathbb{F}_2^b &\to \mathbb{F}_2^b \\ K, C &\mapsto P\end{aligned}$$

Similarly, given a k-bit key K and a b-bit ciphertext block C, $\mathcal{D}$ retrieves the b-bit plaintext P.

A well-known example of block ciphers is the AES [DR02]. It works on plaintext blocks of 128 bits, thus b is equal to 128 bits, and its key has one of the following length k: 128 bits, 192 bits or 256 bits.

A block cipher must have the following properties: given a key K, $\mathcal{E}_K : M \mapsto E(K, M)$ is a cryptographic permutation (see Figure 2.4), and for all keys K, for all plaintexts P, $\mathcal{D}(K, \mathcal{E}(K, P)) = P$.



a permutation $\mathcal{P}$. a block cipher $\mathcal{E}$.

Figure 2.4: Symmetric cryptographic primitives as permutations.

The base security model of a block cipher is the pseudo-random permutation (PRP) assumption: for a randomly unknown chosen key K, the permutation E_K should be hard to distinguish from a b-bit permutation taken randomly over the set of $(2^b)!$ possibles permutations of b bits. This idea of indistinguishability is described later in Section 2.2.2.

Designing a block cipher as a single function that mixes properly all the input bits into a random looking output would be quite a complex task. Additionally, the cost of evaluating the security strength of such operation increases with the size of the input. This is why it is easier to design a simpler (and weaker) function which is then iterated multiple times over the data. Those are called *round functions*.

To remove potential symmetry, round function may use a different key at each iteration. Such *round keys* are derived from the input key.

Most block cipher designs fall into one–or more–of these categories: Feistel Network, Add Rotate and Xor (ARX) or Substitution Permutation Network (SPN).

Feistel networks were made popular by the DES, standardized by the NIST in 1977 [Sta77], and later the Fast data Encipherment ALgorihtm (FEAL) proposed by Shimizu and Miyaguchi [SM87] in 1987. They work on two half state $X_{i,Left}$ and $X_{i,Right}$, first by applying a key-dependent function F on one half and XORing the result into the other half, before swapping halves. Figure 2.5 illustrates a simple 3-round Feistel network.

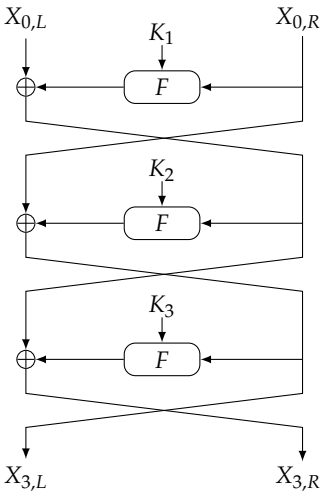


Figure 2.5: A 3-round Feistel cipher.

Such construction has multiple benefits, for example F does not need to be invertible; additionally since XOR and swaps are involutions, it is easy to provide a decryption function by applying the same operations in reverse order. Furthermore, by removing the last swap, the decryption function is the same as the encryption function but with the keys applied in reverse order.

While Feistel Networks help to decrease the complexity of a block cipher by limiting the width of the F-function to half the size of the state, the latter still needs to be carefully thought to avoid weaknesses.

F-function designs sometimes share similarities with SPN (e. g., in the case of DES). They are made of simple components, each aimed at a simple goal, which when merged together provide the desired security property of a block cipher.

The round function of a SPN is usually composed of 3 steps: a round key addition, the application of possibly multiple Substitution box (S-box) in parallel, and the application of large Permutation box (P-box). Figure 2.6 illustrates two rounds of a 16-bit SPN.

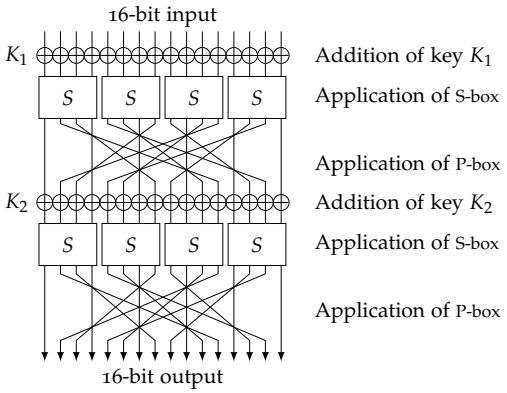


Figure 2.6: A 2-round 16-bit Substitution Permutation Network.

The S-box application is also called the non-linear layer, its role is to ensure that local modifications of a plaintext are changing the neighboring bits.

The P-box application is also called the linear layer which consists often of bit shuffling, rotations, or matrix applications. This layer aims to propagate the local changes of an S-box's output to the full state.

This idea of small local modification and propagation was described as *confusion* and *diffusion* by Shannon in 1945 [Sha45].

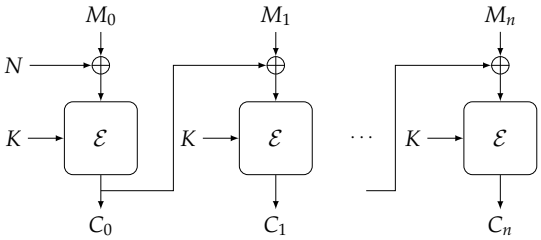
Adding such structure to a block cipher helps to analyze it and to evaluate its security strength as we only need to pay attention to individual simple components.

PADDING.

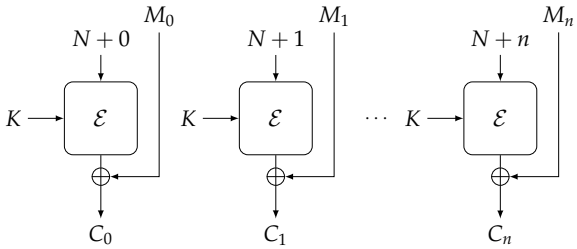
As a message M may be shorter than b bits, it is necessary to use a padding scheme, in other words we need to append a bit-string *pad* such that $M \| pad$ has a length of b bits. For example *pad* can be made of $b - |M|$ bits of value '`0`', we call this a *zero padding scheme*; it is often shortened as $M \| $'`0`*' where '`0`*' represent as many bits as necessary to fill the block.

Zero padding schemes are often considered insecure as they do not allow distinguishing between outputs ending with null bytes: zero padding to 4 bytes `0x00 00` and `0x00` will yield to the same result: `0x00 00 00 00`. For this reason, it is usually preferred to use a reversible scheme, e. g., pad with a byte `0x01` followed by as many `0x00` as necessary.

ENCRYPTION SCHEMES.
Similarly, a message M may also be longer than b bits, this is why it is necessary to use an encryption mode on top of a block cipher. The idea is to produce a padded message M' such that its length fits a multiple of b. M' can then be encrypted by blocks of b bytes $(M_0, M_1, \dots M_n)$. This process can take multiple forms as illustrated in Figure 2.7. Furthermore, to avoid the repetition of ciphertext for the same combination of message and key we use a random message number N. Additionally, notice that the Counter (CTR) mode of encryption allows us to transform a block cipher into a stream cipher.



The Cipher Block Chaining (CBC) mode of encryption.



The CTR mode of encryption.

Figure 2.7: Two examples of modes of encryption.

DEFINITION 2.2.4. *An encryption scheme is defined by a couple of functions: an encryption function $\mathcal{E}$ and a decryption function $\mathcal{D}$.*

$$\begin{aligned} \mathcal{E} : \mathbb{F}_2^n \times \mathbb{F}_2^k \times \mathbb{F}_2^* &\to \mathbb{F}_2^* \\ N, K, P &\mapsto C \end{aligned}$$

Given a n-bit nonce N, k-bit key K and a plaintext P of arbitrary length, $\mathcal{E}$ produces a ciphertext C.

$$\begin{aligned} \mathcal{D} : \mathbb{F}_2^n \times \mathbb{F}_2^k \times \mathbb{F}_2^* &\to \mathbb{F}_2^* \\ N, K, C &\mapsto P \end{aligned}$$

Similarly, given a n-bit nonce N, a k-bit key K and a ciphertext C of arbitrary length, $\mathcal{D}$ retrieves the plaintext P.

Encryption schemes require a similar property as the block ciphers: for all keys K, nonces N, and plaintexts P, $\mathcal{D}(N, K, \mathcal{E}(N, K, P)) = P$. Without a mode of encryption, a combination plaintext and key always yield to the same ciphertext, this leads to attacks where an adversary could guess or re-use (part of) a message due to this repetition property. By using a nonce, we avoid such weakness.

Note that in place of *Nonce*, the literature may sometimes use the terms *Initial Value (IV)* or *Diversifier*. Those three serve the same goal, i. e., avoid the repetition of a ciphertext for the same combination of message and key. On one hand, an IV has a size that is generally tied to the width of the block cipher, while the diversifier is a more generic term. On the other hand, the nonce —NUMBER used only ONCE— emphasizes non-repetition by using for example an incrementing counter for each subsequent encryption.

HASH FUNCTIONS.
Similarly to an encryption scheme, we also define a hash function as follows.

DEFINITION 2.2.5. *A t-bit hash function is a function*

$$\begin{aligned} \mathcal{H} : \mathbb{F}_2^* &\to \mathbb{F}_2^t \\ M &\mapsto H \end{aligned}$$

which maps an arbitrary length message M to a t-bits length value H called either digest or hash.

Such functions are often used to verify the integrity of a document. Indeed, upon reception of a file M, a user will compute its digest H' and compare it to a precomputed value H often obtained prior reception. If $H = H'$ then the user can be reasonably sure that the integrity of the file is preserved.

It is also possible to use hash functions to generate unique IDs, those can later be used to identify a file or a message in e. g., a signature scheme.

MESSAGE AUTHENTICATION CODE.
Protecting the confidentiality and integrity of a message is not the only goal of cryptographic operations, it is also interesting to verify its origin/authenticity; for this we use a Message Authentication Code (MAC).

DEFINITION 2.2.6. *A t-bit Message Authentication Code is a function*

$$\begin{aligned} H : \mathbb{F}_2^k \times \mathbb{F}_2^* &\to \mathbb{F}_2^t \\ K, M &\mapsto T \end{aligned}$$

which given a k-bit key K, maps an arbitrary length message M to a t-bit length tag T.

The process is similar to the verification of the integrity of a message but uses a pre-shared secret key K to compute a candidate tag T'. If the tags does not match ($T' = T$) then we can assume that the message has not been tampered. For this reason, attack on MACs are called forgeries.

Most MAC functions make use of a block cipher. For example, we have CBC-MAC [Sta85] (Figure 2.8), which was standardized as Data Authentication Algorithm (DAA) in FIPS 113 and is now retired (for being broken). It was basically the CBC mode of encryption with DES, but it only returns the last ciphertext as a tag.

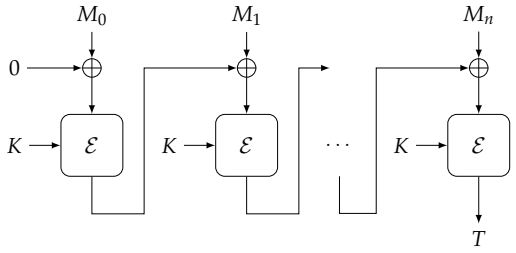


Figure 2.8: CBC-MAC as defined in FIPS 113 [Sta85].

Other well-known examples of MAC are: Poly1305 [Ber05b] which makes use of the AES, and the Galois Message Authentication Code (GMAC) [NIS07].

Not all MAC rely on a block cipher: by combining a hash function with a key, it is also possible to create a MAC.

AUTHENTICATED ENCRYPTION SCHEME WITH ASSOCIATED DATA. By combining a MAC with an encryption scheme we define the following.

DEFINITION 2.2.7. *An AEAD is a couple of functions: an encryption function E and a decryption function D.*

$$E : \mathbb{F}_2^k \times \mathbb{F}_2^n \times \mathbb{F}_2^* \times \mathbb{F}_2^* \to \mathbb{F}_2^* \times \mathbb{F}_2^t$$
$$K, N, A, M \mapsto C, T$$

which given a k-bit key K, a n-bit nonce N, associated data A of an arbitrary length a, and a message M of arbitrary length m produces a ciphertext C of length m and a t-bit tag T,

$$D : \mathbb{F}_2^k \times \mathbb{F}_2^n \times \mathbb{F}_2^* \times \mathbb{F}_2^* \times \mathbb{F}_2^t \to \mathbb{F}_2^* \cup \{\perp\}$$
$$K, N, A, C, T \mapsto M \text{ or } \perp$$

which given a k-bit key K, a n-bit nonce N, associated data A of an arbitrary length a, a ciphertext C of arbitrary length, and a t-bit tag T produces M if the tag T is verified, $\perp$ otherwise.

2.2.2 *Security notions*

SECURITY CLAIM, SECURITY STRENGTH AND COMPLEXITY.
When authors publish a cryptographic primitive, they must provide a *security claim*, in other words the minimum success probability required to break their scheme.

This security claim serves at the same time as an assurance for the users, and as a challenge to fellow cryptographers. Those will try to break the scheme by finding an attack with a higher probability than the claim.

A claim is often referred as the *security strength* of a cryptographic primitive, and it is tied to the complexity of the best attack against it. This notion encapsulates the amount of data to be gathered (we call it data complexity) and/or the amount of computation/number of queries (also called time complexity) required to successfully perform the attack. For 2^s operations required to break the primitive, we say the claimed security strength is of s bits.

As no attacks are known upon publication of a primitive, the default security claim is the exhaustive key search: to try iteratively all the possible keys until the correct one is found. For a n-bit long key, this would require at most 2^n queries and in average $\frac{2^n}{2} = 2^{n-1}$ queries. The complexity of such attack is therefore 2^{n-1} and provide a security strength of $n - 1$ bits.

Due to its 128-bit key length, AES-128 has a claimed security strength of 127 bits. This claim was broken by Bogdanov et al. [BKR11] when they provided a key recovery attack on AES-128 with computational complexity $2^{126.1}$, and later slightly improved by Tao and Wu [TW15] with time complexity $2^{126.01}$ and data complexity 2^{72}. As a result, AES-128 could be considered "broken". However, as Bogdanov et al. stated, their *"attacks are of high computational complexity,"* and they *"do not threaten the practical use of AES in any way."*. Indeed, performing a simple exhaustive key search using the dedicated AES instructions available on most modern CPUs would be significantly faster than doing this computational attack.

SECURITY OF HASH FUNCTIONS.
While the strength of a block cipher is linked to the size of the key, the security claim of a hash function is often tied to the length of the digest. In order to estimate the security of a hash construction, we use a random oracle $\mathcal{RO}$.

DEFINITION 2.2.8. *A random oracle, denoted $\mathcal{RO}$, is an ideal cryptographic primitive which generate random outputs for each query it gets (under the restriction that an input already queried will return the same response).*

When two different inputs of a hash function (or of a random oracle) produce the same output, a collision is found. The difficulty to produce

such event is used to define the security strength for a hash function. By replacing the hash function by a random oracle, we determine the upper bound for such security claims.

Assuming a random oracle producing a n-bit output, the birthday paradox tells us that on average $\sqrt{2^n} = 2^{n/2}$ queries are needed to produce a collision [Yuv79]. From this bound we define the generic collision resistance of a hash function as follows.

DEFINITION 2.2.9. *The collision resistance measures the difficulty of finding two inputs a and b such as $a \neq b$ and $H(a) = H(b)$. The generic attack complexity is $2^{n/2}$ (birthday paradox).*

But sometimes, an adversary may want to replace a document whose digest T has already been precomputed. She needs to find another message M' such that the hash function produces the same tag $T = H(M')$, in other words, find a preimage of the tag. The complexity of such attack is called *preimage resistance*.

DEFINITION 2.2.10. *The preimage resistance measures the difficulty of finding b for a given c such as $H(b) = c$. The complexity of finding a preimage is at most 2^n.*

If, instead of being given a tag T, the original message M is provided (such as $H(M) = T$), finding M' is called a second-preimage. Likewise, the complexity of such attack is called *second-preimage resistance*.

DEFINITION 2.2.11. *The second-preimage resistance measures the difficulty to find b for a given a such as $a \neq b$ and $H(a) = H(b)$. The complexity of finding a (second) preimage is at most 2^n.*

Having briefly described basic cryptographic primitives and the security notions attached to it, we now turn our focus to the sponge constructions which serve as a base in the design of GIMLI (see Chapter 4).

2.3 SPONGE CONSTRUCTIONS

In 2007 Bertoni et al. designed a *sponge functions* [Ber+07; Ber+08a]; such functions make use of cryptographic permutation P applied on a state of b bits. The state consists of an inner part of c bits, also known as the *capacity* of the sponge, and of an outer part of r bits, known as the *rate*. In its simplest form, starting from a zeroed state, the construction is composed of two phases:

- the absorbing phase is processing the input by blocks of r bits, XORing them into the outer part of state;
- and the squeezing phase is extracting output blocks of r bits.

Each block absorption or extraction is interleaved with a call to the cryptographic permutation P.

Note that the c bits constituting the inner part of the sponge are never output during the squeezing phase. Figure 2.9 illustrate the two phases of a sponge construction with the interleaving calls to P. As a result, the inner part of the sponge in only modified by when P is applied, preventing direct tampering of its value.

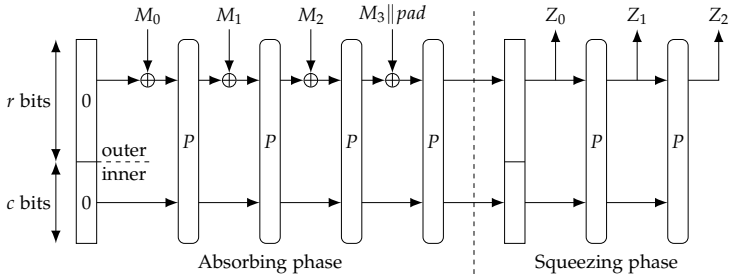


Figure 2.9: Basic sponge construction processing four message blocks $M_0 \| \ldots \| M_3$ and returning three blocks $Z_0 \| Z_1 \| Z_2$.

Similarly to encryption schemes the input is processed by blocks of r bits, it is therefore necessary to use a padding scheme. For this reason a sponge function is described as SPONGE$[P, pad, r]$ where P is the b-bit permutation, r is the rate and pad is the padding scheme. Finally, the capacity $c = b - r$ is inferred from b and r.

With the construction described in Figure 2.9, it is easy to see how it can be exploited to create a hash function or a MAC (by first absorbing the key before processing the message).

In [Ber+11a], Bertoni et al. consider the bound to distinguish a sponge construction using a random permutation P from a random oracle. They notice that it is mainly determined by the size of the inner state (the capacity c) and define the *flat sponge claim* with the following formula.

DEFINITION 2.3.1. *Given a capacity c_{claim}, the success probability of any attack should be not higher than the sum of that for a random oracle and $1 - exp(N^2 2^{-(c_{claim}+1)})$, with the workload of the attack having the computational equivalent of N calls to P (or its inverse P^{-1}).*

2.3.1 *In Hash Functions*

In order to better understand the security strength of a sponge construction with an n-bits output and c-bit capacity, it is easier to illustrate how to generate collisions out of it. To do so the first approach is to target the final n bits extracted, and by using the birthday bound

we know this requires an effort of $2^{n/2}$. The second approach is to find a collision in the inner part (also called inner-collisions) and append a message to cancel remaining the difference in the outer part of the sponge. This second attack has a computational complexity of $2^{c/2}$.

Figure 2.10 illustrates such attack. An adversary will first find M_0 and M_1 such that $M_0 \neq M_1$ and their absorption leads to a state after the application of the permutation such that there is a collision in the inner part. Note that M_0 and M_1 may be of different length or composed of multiple blocks. As the adversary is working in full knowledge of the state, she can produce two messages blocks to cancel the differences in the outer part of the sponge and by doing so, produce a full state identity between the two messages. It is trivial then to produce colliding output from such state.

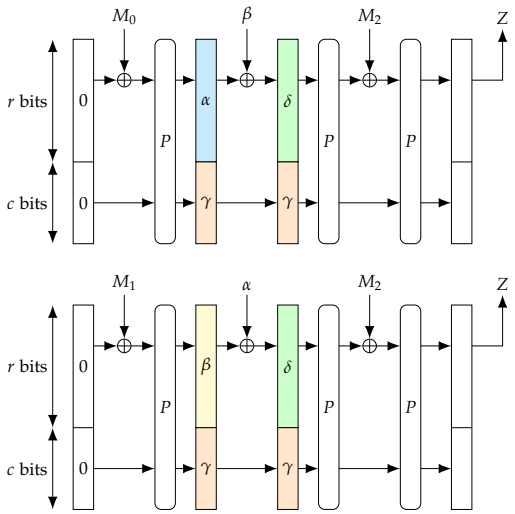


Figure 2.10: Exploiting inner-collisions to generate colliding outputs.

DEFINITION 2.3.2. *For a sponge construction with a capacity c bits and an n-bit output, the security strength against collisions is* $min(c/2, n/2)$ *bits.*

This *security strength*, represents the complexity of a *generic* attack (i. e., an attack which does not take advantage of weaknesses of the permutation used) against a sponge construction using a random permutation. As a result, this gives us a maximum bound on the security claim.

We now turn our focus on preimage resistance, from Definition Definition 2.2.10, we know that a n-bit output hash function will require at most 2^n queries. As with the collision resistance, it is possible

to target inner-collisions. Figure 2.11 illustrate such attack against an output Z, where we use a Meet-In-The-Middle (MITM) approach to produce inner-collisions.

On one hand we generate multiple couple values (M, δ_1), on the other hand we attempt to perform a state recovery from the output Z, in other words we try to guess a value σ such that $Z||\sigma = P(\beta||\delta_2)$. Because P is a permutation, we make use of its inverse P^{-1} and generate candidate values $\beta||\delta_2$.

The underlying idea is to find M and σ such that $\delta_1 = \delta_2$. Upon discovery, the following steps are the same as for a collision search, we craft the message $M||(\alpha \oplus \beta)$ and generate the requested Z.

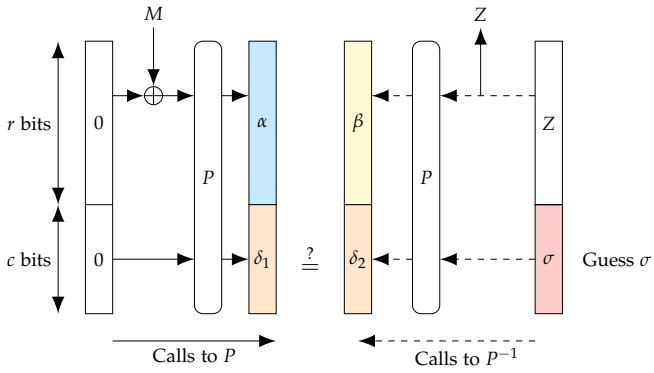


Figure 2.11: Targeting a preimage of Z by finding inner-collisions with a MITM approach.

This is similar to finding collisions, thus with the birthday paradox we know that this attack requires an effort of $2^{c/2}$ queries.

DEFINITION 2.3.3. *For a sponge construction with a c-bit capacity and an n-bit output, the preimage resistance has a security strength of is $min(c/2, n)$ bits.*

Note that this generic attack is also applicable in second-preimage search. As a result, the security strength directly translates to the second-preimage resistance.

DEFINITION 2.3.4. *For a sponge construction with a c-bit capacity and an n-bit output, the second-preimage resistance has a security strength of is $min(c/2, n)$ bits.*

The iterative nature on a state of the sponge construction makes it easy to produce an output of arbitrary length, and thus define a eXtendable Output Function (XOF) [Per14]. Such function takes two inputs: a message and an output length. This allows us to create a

stream-cipher: we first absorb a Nonce N and key K and extract a keystream by blocks of r bits (Z_i) interleaved with calls to the permutation P. Output blocks are then XORed into each message blocks (M_i), producing a ciphertext (C_i). Figure 2.12 illustrates this process.

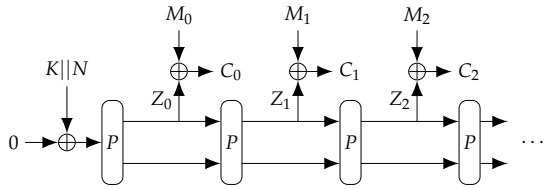


Figure 2.12: Using a sponge construction as a stream cipher.

Stream cipher only aims to protect the confidentiality of the message, it is good practice to additionally produce a MAC so that the integrity of the plaintext or ciphertext can be verified.

As we already presented in Definition 2.2.7, this combination defines an AEAD scheme. We now describe here how sponges mode are used to produce such constructions.

2.3.2 *Duplex constructions for AEAD Schemes*

In order to produce an authenticated tag, we initialize the state with a nonce N and key *key*; such constructions are called KEYED-SPONGE [Ber+11c]. Then similarly to hashing, the Associated Data (AD) (e. g., an HTTP header) and messages are absorbed into the state by blocks of r bits. Likewise, a ciphertext is produced upon each message-block absorption: the outer part of the state is seen as a *keystream* and XOR it with the message. In other words, the keystream used for the block M_i is the "*digest*" of the key and the previous blocks $M_0 \dots M_{i-1}$. Once the last message block has been processed, we squeeze the sponge one last time to retrieve the authentication tag T. This construction is called SPONGEWRAP [Ber+11b] and illustrated in Figure 2.13.

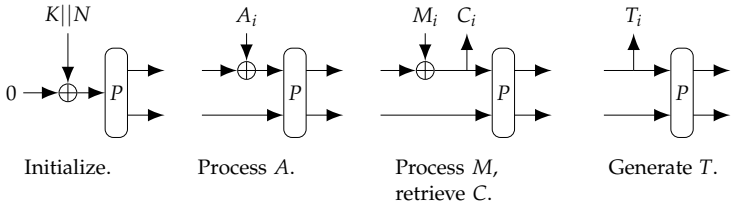
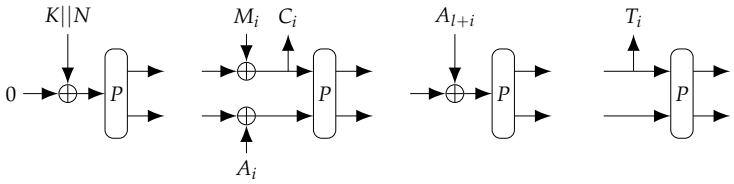


Figure 2.13: An example of Authenticated Encryption sponge construction: SPONGEWRAP (with N, simplified).

This mode was subsequently improved by Sasaki and Yasuda [SY15] by absorbing the additional data into the inner part of the keyed sponge as it does not produce ciphertext. To provide speed-ups, it is also possible to apply full-state absorption: as proposed by Bertoni et al. [Ber+12], the entirety of the state is used to process data. This idea, pictured in Figure 2.14, was later proven secure by Mennink, Reyhabinitabar and Vizár [MRV15] and improved by Daemen, Mennink and Van Assche [DMA17].



Initialize. Process M and A in parallel. Process A Generate T.

Figure 2.14: Another example of Authenticated Encryption sponge construction: FSW (Full-state SpongeWrap, simplified).

2.4 KECCAK & SHA-3

In August 2015, the National Institute of Standards and Technology (NIST) published the new FIPS 202 [NIS15] for the hash function family known as SHA-3. It makes use of a sponge construction and a permutation Keccak-f [Ber+08b] which we describe in detail below. Then we describe the different variations proposed by NIST and their respective security claims.

2.4.1 Keccak-f

Keccak-f [Ber+11d] is a family of seven permutations denoted by Keccak-$f[b]$ with a bit-width $b \in \{25, 50, 100, 200, 400, 800, 1600\}$. The state of Keccak-$f$ is organized as a parallelepiped of dimension $5 \times 5 \times w$ where $w \in \{1, 2, 4, 8, 16, 32, 64\}$. Each bit is located by its coordinates (x, y, z) (See Figure 2.15). A bit is active if it has value 1 and passive otherwise.

Given this representation of a state, we have:

- in orange in Figure 2.15, a set of 25 bits with fixed z coordinate is called a *slice*,
- in blue in Figure 2.15, a set of w bits with fixed (x, y) coordinate is called a *lane*,
- in cyan in Figure 2.15, a set of 5 bits with fixed (y, z) coordinates is called a *row*,
- in red in Figure 2.15, a set of 5 bits with fixed (x, z) coordinates is called a *column*.

For a better visualization of active bit positions inside the state, a lighter representation will also be used (see Figure 2.16).

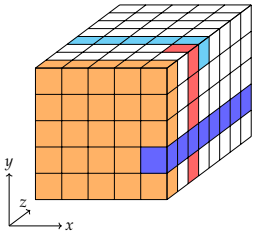


Figure 2.15: State structure of KECCAK-$f[200]$.

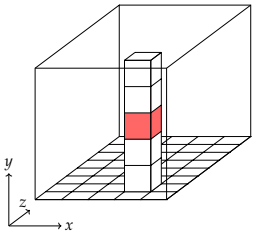


Figure 2.16: Light representation of the state structure where the bit $(3, 2, 1)$ is active.

KECCAK-f is composed of $12 + 2 \times log_2(w)$ iterations of a round permutation composed of 5 transformations:

$$f = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

θ (THETA).
θ is a linear mixing layer which operates on columns.

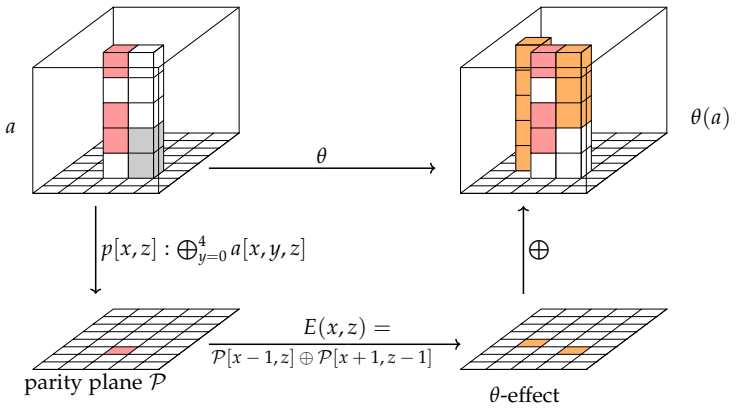


Figure 2.17: θ.

The parity function (p) computes the parity plane $\mathcal{P}$ as the sum over the columns. From this plane the θ-effect can be deducted before being added back to the state.

ρ (RHO) AND π (PI).
Both ρ and π operate on lanes. While ρ is a bit-wise cyclic shift (Figure 2.18), π transposes the position of the lanes (Figure 2.19).

Because θ operates on columns, it is important to separate every bit of a column after one application to avoid the appearance of patterns. Therefore, π guarantees that all bits in a column are evenly spread over the slice. The role of ρ is make sure that differences are spread over slices. These two transformations aim to propagate the modifications applied to the state.

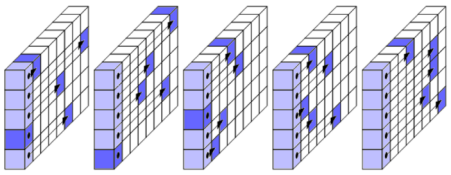


Figure 2.18: The ρ transformation.

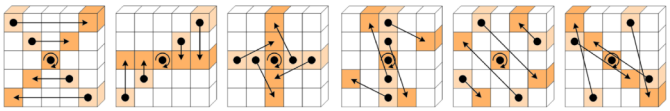


Figure 2.19: The π transposition.

χ (CHI).
χ is the only non-linear mapping in KECCAK-f. Without it, the round function of KECCAK-f would be linear (which would imply a big weakness against differential cryptanalysis, Section 2.5). χ has an algebraic degree of 2 and operates on rows. Each application of χ could be seen as the application of a 5-bit S-box.

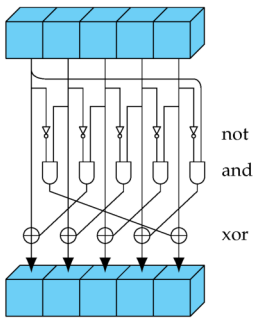


Figure 2.20: χ.

ι (IOTA).
ι consists of the addition of round constants to the lane $(0,0)$ and is aimed at disrupting symmetry. Without it, the round function would be translation-invariant in the z direction and all rounds would be equal making KECCAK-f subject to attacks exploiting symmetry such as slide attacks.

2.4.2 *SHA-3*

While FIPS-202 defines the permutation KECCAK-$f[b]$ for a width $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, only KECCAK-$f[1600]$ is used in the standardized hash function and eXtendable Output Function (XOF). FIPS-202 defines KECCAK[c] as a sponge on top of KECCAK-$f[1600]$ with a c-bit capacity and a *multi-rate padding*. The latter is $pad10^*1$: it returns the input followed by '1', then by a (possibly empty) string of '0's, and finally by a '1'.

The standard defines four hash functions: SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, with respective output length 224, 256, 384 and 512 bits; and two XOFs: SHAKE128 and SHAKE256 where the output length is free. In order get the preimage resistance at the required level, the capacity must be of 448 bits (and respectively 512, 758, 1024 bits). Likewise, the XOFs will have a capacity of 256 and 512 bits.

Note that SHA-3-256 and SHAKE256 both uses KECCAK[512], thus for a same message M, it would not be possible to differentiate their output truncated to 256 bits. This is why FIPS-202 defines a different suffix appended to the input before the multi rate padding: the hash functions will use a '01' suffix (Equation (2.1)), and the XOFs will use '1111' suffix (Equation (2.2)).

$$\text{SHA-3-256}(M) = \text{KECCAK}[512](M \| \text{'01'}) \tag{2.1}$$

$$\text{SHAKE256}(M) = \text{KECCAK}[512](M \| \text{'1111'}) \tag{2.2}$$

Table 2.1 summarizes the differences between the functions standardized in FIPS 202 while Table 2.2 sums up the security claim of the SHA-3 functions.

Table 2.1: Capacity, rate, output length, and suffix of the SHA-3 functions.

Algorithm	capacity c	rate r	output length	suffix
SHA-3-224	448	1152	224	'01'
SHA-3-256	512	1088	256	'01'
SHA-3-384	768	832	384	'01'
SHA-3-512	1024	576	512	'01'
SHAKE128	256	1344	$*$	'1111'
SHAKE256	512	1088	$*$	'1111'

Table 2.2: Security claim of the SHA-3 functions.

Algorithm	output size	Security strength in bit	
		collision	preimage
SHA-3-224	224	112	224
SHA-3-256	256	128	256
SHA-3-384	384	196	384
SHA-3-512	512	256	512
SHAKE128	n	$min(n/2, 128)$	$min(n, 128)$
SHAKE256	n	$min(n/2, 256)$	$min(n, 256)$

2.5 DIFFERENTIAL CRYPTANALYSIS

Differential cryptanalysis is a statistical attack on the DES presented at Crypto 90 by Biham and Shamir [BS90; BS91]. Their approach uses a distinguisher based on difference propagation with relatively high probability, leading in the collection of 2^{47} chosen plaintexts. After some data filtering, they compute the encryption key by analyzing 2^{36} plaintexts in 2^{37} time, effectively breaking DES faster than brute force [BS92]. Note that this break claims does not differentiate between the data complexity and the computational complexity. As a result, a brute force attack with dedicated hardware as performed in 1999 would be more efficient [FLG98].

2.5.1 *Differences*

This method focuses on pairs of inputs x_P and x'_P, and their *difference*:

$$\Delta_P = x_P \oplus x'_P. \tag{2.3}$$

When x_P and x'_P are inputs of a function f (such as a round function, a S-box...) with a difference Δ_P, we study the resulting difference Δ_T of their output $x_T = f(x_P)$ and $x'_T = f(x_T)$:

$$\begin{aligned} \Delta_T = x_T \oplus x'_T &= f(x_P) \oplus f(x'_P) && (2.4)\\ &= f(x_P) \oplus f(x_P \oplus \Delta_P). && (2.5) \end{aligned}$$

We refer to a pair of differences Δ_P and Δ_T as a *differential* $(\Delta_P \Rightarrow \Delta_T)$ (see Figure 2.21).

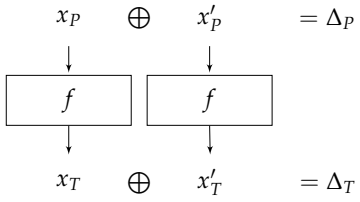


Figure 2.21: A differential $(\Delta_P \Rightarrow \Delta_T)$.

By studying how differences behave when applied to different kind of functions, we notice that:

- the key addition, i. e., the XOR with a key, does not change the difference Δ in the pair of plaintexts (x, x'),

$$\begin{aligned} (x \oplus K) \oplus (x' \oplus K) &= (x \oplus K) \oplus ((x \oplus \Delta) \oplus K) \\ &= (x \oplus K) \oplus (x \oplus K) \oplus \Delta \\ &= \Delta \end{aligned}$$

- a linear application L (e. g., a multiplication by a boolean matrix) on the inputs only change the shape of the difference Δ in the pair of plaintexts (x, x').

$$\begin{aligned} L(x) \oplus L(x') &= L(x) \oplus L(x \oplus \Delta) \\ &= L(x) \oplus L(x) \oplus L(\Delta) \\ &= L(\Delta) \end{aligned}$$

- Substitution boxes are known to be non-linear. Even if we know the value of the XOR of inputs, we cannot know with certainty the XOR of the outputs as several options are possible, aside from the case of equal inputs leading to equal outputs. However, we notice that for a given input difference not all output differences are possible, and some are much more frequent than others.

With those properties in mind, we study the probability of differentials, and when possible, they are usually computed and summarized in a *differential distribution table (DDT)* [BS90].

2.5.2 *Differential Probability and Weight*

For a cryptographic function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$, even if the input value x is unknown, we may be able to derive some statistical information about difference propagation. For example, considering all possible inputs x, we can infer the probability of each differential $(\alpha \Rightarrow \beta)$ by computing the cardinality of the solution set $S(\alpha, \beta)$.

$$S(\alpha, \beta) = \{x \in \mathbb{F}_2^n : \beta = f(x) \oplus f(x \oplus \alpha)\}. \tag{2.6}$$

If the cardinal $\#S(\alpha, \beta)$ is 0, it means that the input difference α will never lead to the output difference β for all possible input x, as a result we call $(\alpha \Rightarrow \beta)$ an *impossible differential*. Otherwise, a differential is called *possible*.

DEFINITION 2.5.1. *For a function f with domain $\mathbb{F}_2^n$, given a differential $(\alpha \Rightarrow \beta)$, we denote the Differential Probability (DP) as:*

$$DP(\alpha, \beta) = \frac{\#\{x \in \mathbb{F}_2^n : \beta = f(x) \oplus f(x \oplus \alpha)\}}{2^n} = \frac{\#S(\alpha, \beta)}{2^n} \tag{2.7}$$

Another way to quantify the DP is to consider the weight of a differential [Dae95]:

DEFINITION 2.5.2. *For a function f with domain $\mathbb{F}_2^n$, the weight can also be seen as :*

$$w(\alpha \overset{f}{\Rightarrow} \beta) = n - \log_2 \#S(\alpha, \beta) \tag{2.8}$$

Note the weight is undefined for impossible differentials as $\#S(\alpha, \beta) = 0$.

A simpler relation between the DP and its weight w is illustrated in Equation (2.9).

$$DP = \frac{1}{2^w} \tag{2.9}$$

As a result, the greater the weight, the harder it will be to exploit the differential.

2.5.3 *Trails*

While a single differential may cover an S-box or a simple round function, it is possible to combine them in order to propagate the statistical information through the multiple rounds of the cipher. For example, given the differentials $(\alpha \Rightarrow \beta)$ and $(\beta \Rightarrow \gamma)$, we build the trail $(\alpha \Rightarrow \beta \Rightarrow \gamma)$ with $\mathrm{DP}(\alpha, \beta, \gamma)$ (see Figure 2.22).

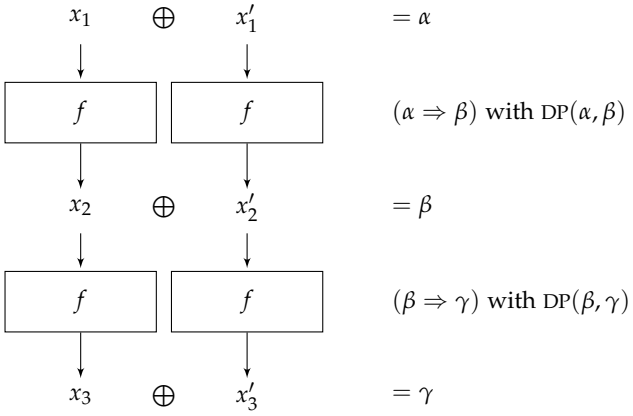


Figure 2.22: By combining two differentials $(\alpha \Rightarrow \beta)$ and $(\beta \Rightarrow \gamma)$, we build a trail $(\alpha \Rightarrow \beta \Rightarrow \gamma)$.

The probability of $DP(\alpha, \beta, \gamma)$ can be seen as the probability that the events $(\alpha \Rightarrow \beta)$ and $(\beta \Rightarrow \gamma)$ with respective probability $DP(\alpha, \beta)$ and $DP(\beta, \gamma)$ happen at the same time. As a result for independent differentials, the probability of the trail $(\alpha \Rightarrow \beta \Rightarrow \gamma)$ is

$$DP(\alpha, \beta, \gamma) = DP(\alpha, \beta) \times DP(\beta, \gamma).$$

In practice the round function f makes use of a key K, as a result the differential probability $DP(\alpha, \beta)$ depends on that key and becomes a stochastic variable; in such instance we consider the *Expected Differential Probability (EDP)* of the set of functions $\{f_K\}$, and by using the *Hypothesis of Stochastic Equivalence* [LMM91], we assume that

$$DP(\alpha, \beta) \approx EDP(\alpha, \beta).$$

We extend this notion to trails: for $\chi = (\Delta_0 \Rightarrow \ldots \Rightarrow \Delta_n)$, we infer the *Expected Differential Probability* of the trail as the product of the EDP composing that trail:

$$EDP(\chi) = \prod_{i=0}^{n-1} EDP(\Delta_i \Rightarrow \Delta_{i+1}).$$

From this expression, we extend the notion of weight to trails.

DEFINITION 2.5.3. *The weight of a trail is the sum of the weight of the differentials that compose this trail.*

As a multi-round differential (Δ_0, Δ_n) is made of multiple trails, its Expected Differential Probability is the sum of the trails EDP:

$$EDP(\Delta_0, \Delta_n) = \sum_{\Delta_1} \cdots \sum_{\Delta_{n-1}} EDP(\Delta_0, \Delta_1, \ldots, \Delta_{n-1}, \Delta_n)$$

As a consequence, the Expected Differential Probability of a trail gives a direct lower bound of the Expected Differential Probability of the corresponding multi-round differential.

In place of *trails*, the literature may use *paths*, *differential characteristics* [BS90] or *characteristics*. Nevertheless, the idea stays the same: propagating differences through multiple rounds of the cryptographic function.

2.5.4 *Exploiting Trails*

By combining differentials into a trail, it may be possible to build what is called an *iterative differential*, a differential whose input difference is equal to its output difference. Discovering such a property usually reveals a major weakness in the design of the cryptographic function being analyzed. For example, in order to break the DES, Shamir and Biham [BS92] used the 2-round iterative differential $((\psi, 0) \Rightarrow (\psi, 0))$ where $\psi =$ `0x16900000`. The base differential $(\psi \Rightarrow 0)$ has a DP of $\frac{1}{234}$ and is easily extended to $((\psi, 0) \Rightarrow (\psi, 0))$ (see Figure 2.23).

By finding a low weight trail traversing multiple rounds of a block cipher or permutation, it is possible to perform a key recovery attacks.

OR ATTACKS. Given block cipher E_K and assuming we found a low weight n-round characteristic χ such as $(\Delta_0 \Rightarrow \Delta_1 \Rightarrow \ldots \Rightarrow \Delta_n)$, we can apply a oR attack [BS90; Biho4; Biho5] in two steps.

First we consider the differential $\delta = (\Delta_0 \Rightarrow \Delta_n)$ and generate a large number of message couple (M, M') such that $M \oplus M' = \Delta_0$. Then we collect their respective ciphertext (C, C') such that $C = E_K(M)$, $C' = E_K(M')$. We only keep the ones following the equation $C \oplus C' = \Delta_n$, and discard the others as we are only interested in the pairs satisfying δ.

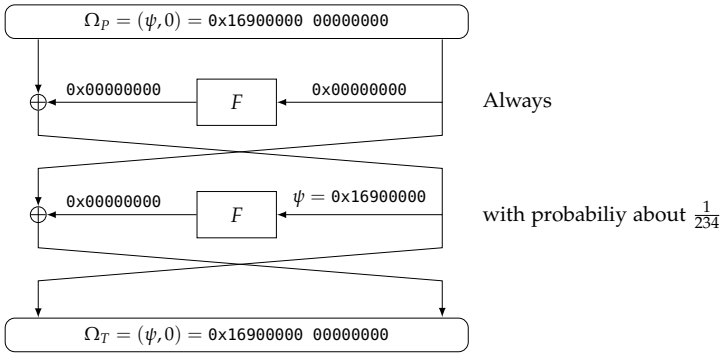


Figure 2.23: The 2-round iterative differential with a probability of $\frac{1}{234}$.

Under the assumption that most of the pairs following δ also follow χ (i. e., δ is dominated by a single trail), their last round are likely to satisfy $\delta' = (\Delta_{n-1} \Rightarrow \Delta_n)$. We take each S-box or other non-linear operations of the last round and use δ' to derive their individual differentials $(\alpha_i \Rightarrow \beta_i)$. By using the solution set $S(\alpha_i, \beta_i)$, we recover possible intermediate values during the encryption. If we combine those pieces of information with the values of C and C', we learn segments of the last round key K_n and thus reduce the size of the key space.

Note that due to the nature of the solution set $S(\alpha, \beta)$, multiple equivalent keys will be produced for each S-box. By using multiple ciphertext pairs, and intersecting the sets of key candidates, we are able to speed-up the key recovery process.

1+R ATTACKS. Instead of attacking the last key by using output differences, it is also possible to attack a previous subkey of a n-round cipher. Such methods are called 1R (or 2R, ...) attacks and need a trail χ for $n-1$ (respectively $n-2$ ...) rounds with low weight.

Considering a low weight trail $\chi = (\Delta_0 \Rightarrow \ldots \Rightarrow \Delta_{n-1})$ and the differential $\delta = (\Delta_0 \Rightarrow \Delta_{n-1})$, we encrypt a large number of pairs of plaintexts satisfying Δ_0. Similarly to a 0R attack, we decompose the last difference Δ_{n-1} into fragments α_i matching the next non-linear operations, and we use them to target selected bits of the encryption key. By combining those bits into partial keys k_j we are able to recover the correct key by a simple statistical analysis. Indeed, for each collected ciphertext pair (C, C') we decrypt the last round with the partial keys k_j and compute the resulting difference α'_i. If $\alpha_i = \alpha'_i$, we increase the score of k_j. In the end, the partial key with the highest score gives us the desired bits of the encryption key. With this knowledge, the rest of the key-space is left to brute force. Figure 2.24 illustrates such attack with a 2-round differential (0x00A0 $\Rightarrow$ 0x0100).

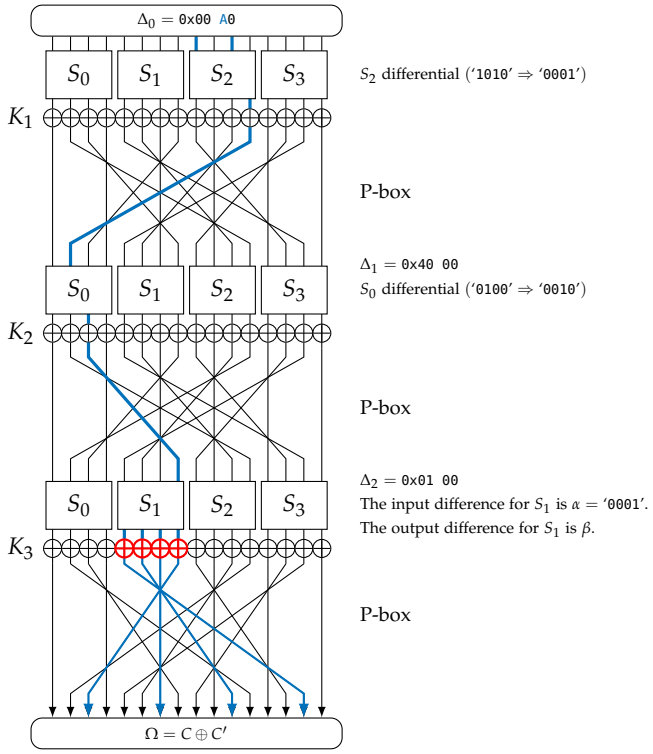


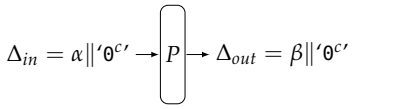
Figure 2.24: A 2-round trail ($\Delta_0 \Rightarrow \Delta_1 \Rightarrow \Delta_2$) over a 3-round SPN leading to the recovery of 4 bits of K_3 (highlighted in red).

The basic assumption for such attack is the *wrong-key randomization hypothesis*: upon decryption of the last round with a wrong partial key, the observed difference will be uniformly random. This means that the desired difference α_i will be no different from all the others. On the other hand, for a correct partial key, the difference α_i will appear significantly more often. The underlying idea is that E_K^{-1} will behave as a random permutation for a wrong key K, but is likely to follow the trail χ with the correct key fragment.

It is possible to speed up the process by using *impossible differentials*. For that we need to distinguish two kinds of pairs of plaintexts: the *right pairs* which satisfy the differential ($\Delta_0 \Rightarrow \Delta_{n-1}$), and the one who does not, namely the *wrong pairs* which we would like to discard as they do not follow χ. For example in Figure 2.24, by computing the difference $\Omega = C \oplus C'$, we can check if the differential ($\Delta_2 \Rightarrow \Omega$) is possible, if not, thus an impossible differential, we can safely discard the pair.

SEARCHING FOR COLLISIONS. Differential cryptanalysis is also applicable to hash functions. Remember that in the case of a sponge construction, it is possible to construct collisions either in the outer part during the squeezing phase or in the inner part during the absorption phase.

For example, given a sponge construction using a cryptographic permutation P, we study the trails of P and try to find one starting from a non-null difference $\Delta_{in} = \alpha \| \texttt{`0}^c\texttt{'}$ and such that the output difference is $\Delta_{out} = \beta \| \texttt{`0}^c\texttt{'}$. Then by absorbing multiple pairs of messages (M, M') such that $M \oplus M' = \Delta_{in}$, we try to find a *right pair* satisfying Δ_{out}. Once discovered it is easy to cancel the remaining difference β and extract as many collisions as desired. This simplified attack is illustrated in Figure 2.25.



Permutation P with a differential $(\Delta_{in} \Rightarrow \Delta_{out})$.

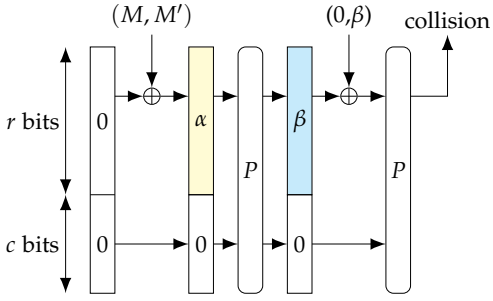


Figure 2.25: Simplified collision generation from a differential trail in a sponge construction.

As a result, the difficulty to find a collision is directly tied to the DP of the trail. An adversary is thus faced to two problems: (1) finding a trail with a weight low enough to be exploitable and with differences satisfying collision conditions, and (2) using this trail to generate the collision.

3 FORMAL REASONING IN A NUTSHELL

This chapter aims to provide the foundations necessary for the understanding of the work presented in Chapter 7. A reader already familiar with Coq and the separation logic in VST may skip directly ahead.

In the following, in Section 3.1, we provide a refresher about logic by first describing the notation used before introducing the intuitionistic logic.

We continue in Section 3.2, where we briefly introduce the Coq theorem prover before turning our attention to the verification of software in Section 3.3. We review the Floyd-Hoare logic and describe its basic rules. Finally, we exhibit its limitations and introduce the separation logic.

Section 3.4 provides a short introduction to the tools that we use later, while Section 3.5 illustrates how such methods are applied in practice.

Note that Chapter 7 of this manuscript is focused on the use in practice of formal verification rather than their development and the theory behind it. As a result some of the notations and ideas presented here have been abbreviated to ease the reader's experience and provide a broad intuition of the field.

For a more thorough introduction into the formal verification world and the Coq theorem prover, we highly recommend reading the Software Foundation series[1] by Pierce et al. [Pie+18b; Pie+18a].

3.1 LOGIC

3.1.1 *Notations*

We adopt the same logical notations as one for binary operations described in Section 2.1. Additionally, we use the symbols $\top$ and $\bot$ to refer to the notions of *True* and *False* (respectively corresponding to the bits '`1`' and '`0`'), and $A \to B$ to denote the implication *A implies B*. Furthermore, the symbol $\vdash$ (called *turnstile*) is read as "infers", or "has for logical consequence". This notation is often used to give a *context* under which a *conclusion* is true, see Equation (3.1).

$$\textit{context/hypotheses} \vdash \textit{conclusion}. \tag{3.1}$$

Using this notation, a theorem is a formula in the form of $\vdash P$ (with an empty left-hand side) and is the conclusion of a valid proof.

1 https://softwarefoundations.cis.upenn.edu/

For example the judgment $A, B, C \vdash E$ (read as "E is true under assumptions A, B, and C") is equivalent to $\vdash (A \wedge B \wedge C) \to E$.

In classical logic, the most commonly used system, a proposition is either *True* or *False*. This results in the following very basic theorem:

THEOREM 3.1.1 (TERTIUM NON DATUR). *The law of excluded middle states that for all propositions P, P is either True or Not True:*

$$\forall P, P \vee \neg P.$$

Similarly, the principle of double negation (Corollary 3.1.2) is easily derived from the law of excluded middle (Theorem 3.1.1).

COROLLARY 3.1.2 (PRINCIPLE OF DOUBLE NEGATION). *The principle of double negations states that for all propositions, if the negation of P is False, then P is True:*

$$\forall P, \neg\neg P \to P.$$

3.1.2 *Intuitionistic Logic*

In 1907, Brouwer [Bro07b; Bro07a] introduced the *intuitionistic logic* where the Truth of a statement is only accepted if one has a proof of the statement.

This idea of exhibiting a proof as an object or giving a method for the creation of such object is the core of intuitionistic logic. Because of this particularity, it is also called the *constructive logic*.

As a result, a proof of:

- $A \wedge B$ is a pair (a, b) where a is proof of A and b is a proof of B.
- $A \vee B$ is a pair (n, p) where $n = 0$ and p is a proof A or $n = 1$ and p is a proof of B.
- $A \to B$ is a function f that converts a proof of A into a proof of B.
- $\bot$ cannot exist.
- $\forall x \in A, B(x)$ is a function f that converts any element a of A into a proof of $B(a)$.
- $\exists x \in A, B(x)$ is a pair (a, b) such that a is an element of A and b is a proof of $B(a)$.
- $\neg P$ seen as $P \to \bot$, is a function f that converts a proof of P into a proof of $\bot$.

As a consequence, the intuitionistic view of proofs is closely tied to functions and types. This was noticed by Curry and Howard [How95] and is known as the Curry-Howard correspondence.

3.2 COQ

The Curry-Howard correspondence [How95] allows us to represent proofs as term. More precisely, the proof is a typed term where the type

displays the formula that is proven. By type-checking the operations inside the term, we can ensure the correctness of the proof.

For example, given a predicate $\forall n \in \mathbb{N}, P(n)$, its proof will be function f which converts an integer n into a term of type $P(n)$, that is, a proof of $P(n)$. This function will have the type: $f : (n : \mathbb{N}) \to P(n)$.

A proof by induction of $\forall n \in \mathbb{N}, P(n)$ will require proving $P(0)$, and to prove $P(n+1)$ under the assumption that $P(n)$ holds. Therefore, the body of f will feature a simple match on n and apply the recursion call in place of the induction hypothesis:

$$
\begin{aligned}
f(n) := \text{match } n \text{ with} \\
&\mid 0 \Rightarrow \text{proof in the case of } 0 \\
&\mid n' + 1 \Rightarrow \text{reduction and call to } f(n') \\
&\text{end.}
\end{aligned}
$$

In 1985, Coquand and Huet developed the Calculus of Constructions [CH88], a type theory representing a constructive higher-order logic following the Curry-Howard correspondence, and which serves as the base for the Coq theorem prover [Coq]. In this formal proof management system, a proof is a type-theoretical term that represents a proof object. To construct proof objects, we use a set of tactics and the Gallina language to interactively create proofs without exposing the complexity of the underlying proof terms.

The proving process of Coq is similar to building a proof tree: the user starts from the bottom and solves the branches and, creates a proof incrementally. The operation is quite similar to asking "why" constantly: "I need to prove B, but I know that $A \to B$, so now I need to prove A."

Additionally, Coq also includes a functional programming language where one can define data types, and write functions for these data types, and execute them. It is worth noting that Coq is not Turing-complete; in particular, all programs written in this formal system must terminate. For example, writing a recursive function requires the recursive call to be on a structurally decreasing argument in order to guarantee termination.

3.3 VERIFYING PROGRAMS

One of the goals of Formal Verification is to be able to prove that the execution of a piece of software respects a set of properties. Those could be safety related, e. g., the software never crashes, or security related, e. g., the software does not leak secret data, or simply that the software is correct: the function it performs is as defined in the specifications.

3.3.1 *Floyd-Hoare Logic*

In 1967 Floyd [Flo67; Flo93], and subsequently in 1969 Hoare [Hoa69], developed what is now known as the *Floyd–Hoare logic*. It is a set of logical rules used to formally prove properties about pieces of code and infer their correctness.

Those rules use triples (known as *Hoare triple*) of the form

$$\{Pre\}\ \texttt{Prog}\ \{Post\}$$

where *Pre* and *Post* are predicates and `Prog` is a fragment of code. The triple is read as "if the precondition *Pre* holds in the initial state, and we execute `Prog`, then the postcondition *Post* will hold in the final state". In the following, we present the rules using an inference bar.

SKIP RULE. Also known as *do nothing* or empty statement, this rule asserts that the state of a program does not change during its execution. Whatever was *True* before remains *True* after. It could be seen as equivalent to an `NOP` instruction.

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ \text{Skip}$$

ASSIGN RULE. This rule states that any predicates that were true for the right-hand side of the assignment are now true for the left-hand side.

$$\frac{}{\{Q[e/x]\}\ \texttt{x} \leftarrow \texttt{e}\ \{Q\}}\ \text{Assign}$$

In the rule x is any variable and e is any expression. The notation $Q[e/x]$ denotes the result of substituting the term e for all occurrences of the free variable x in Q. For example:

$$\frac{}{\{x + 2 = 42\}\ \texttt{y} \leftarrow \texttt{x} + 2\ \{y = 42\}}\ \text{Assign}$$

SEQUENCE RULE. This rule is the foundation of Hoare logic, and is often used to go through each step of a piece of code.

$$\frac{\{P\}\ \texttt{C}_1\ \{Q\} \qquad \{Q\}\ \texttt{C}_2\ \{R\}}{\{P\}\ \texttt{C}_1\ \texttt{;C}_2\ \{R\}}\ \text{Sequence}$$

It is read as: "For two blocks of code $\texttt{C}_1$ and $\texttt{C}_2$, if $\{P\}\ \texttt{C}_1\ \{Q\}$ [is *True*] and if $\{Q\}\ \texttt{C}_2\ \{R\}$ [is *True*], we infer $\{P\}\ \texttt{C}_1\ \texttt{;C}_2\ \{R\}$." When a proof tree is built from bottom to top, in order to prove $\{P\}\ \texttt{C}_1\ \texttt{;C}_2\ \{R\}$, the user will need to give a proposition Q such that $\{P\}\ \texttt{C}_1\ \{Q\}$ and $\{Q\}\ \texttt{C}_2\ \{R\}$.

CONSEQUENCE RULE. Also known as *Postcondition and Precondition weakening*, this rule allows us to simplify goals and is often combined with the Sequence rule.

$$\frac{\{P \to P'\} \qquad \{P'\}\ \texttt{C}\ \{Q'\} \qquad \{Q' \to Q\}}{\{P\}\ \texttt{C}\ \{Q\}}\ \text{Consequence}$$

CONDITIONAL RULE. This rule allows us to prove *if* statements, and generate a subgoal for each case: the condition where B is true and the condition where B is false. However, B should only contain a boolean test and not have side effects, in other cases, it is necessary to apply the Sequence rule and separate the effect from the check.

$$\frac{\{B \wedge P\}\ \mathtt{C}_1\ \{Q\} \qquad \{\neg B \wedge P\}\ \mathtt{C}_2\ \{Q\}}{\{P\}\ \mathtt{if\ B\ then\ C}_1\ \mathtt{else\ C}_2\ \mathtt{endif}\ \{Q\}}\ \text{Cond}$$

WHILE RULE. And finally with this rule, we are able to prove the correctness of loops. For this purpose we use a *loop invariant* P, i. e., a proposition that remains true through the execution of the loop. Similarly to the conditional rule, B cannot have side effects.

$$\frac{\{B \wedge P\}\ \mathtt{C}\ \{P\}}{\{P\}\ \mathtt{while\ B\ do\ C\ end}\ \{\neg B \wedge P\}}\ \text{While}$$

PARTIAL AND COMPLETE CORRECTNESS. The rules presented here allows us to prove the correctness of a program under the assumption that it terminates, such property is called "partial correctness". In other words for $\{P\}\ \mathtt{C}\ \{Q\}$, if P holds in the initial state σ_i, and we execute C in state σ_i, terminating in a state σ_t, then Q holds in σ_t. This implies that $\{P\}\ \mathtt{C}\ \{Q\}$ only says something meaningful in the case that C terminates.

In order to prove the termination of an algorithm, thus the "complete correctness", it is necessary to include a measure over an additional expression and prove that its values strictly decrease with respect to a well-founded relation i. e., there exists a smallest element.

3.3.2 *Separation Logic*

While the Hoare Logic allows us to reason about most programs, it will struggle to provide meaning to code using pointers and arrays. We illustrate this problem with a simple program in C iterating over two arrays (a, b) and summing their values in an output array c (see Code 3.1).

```
1  void A(int* c, int* a, int*b){
2      int i, ai, bi; i = 0;
3      while (i < 16) {
4          ai = a[i];
5          bi = b[i];
6          c[i] = ai + bi;
7          i++;
8      }
9  }
```

Code 3.1: Simple Addition.

An invariant of this while loop must state that a, b, and c are arrays representing sequences α, β and γ such that after i iterations, $\forall j < i, \gamma_j = \alpha_j + \beta_j$. Unfortunately this invariant is not strong enough: if

there is sharing between `a` and `c`, e. g., the address `c` is `a + 1`, the content of `a` will be overwritten upon execution of the loop. The final result will not respect the intended specification. In order to prevent such problems it would be possible to add additional conditions on memory addresses but this quickly expands the complexity of the invariant as the number of variables increases.

To solve this, Reynolds proposed in 2002 *separation logic* [Rey02] which complements Hoare logic with an extension. He considers the store s and the heap h which correspond respectively to the content of the local variables and to the content of the memory. The precondition P and postcondition Q of the Hoare triple are thus split into two parts s_P, h_P and respectively s_Q, h_Q.

$$\{s_P, h_P\} \texttt{ Prog } \{s_Q, h_Q\}$$

In the following, we provide two illustrations of how the heap and the store are working together. In this specific case, the heap has been initialized with two cells of addresses 10 and 11 with respective values 1 and 2. We use fragments of C code to illustrate reading from and writing to the heap.

{ Store,	Heap }	`x = *(y+1)`	{ Store,	Heap }
$y = 10$	10 11		$y = 10$	10 11
	1 2		$x = 2$	1 2

{ Store,	Heap }	`*(y+1) = 7`	{ Store,	Heap }
$y = 10$	10 11		$y = 10$	10 11
	1 2			1 7

In addition to the standard logical proposition in the Hoare triple, separation logic defines the following:

THE CONSTANT `emp` This constant states that the heap is empty. In other words, for all s and h, we have $s, h \vdash$ `emp` when h is undefined for all addresses.

THE BINARY OPERATOR $\mapsto$ It defines the heap value at exactly one location. In other words, it maps a given address to a given value.

THE BINARY OPERATOR $*$ Called *star* or "separating conjunction", this operator asserts that the heap can be split into two disjoint parts. In other words, this connective is used to specify non-aliasing segment of memory.

Note that as a consequence, $\{a \mapsto \alpha * a \mapsto \alpha\}$ is not a valid statement in separation logic as this would imply that the address a is mapped to two disjoint segments of the memory.

With those rules at hand, we take a look again at Code 3.1 and we are able to write the following Hoare triple augmented with separation logic:

$$\forall \alpha, \beta, \gamma, \exists \delta, \delta = \alpha + \beta,$$

{ Store,	Heap }	`A(c,a,b)`	{ Store,	Heap }
`a`	$\texttt{a} \mapsto \alpha *$		`a`	$\texttt{a} \mapsto \alpha *$
`b`	$\texttt{b} \mapsto \beta *$		`b`	$\texttt{b} \mapsto \beta *$
`c`	$\texttt{c} \mapsto \gamma$		`c`	$\texttt{c} \mapsto \delta$

By stating the heap as $\{a \mapsto \alpha * b \mapsto \beta * c \mapsto \gamma\}$, we assume and enforce the non-aliasing of the variables `a`, `b`, and `c`. To prove the same statement with aliasing between specific variables we would need to change the triple, a generic solution to this problem is proposed in Section 7.4.1.

As most functions only work on a small part of the heap, we define an additional rule to simplify proofs.

FRAME RULE Given a fragment of code `C` working on a part of memory for which we know P and modifying it into a part of memory for which we know Q, so we have $\{P\}\ \texttt{C}\ \{Q\}$, any other section R of the heap disjoint from P and Q is left untouched, in other words it remains identical. As a result, the frame rule is defined as follows:

$$\frac{\{P\}\ \texttt{C}\ \{Q\}}{\{P * R\}\ \texttt{C}\ \{Q * R\}}\ \text{Frame}$$

3.4 COMPCERT AND THE VERIFIABLE SOFTWARE TOOLCHAIN

COMPCERT is a formally verified compiler [Ler09a; Ler09b] —developed mainly by INRIA— that implements almost all of the C language (ISO C99). It uses a compilation chain (Figure 3.1) that is proven correct in Coq, therefore guaranteeing that the executable assembly code produced behaves as the semantics of the original C source code.

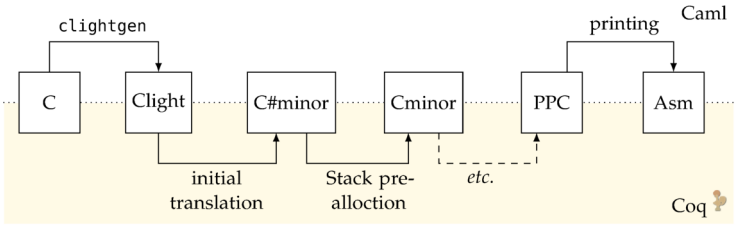


Figure 3.1: The CompCert compilation chain.

As highlighted in Figure 3.1, while the compilation chain is certified, it is worth noting that `clightgen` —which is responsible for the translation from C into the domain-specific language (DSL) Clight— is written in Caml, and thus its correctness is not proven.

VERIFIABLE SOFTWARE TOOLCHAIN. The VST framework [App11; App+14; Cao+18], developed by Appel *et al.* at Princeton, uses the output of `clightgen` to prove the correctness of a program with respect to the semantics of CompCert.

The user must first write a formal specification —a Hoare triple— of a function to be verified in Coq. At this stage, it is important to make the difference between a high-level specification and a low-level one. The first will describe the expected behavior in its simplest form, while the later will aim to replicate the behavior of the implementation. This is done by defining a Coq function as close as possible to the execution steps of the C code, and it results in much simpler verification proof in VST when working with complex pieces of software.

With the range of inputs defined, the VST starts from the precondition of the triple, it mechanically steps through each instruction, and asks the user to verify auxiliary goals such as array bound access, or absence of overflows/underflows. Once done, all that is left to be proven is the link between the low-level and high-level specification.

3.5 A SIMPLE PROOF OF THE CORRECTNESS OF A BIG-NUMBER ADDITION

We prove here the correctness of the 256-bit integer addition function `A` as defined in the TweetNaCl cryptographic library (see Code 3.2). More details of the implementation are given in Section 7.2.3 however we provide a brief summary as follows.

Numbers of 256-bits are split into 16 16-bits limbs, each limb is placed in a 64-bit `long long` signed integer (aliased as `i64`); consequently, 256-bit numbers are represented as an array of 16 `i64`. The result of the 256-bit addition is computed by a simple `for` loop, applying the addition to the respective limbs.

```
1  #define FOR(i,n) for (i = 0;i < n;++i)
2  #define sv static void
3
4  typedef long long i64 __attribute__((aligned(8)));
5  typedef i64 gf[16];
6
7  sv A(gf o,const gf a,const gf b)
8  {
9    int i;
10   FOR(i,16) o[i]=a[i]+b[i];
11 }
```

Code 3.2: `A.c` – Big-number arithmetic addition in TweetNaCl

The high-level view of the proof is illustrated in Figure 3.2. First we compile the source `A.c` with `clightgen`, this produces the Coq file

`A.v`, then using VST we prove the correctness of our Hoare triple in `verif_A.v`.

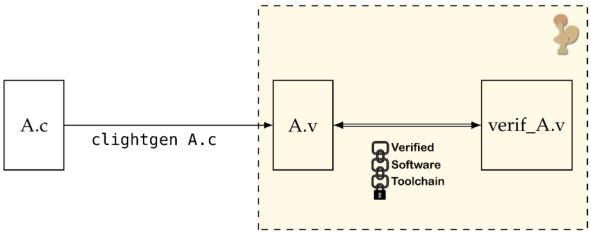


Figure 3.2: High-level overview of a proof of correctness with VST.

CLIGHTGEN. By processing the C source (`A.c`) with `clightgen --normalize`, we produce `A.v`, the Clight Abstract Syntax Tree (AST) of the function and presented in Code 3.3.

```
Definition f_A := {|
  fn_return := tvoid;
  fn_callconv := cc_default;
  fn_params := [(_o, tptr (talignas 3 tlong));
                (_a, tptr (talignas 3 tlong));
                (_b, tptr (talignas 3 tlong))];
  fn_vars := [];
  fn_temps := [(_i, tint);
               (_t'2, talignas 3 tlong);
               (_t'1, talignas 3 tlong)];
  fn_body := (_i = (0);
            for (;
            1;
            _i = (_i + (1));) {
              (if (! _i < (16)) {
                 break;
               }
               _t'1 = _a[_i];
               _t'2 = _b[_i];
               (_o[_i]) = (_t'1 + _t'2);)
            })%C |} : function.
```

Code 3.3: `A.v` – Translation of A into Clight.

When used with the `--normalize` flag, `clightgen` factors out the memory dereference into a top level expression. As a result, Code 3.3 is equivalent to Code 3.4 described below.

```
1  sv A(gf o,const gf a,const gf b)
2  {
3    int i;
4    i64 t1, t2;
5    FOR(i,16) {
6      t1 = a[i];
7      t2 = a[i];
8      o[i]=t1+t2;
9    }
10 }
```

Code 3.4: Big-number arithmetic addition in TweetNaCl.

SPECIFICATIONS. Using the VST, we write the following specifications in Coq for the `A` function.

```
Definition A_spec :=
 DECLARE _A
  WITH v_o: val, v_a: val, v_b: val,
    sh : share,
    o : list val,
    a : list Z,
    b : list Z
  PRE [ _o OF (tptr tlg), _a OF (tptr tlg), _b OF (tptr tlg) ]
        PROP  (writable_share sh;
               Forall (λ x ↦ -2^62 < x < 2^62) a;
               Forall (λ x ↦ -2^62 < x < 2^62) b;
               Zlength a = 16;
               Zlength b = 16;
               Zlength o = 16)
        LOCAL (temp _a v_a; temp _b v_b; temp _o v_o)
        SEP   (sh⟦ v_o ⟧←(lg16)– o;
               sh⟦ v_a ⟧←(lg16)– mVI64 a;
               sh⟦ v_b ⟧←(lg16)– mVI64 b)
  POST [ tvoid ]
        PROP  ()
        LOCAL ()
        SEP   (sh⟦ v_o ⟧←(lg16)– mVI64 (A a b);
               sh⟦ v_a ⟧←(lg16)– mVI64 a;
               sh⟦ v_b ⟧←(lg16)– mVI64 b).
```

Code 3.5: Specification of the A function.

In this specification we state preconditions like:

- PRE: `_o OF (tptr tlg)`
  The function `A` takes as input three pointers to arrays of `i64` (`tptr tlg`) `_o`, `_a` and `_b`.
- LOCAL: `temp _o v_o`
  Each pointer represents an address `v_o`, `v_a` and `v_b`.
- SEP: `sh⟦ v_a ⟧←(lg16)– mVI a`
  In the memory share `sh`, the address `v_a` points to a list of 64-bit integer values `mVI64 a`.
- PROP: `Forall (λ x ↦ -2^62 < x < 2^62) a`
  In order to consider all the possible inputs, we assume each element of the list `a` to be strictly bounded by -2^{62} and 2^{62}.
- PROP: `Zlength a = 16`
  We also assume that the length of the list `a` is 16. This defines the complete representation of `i64[16]` or `gf`.

As postcondition we have conditions like:

- POST: `tvoid`
  The function `A` returns nothing.
- SEP: `sh⟦ v_o ⟧←(lg16)– mVI64 (A a b)`
  In the memory share `sh`, the address `v_o` points to a list of 64-bit integer values `mVI64 (A n p)` where `A` is defined as in Code 3.6 and computes recursively the additions between the input limbs `a` and `b`.

While the PROP section of the post condition is empty, it is also possible to include additional propositional conditions such as "Forall (λ x ↦ -2^63 < x < 2^63) (A a b)".

```
Fixpoint A (a b : list Z) : list Z := match a,b with
| [], q ⇒ q
| q,[] ⇒ q
| h1::q1,h2::q2 ⇒ (Z.add h1 h2) :: A q1 q2
end.
```

Code 3.6: Coq definition of A.

Because the A function in C uses a for loop, we also need to define the following loop invariant (Code 3.7).

```
Definition A_Inv sh v_o v_a v_b o (a:list Z) (b:list Z) :=
  EX i : Z,
   PROP  ()
   LOCAL (temp _a v_a; temp _b v_b; temp _o v_o)
   SEP   (sh⟦ v_o ⟧⟷(lg16)⊢ (tkdp i (mVI64 (A a b)) o);
          sh⟦ v_a ⟧⟷(lg16)⊢ mVI64 a;
          sh⟦ v_b ⟧⟷(lg16)⊢ mVI64 b).
```

Code 3.7: Specification of the loop invariant in the A function.

The structure of the invariant is very similar to the specification of a function except for the EX i : Z, stating that there exists and integer i (our loop index) such that the proposition is true.

In order to represent the intermediate state of sh⟦ v_o ⟧ in the invariant we define the tkdp function as follows. Given an index i and two lists a and b, tkdp returns the concatenation of the first i terms of a and the b list with the first i terms dropped as illustrated in Figure 3.3.

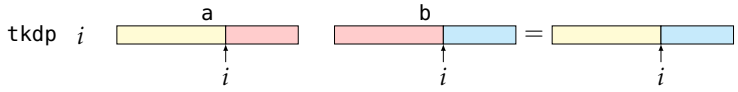


Figure 3.3: Visual representation of the tkdp function.

STEPPING THROUGH THE PROOF. We now turn our focus towards the proof that the C code matches our specifications (Code 3.5), this is formalized in Code 3.8.

```
Lemma body_A:
  (* VST boiler plate. *)
  semax_body
    (* Global variables used in the code. *)
    Vprog
    (* Hoare triples for function calls. *)
    Gprog
    (* Clight AST of the function we verify. *)
    f_A
    (* Our Hoare triple, see below. *)
    A_spec.
```

Code 3.8: Correctness of the 256-bit addition in TweetNaCl.

We start the interactive proof by using the `start_function` tactic from VST. This command initializes the Hoare triple with the pre and postcondition of our specifications, and with the body of the function.

```
Proof.
start_function.
```

As a result we are presented with the VST definition of a Hoare triple:

```
semax Delta PRECONDITION CODE POSTCONDITION.
```

which is equivalent to:

$$\Delta \vdash \{PRECONDITION\}\ \ A\ \ \{POSTCONDITION\}$$

where Δ is the *type-context*, in other words the association between variable names and their data types.

In practice the propositional section of the precondition is considered as a set of hypotheses; as a result the content of `PROP` is automatically introduced in the context, this results the following Coq proof state.

```
1 subgoal
Espec : OracleKind
v_o, v_a, v_b : val
sh : share
o : list val
a, b : list Z
Delta_specs : PTree.t funspec
Delta := abbreviate : tycontext
SH : writable_share sh
H : Forall (λ x : Z ↦ - 2 ^ 62 < x < 2 ^ 62) a
H0 : Forall (λ x : Z ↦ - 2 ^ 62 < x < 2 ^ 62) b
H1 : Zlength a = 16
H2 : Zlength b = 16
H3 : Zlength o = 16
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS := abbreviate : statement
______________________________________(1/1)
semax Delta
  (PROP ( )
   LOCAL (temp _a v_a; temp _b v_b; temp _o v_o)
   SEP (sh⟦ v_o ⟧⟵(lg16)⟝ o;
        sh⟦ v_a ⟧⟵(lg16)⟝ mVI64 a;
        sh⟦ v_b ⟧⟵(lg16)⟝ mVI64 b))
  (Sfor (_i = (0);)
        (_i < (16))%expr
        (_t'1 = _a[_i];
         _t'2 = _b[_i];
         (_o[_i]) = (_t'1 + _t'2);)
        (_i = (_i + (1));)
   MORE_COMMANDS) POSTCONDITION
```

Because the VST approach is similar to a step-by-step symbolic execution, the final postcondition will not be modified through the proof. As a result, it is *hidden* in the `POSTCONDITION` variable. However, this does not prevent us from taking a peek at it.

```
rewrite /POSTCONDITION /abbreviate.
```

This results in the following goal where we recover the POST section of the specifications we defined earlier in Code 3.5.

```
...
_____________________________________(1/1)
semax Delta
  (PROP ( )
   LOCAL (temp _a v_a; temp _b v_b; temp _o v_o)
   SEP (sh[ v_o ]↤(lg16)⊢ o;
        sh[ v_a ]↤(lg16)⊢ mVI64 a;
        sh[ v_b ]↤(lg16)⊢ mVI64 b))
  (Sfor (_i = (0);)
        (_i < (16))%expr
        (_t'1 = _a[_i];
         _t'2 = _b[_i];
         (_o[_i]) = (_t'1 + _t'2);)
        (_i = (_i + (1));)
   MORE_COMMANDS)
  (frame_ret_assert
     (function_body_ret_assert tvoid
        (PROP ( )
         LOCAL ()
         SEP (sh[ v_o ]↤(lg16)⊢ mVI64 (A a b);
              sh[ v_a ]↤(lg16)⊢ mVI64 a;
              sh[ v_b ]↤(lg16)⊢ mVI64 b)))
     (stackframe_of f_A))
```

From this point we will no longer consider the context part of the proof state (above the line) as it remains mostly the same through the steps of the proof.

The first piece of code we are faced with is a `Sfor`, denoting a `for` loop. We step through it by using `forward_for_simple_bound` with the loop invariant we previously defined in Code 3.7.

```
forward_for_simple_bound 16 (A_Inv sh v_o v_a v_b o a b).
```

This creates 3 subgoals.

1. The first subgoal corresponds to the entailment of the initialization of the `for` loop, in other words when $i = 0$.

   ```
   _____________________________________(1/3)
   ENTAIL Delta,
   PROP ( )
   LOCAL (temp _a v_a; temp _b v_b; temp _o v_o)
   SEP (sh[ v_o ]↤(lg16)⊢ o;
        sh[ v_a ]↤(lg16)⊢ mVI64 a;
        sh[ v_b ]↤(lg16)⊢ mVI64 b)
   |-- PROP ( )
       LOCAL (temp _a v_a; temp _b v_b; temp _o v_o)
       SEP (sh[ v_o ]↤(lg16)⊢ tkdp 0 (mVI64 (A a b)) o;
            sh[ v_a ]↤(lg16)⊢ mVI64 a;
            sh[ v_b ]↤(lg16)⊢ mVI64 b)
   ```

   This first subgoal is equivalent to $\Delta, P \vdash Q$ which is roughly translated as "P implies Q in the *type-context* Δ". In practice, it is most often the case of proving that (1) the logical PROP section in Q is correct with respect to the proof context, and (2) the separation logic part of P and Q are equivalent.

To help us in this task, the VST provides us with the tactic `entailer!` which tries to solve this goal. On failure to do so, we are presented with subgoals for each unproven proposition of Q and with the entailment of the separation logic part.

2. The second subgoal corresponds to the Hoare triple of the body of the loop.

```
_____________________________________(2/3)
semax Delta
  (PROP ( )
   LOCAL (temp _i (Vint (Int.repr i));
          temp _a v_a;
          temp _b v_b;
          temp _o v_o)
   SEP (sh⟦v_o⟧↤(lg16)⊢ tkdp i (mVI64 (A a b)) o;
        sh⟦v_a⟧↤(lg16)⊢ mVI64 a;
        sh⟦v_b⟧↤(lg16)⊢ mVI64 b))
  (_t'1 = _a[_i];
   MORE_COMMANDS)
  POSTCONDITION
```

3. And finally the last subgoal is the Hoare triple of the code after the execution of the loop.

```
_____________________________________(3/3)
semax Delta
  (PROP ( )
   LOCAL (temp _i (Vint (Int.repr 16));
          temp _a v_a;
          temp _b v_b;
          temp _o v_o)
   SEP (sh⟦v_o⟧↤(lg16)⊢ tkdp 16 (mVI64 (A a b)) o;
        sh⟦v_a⟧↤(lg16)⊢ mVI64 a;
        sh⟦v_b⟧↤(lg16)⊢ mVI64 b))
  (return;)
  POSTCONDITION
```

Having solved the entailment of the `for`-loop initialization, we now focus on the loop body (see 2. above). For simple code steps such as variable assignment, memory load, memory store, and a few others, the VST provides us with the tactical `forward`. On more complex expressions such as `if` statement, `while` loops, function calls, etc. we have to use dedicated tactics specialized in handling such situations.

By using `forward`, we are applying SEQUENCE RULE and VST identifies the assignment with dereference and selects the associated ASSIGN RULE.

$$\frac{\{P\}\ \texttt{t1 = a[i]}\ \{Q\} \qquad \{Q\}\ \texttt{MORE\_COMMANDS}\ \{R\}}{\{P\}\ \texttt{t1 = a[i]; MORE\_COMMANDS}\ \{R\}}\ \texttt{forward}$$

This results in 2 new subgoals:

1. the entailment of the assignment (type checking, bound checks)
2. the continuation of the code.

```
_____________________________________(1/3)
ENTAIL Delta,
PROP ( )
LOCAL (temp _i (Vint (Int.repr i));
       temp _a v_a;
       temp _b v_b;
       temp _o v_o)
SEP (sh⟦ v_o ⟧⟵(lg16)⟞ tkdp i (mVI64 (A a b)) o;
     sh⟦ v_a ⟧⟵(lg16)⟞ mVI64 a;
     sh⟦ v_b ⟧⟵(lg16)⟞ mVI64 b)
|-- (tc_expr Delta (_a)%expr &&
     local ` (tc_val tlg (Znth i (mVI64 a) Vundef)) &&
     denote_tc_assert
        (typecheck_efield Delta [eArraySubsc (_i)%expr]))%logic
_____________________________________(2/3)
semax Delta
  (PROP ( )
   LOCAL (temp _t'1 (Znth i (mVI64 a) Vundef);
          temp _i (Vint (Int.repr i));
          temp _a v_a;
          temp _b v_b;
          temp _o v_o)
   SEP (sh⟦ v_o ⟧⟵(lg16)⟞ tkdp i (mVI64 (A a b)) o;
        sh⟦ v_a ⟧⟵(lg16)⟞ mVI64 a;
        sh⟦ v_b ⟧⟵(lg16)⟞ mVI64 b))
  (_t'2 = _b[_i];
   MORE_COMMANDS)
  POSTCONDITION
```

We try to solve the first subgoal with the `entailer!` tactic but we are left with a type checking condition:

```
is_long (Znth i (mVI64 a) Vundef)
```

where `Znth` is defined as follows:

$$\texttt{Znth i (mVI64 a) Vundef} = \begin{cases} \texttt{Vlong a[i]}, \text{ if } 0 \leq i < length(a) \\ \texttt{Vundef}, \text{otherwise} \end{cases}$$

This is proven by using the fact that $0 \leq i < 16$, thus that there exists an auxiliary `long` value at index i in the array `a`.

By adding in the proof context additional conditions such as:

```
assert(Haux2: exists aux2, Vlong aux2 = Znth i (mVI64 b) Vundef).
```

we are able to anticipate the need of the VST tactics and let them solve intermediate subgoals automatically. As a result, we solve the subsequent subgoals in similar fashion with a combination of `forward` and `entailer!`.

At this point, Coq asks us to prove that the computation does not present an overflow; this is solved with simple arithmetic.

```
Int64.min_signed ≤ (Znth i a 0) + (Znth i b 0) ≤ Int64.max_signed
```

The next subgoals we are left with is the proof that once the body of the `for` loop is executed, we still preserve the invariant. This is done with the following entailment.

```
ENTAIL Delta,
PROP ( )
LOCAL (temp _t'2 (Vlong (Int64.repr (Znth i b 0)));
       temp _t'1 (Vlong (Int64.repr (Znth i a 0)));
       temp _i (Vint (Int.repr i));
       temp _a v_a;
       temp _b v_b;
       temp _o v_o)
SEP (sh[ v_o ]↤(lg16)⊢ upd_Znth i (tkdp i (mVI64 (A a b)) o)
        (Vlong (Int64.add (Int64.repr (Znth i a 0))
                          (Int64.repr (Znth i b 0))));
     sh[ v_a ]↤(lg16)⊢ mVI64 a;
     sh[ v_b ]↤(lg16)⊢ mVI64 b)
|-- PROP (0 < i + 1 ≤ 16)
    LOCAL (temp _i (Vint (Int.repr i));
           temp _a v_a;
           temp _b v_b;
           temp _o v_o)
    SEP (sh[ v_o ]↤(lg16)⊢ tkdp (i + 1) (mVI64 (A a b)) o;
         sh[ v_a ]↤(lg16)⊢ mVI64 a;
         sh[ v_b ]↤(lg16)⊢ mVI64 b)
```

Once again, by using `entailer!` the size of the goal is significantly reduced, and we are left with a single separation logic expression.

```
_____________________________________(1/2)
sh[ v_o ]↤(lg16)⊢ upd_Znth i
                    (tkdp i (mVI64 (A a b)) o)
                    (Vlong (Int64.repr (Znth i a 0 + Znth i b 0)))
|-- sh[ v_o ]↤(lg16)⊢ tkdp (i + 1) (mVI64 (A a b)) o
```

The fastest approach at this point is to use `replace` tactic in the goal with an appropriate choice of variable. As a result, we are left two subgoals: a trivial expression of type $P \vdash P$ which is solved by the VST tactic `cancel`, and the replace subgoal below.

```
_____________________________________(1/2)
tkdp (i + 1) (mVI64 (A a b)) o =
upd_Znth i (tkdp i (mVI64 (A a b)) o)
           (Vlong (Int64.repr (Znth i a 0 + Znth i b 0)))
```

Such a goal does not depend on any separation logic. From that point it is good practice to extract such a proof into a separate lemma. This ensures that the verification proof stays small and focused on stepping through the function rather exploding in complexity.

Having proven the loop body, we are left with the last Hoare triple which was introduced by the `forward_for_simple_bound` tactic.

```
semax Delta
  (PROP ( )
   LOCAL (temp _i (Vint (Int.repr 16));
          temp _a v_a;
          temp _b v_b;
          temp _o v_o)
   SEP (sh[ v_o ]↤(lg16)⊢ tkdp 16 (mVI64 (A a b)) o;
        sh[ v_a ]↤(lg16)⊢ mVI64 a;
        sh[ v_b ]↤(lg16)⊢ mVI64 b))
  (return;)
  POSTCONDITION
```

As previously, we process the `return` step with `forward` and are left with the final entailment.

```
((sh[ v_o ]<-(lg16)- tkdp 16 (mVI64 (A a b)) o) *
  sh[ v_a ]<-(lg16)- mVI64 a) *
 (sh[ v_b ]<-(lg16)- mVI64 b))%logic
|-- ((sh[ v_o ]<-(lg16)- mVI64 (A a b)) *
      sh[ v_a ]<-(lg16)- mVI64 a) *
     (sh[ v_b ]<-(lg16)- mVI64 b))%logic
```

Once again, by using the same `replace` & `cancel` approach, we solve the last goal and thus prove the correctness of the addition over 256-bit numbers.

The script of the full proof without the ellipsis required for readability purposes is provided in Section 3.A.

TAKEAWAY. At that stage, it is important to notice two things. First that the proof presented above only considers the case where the function arguments are non aliased (we tackle this issue in Section 7.4.1). Secondly that the proof only ensures that the C implementation of the `A` function matches its respective definition in Coq. Proving that the Coq `A` function does implement an addition over 256-bit number does not require the use of the VST, as a result we do not cover it here; however we provide a rough idea of the proof in Chapter 7.

3.6 FROM THEORY TO PRACTICE

In this chapter we briefly reviewed logical notations, intuitionistic logic and the Coq theorem prover. We then introduced Floyd-Hoare logic and separation logic and described how they are used in practice to formally verify pieces of software. We made use of the Verifiable Software Toolchain in Coq to illustrate how such a proof would be conducted on a simple function.

APPENDIX OF CHAPTER 3

3.A VERIFICATION OF THE CORRECTNESS OF A IN TWEETNACL

We provide here the complete VST Coq script of the proof presented in Section 3.5.

```
Set Warnings "-notation-overridden,-parsing".
Require Import Tweetnacl_verif.init_tweetnacl.
Require Import Tweetnacl.Libs.Export.
Require Import Tweetnacl.ListsOp.Export.
Require Import Tweetnacl.Low.A.

Open Scope Z.

Import Low.

(* Simpler proof of an Entailment for later. *)
Lemma end_inv o a b i :
  Zlength a = 16 →
  Zlength b = 16 →
  Zlength o = 16 →
  Zlength (A a b) = 16 →
  0 ≤ i < 16 →
  tkdp (i + 1) (mVI64 (A a b)) o =
  upd_Znth i (tkdp i (mVI64 (A a b)) o)
             (Vlong (Int64.repr (Znth i a 0 + Znth i b 0))).
Proof with try reflexivity ; try omega.
  intros Ha Hb Ho HA Hi.
  rewrite /A in HA.
  rewrite ?Znth_nth ...
  rewrite -ZsubList_nth_Zlength ...
  rewrite /A /tkdp ?simple_S_i ...
  rewrite (upd_Znth_app_step_Zlength _ _ _ Vundef) ?Zlength_map ...
  f_equal ; rewrite map_map (Znth_map 0) ?Znth_nth ...
Qed.

Definition A_spec :=
 DECLARE _A
  WITH v_o: val, v_a: val, v_b: val,
    sh : share,
    o : list val,
    a : list Z,
    b : list Z
  PRE [ _o OF (tptr tlg), _a OF (tptr tlg), _b OF (tptr tlg) ]
        PROP (writable_share sh;
              Forall (fun x ↦ -Z.pow 2 62 < x < Z.pow 2 62) a;
              Forall (fun x ↦ -Z.pow 2 62 < x < Z.pow 2 62) b;
              Zlength a = 16;
              Zlength b = 16;
              Zlength o = 16)
        LOCAL (temp _a v_a; temp _b v_b; temp _o v_o)
        SEP   (sh⟦v_o⟧↤(lg16)⊢ o;
               sh⟦v_a⟧↤(lg16)⊢ mVI64 a;
               sh⟦v_b⟧↤(lg16)⊢ mVI64 b)
  POST [ tvoid ]
        PROP ()
        LOCAL()
```

```
        SEP   (sh[ v_o ]<-(lg16)- mVI64 (A a b);
               sh[ v_a ]<-(lg16)- mVI64 a;
               sh[ v_b ]<-(lg16)- mVI64 b).

Definition A_Inv sh v_o v_a v_b o (a:list Z) (b:list Z) :=
  EX i : Z,
   PROP  ()
   LOCAL (temp _a v_a; temp _b v_b; temp _o v_o)
   SEP   (sh[ v_o ]<-(lg16)- (tkdp i (mVI64 (A a b)) o);
          sh[ v_a ]<-(lg16)- mVI64 a;
          sh[ v_b ]<-(lg16)- mVI64 b).


Definition Gprog : funspecs :=
      ltac:(with_library prog [A_spec]).

Lemma body_A: semax_body Vprog Gprog f_A A_spec.
Proof.
start_function.
assert(HA: Zlength (A a b) = 16).
  rewrite A_Zlength //.

(*
Sfor (_i = (0);)
  (_i < (16))%expr
  (
    _t'1 = _a[_i];
    t'2 = _b[_i];
    (_o[_i]) = (_t'1 + _t'2);
  )
  (_i = (_i + (1));
*)
forward_for_simple_bound 16 (A_Inv sh v_o v_a v_b o a b).
(* Initialization of the For loop *)
entailer!.

(* Proof of the body of the For loop *)
assert(Haux1: exists aux1, Vlong aux1 = Znth i (mVI64 a) Vundef).
  erewrite (Znth_map Int64.zero);
  [eexists ; reflexivity | rewrite Zlength_map; omega].
assert(Haux2: exists aux2, Vlong aux2 = Znth i (mVI64 b) Vundef).
  erewrite (Znth_map Int64.zero);
  [eexists ; reflexivity | rewrite Zlength_map; omega].
destruct Haux1 as [aux1 Haux1].
destruct Haux2 as [aux2 Haux2].

(* _t'1 = _a[_i]; *)
forward; rewrite -Haux1.
entailer!.

(* _t'2 = _b[_i]; *)
forward; rewrite -Haux2.
entailer!.

(* (_o[_i]) = (_t'1 + _t'2); *)
rewrite map_map (Znth_map 0) in Haux1 ; [ | omega ].
rewrite map_map (Znth_map 0) in Haux2 ; [ | omega ].
inversion Haux1 ; clear Haux1; subst aux1.
inversion Haux2 ; clear Haux2; subst aux2.
forward.
entailer!.

(* Proof of correctness of Addition - no overflow *)
clean_context_from_VST.
```

```
assert(-2^62 < (Znth i a 0) < 2 ^ 62)
  by (solve_bounds_by_values_ H).
assert(-2^62 < (Znth i b 0) < 2 ^ 62)
  by (solve_bounds_by_values_ H0).
assert((-2^62) + (-2^62) ≤ Znth i a 0 + Znth i b 0 ≤ 2^62 + 2^62)
  by omega.

rewrite ?Int64.signed_repr ; solve_bounds_by_values.

(* Proof of Entailment of assignment. *)
entailer!.
data_atify; replace_cancel.
apply end_inv ↦ //.

(* Out of the foor loop. Return. *)
(* return; *)
assert(HmA: Zlength (mVI64 (A a b)) = 16)
  by rewrite ?Zlength_map //.
assert(Htkdp: tkdp 16 (mVI64 (A a b)) o = mVI64 (A a b)).
  rewrite -HmA tkdp_all ?Zlength_map //; omega.
forward ; rewrite Htkdp ; cancel.
Qed.

Close Scope Z.
```

Part II

DESIGNING, IMPLEMENTING, BREAKING

4 GIMLI

This chapter introduces Gimli, a 384-bit permutation designed to achieve high security with high performance across a broad range of platforms. After a short introduction (Section 4.1), we present a complete specification of Gimli and its associated schemes (Section 4.2), a detailed design rationale (Section 4.3), an in-depth security analysis (Section 4.4), and performance results for a wide range of platforms (Section 4.5).

4.1 INTRODUCTION

Keccak [Ber+13a], the 1600-bit permutation inside SHA-3, is well known to be extremely energy-efficient: specifically, it achieves very high throughput in moderate-area hardware. Keccak is also well known to be easy to protect against side-channel attacks: each of its 24 rounds has algebraic degree only 2, allowing low-cost masking. The reason that Keccak is well known for these features is that *most symmetric primitives are much worse in these metrics*.

Chaskey [Mou+14], a 128-bit-permutation-based MAC with a 128-bit key, is well known to be very fast on 32-bit embedded microcontrollers: for example, it runs at just 7.0 cycles/byte on an ARM Cortex-M3 microcontroller. The reason that Chaskey is well known for this microcontroller performance is that *most symmetric primitives are much worse in this metric*.

Salsa20 [Ber08c], a 512-bit-permutation-based stream cipher, is well known to be very fast on CPUs with vector units. For example, [BS12] shows that Salsa20 runs at 5.47 cycles/byte using the 128-bit NEON vector unit on a classic ARM Cortex-A8 (iPad 1, iPhone 4) CPU core. The reason that Salsa20 and its variant ChaCha20 [Ber08b] are well known for this performance is again that *most symmetric primitives are much worse in this metric*. This is also why ChaCha20 is now used by smartphones for HTTPS connections to Google [Bur14] and Cloudflare [Sul15] before being standardized in TLS [Lan+].

Cryptography appears in a wide range of application environments, and each new environment seems to provide more reasons to be dissatisfied with most symmetric primitives. For example, Keccak, Salsa20, and ChaCha20 slow down dramatically when messages are short. As another example, Chaskey has a limited security level, and slows down dramatically when the same permutation is used inside a mode aiming for a higher security level.

CONTRIBUTION. We introduce GIMLI, a 384-bit permutation. Like other permutations with sufficiently large state sizes, GIMLI can easily be used to build high-security block ciphers, tweakable block ciphers, stream ciphers, message-authentication codes, authenticated ciphers, hash functions, etc.

What distinguishes GIMLI from other permutations is its *cross-platform* performance. GIMLI is designed for energy-efficient hardware *and* for side-channel-protected hardware *and* for microcontrollers *and* for compactness *and* for vectorization *and* for short messages *and* for a high security level.

Additionally, in light of the NIST lightweight cryptography project (NIST-LWC) [SN15], we propose GIMLI-CIPHER for authenticated-encryption with associated data (AEAD), and GIMLI-HASH for hashing.

AVAILABILITY OF IMPLEMENTATIONS. We place all software and hardware implementations described in this chapter into the public domain to maximize reusability of our results. They are available in the associated materials of this thesis (Section 1.3) and at `https://gimli.cr.yp.to`.

4.2 GIMLI SPECIFICATION

This section defines GIMLI, and subsequently the two constructions GIMLI-HASH and GIMLI-CIPHER. Our motivations are described in Section 4.3.

4.2.1 *Notation*

We denote by $\mathcal{W} = \mathbb{F}_2^{32}$ the set of bit-strings of length 32. We will refer to the elements of this set as "words". We index all vectors and matrices starting at zero. We encode words as bytes in little-endian form.

4.2.2 *The state*

The GIMLI permutation applies a sequence of rounds to a 384-bit state. The state is represented as a parallelepiped with dimensions $3 \times 4 \times 32$ (see Figure 4.1) or, equivalently, as a 3×4 matrix of 32-bit words.

We name the following sets of bits:

- a column j is a sequence of 96 bits such that $\mathcal{S}_j = \{s_{0,j}; s_{1,j}; s_{2,j}\} \in \mathcal{W}^3$
- a row i is a sequence of 128 bits such that $\mathcal{S}_i = \{s_{i,0}; s_{i,1}; s_{i,2}; s_{i,3}\} \in \mathcal{W}^4$

Each round is a sequence of three operations: (1) a non-linear layer, specifically a 96-bit SP-box applied to each column; (2) in every second

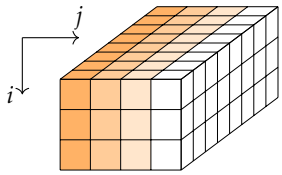


Figure 4.1: State Representation.

round, a linear mixing layer; (3) in every fourth round, a constant addition.

4.2.3 *The non-linear layer*

The SP-box consists 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. See Figure 4.2 for details.

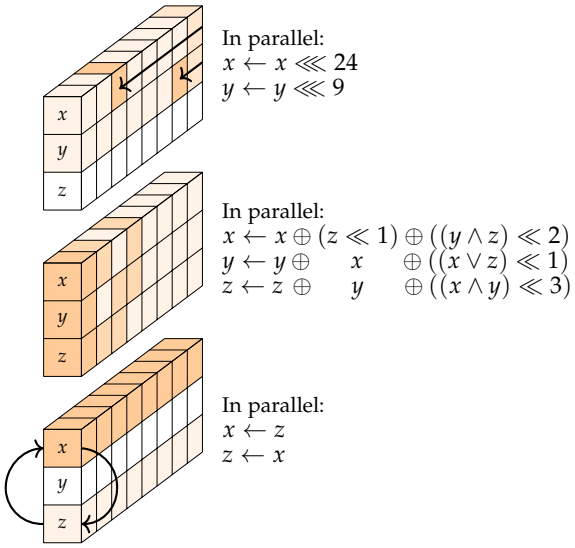


Figure 4.2: The SP-box applied to a column.

4.2.4 *The linear layer*

The linear layer 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. See Figure 4.3 for details of these swaps.

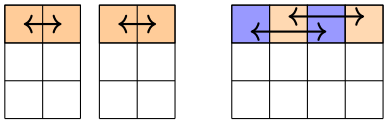


Figure 4.3: The linear layer.

4.2.5 *The round constants*

There are 24 rounds in GIMLI, numbered $24, 23, \ldots, 1$. When the round number r is $24, 20, 16, 12, 8, 4$ we XOR the round constant `0x9e377900` $\oplus$ r to the first state word $s_{0,0}$.

4.2.6 *Putting it together*

Algorithm 1 is pseudocode for the full GIMLI permutation. Code 4.1 in Section 4.A is a C reference implementation.

ALGORITHM 1. The GIMLI permutation.

`Input:` $\mathcal{S} = (s_{i,j}) \in \mathcal{W}^{3\times 4}$
`Output:` $\textsc{Gimli}(\mathcal{S}) = (s_{i,j}) \in \mathcal{W}^{3\times 4}$
  `for` r `from` 24 `down to` 1 *inclusive* `do`
    `for` j `from` 0 `to` 3 *inclusive* `do`
      $x \leftarrow s_{0,j} \lll 24$ ▷*SP-box*
      $y \leftarrow s_{1,j} \lll 9$
      $z \leftarrow s_{2,j}$
      $s_{2,j} \leftarrow x \oplus (z \ll 1) \oplus ((y \wedge z) \ll 2)$
      $s_{1,j} \leftarrow y \oplus x \oplus ((x \vee z) \ll 1)$
      $s_{0,j} \leftarrow z \oplus y \oplus ((x \wedge y) \ll 3)$
    `end for`
    ▷*linear layer*
    `if` $r \mod 4 = 0$ `then`
      $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,1}, s_{0,0}, s_{0,3}, s_{0,2}$ ▷*Small-Swap*
    `else if` $r \mod 4 = 2$ `then`
      $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,2}, s_{0,3}, s_{0,0}, s_{0,1}$ ▷*Big-Swap*
    `end if`

    `if` $r \mod 4 = 0$ `then`
      $s_{0,0} = s_{0,0} \oplus$ `0x9e377900` $\oplus r$ ▷*Add constant*
    `end if`
  `end for`
  `return` $(s_{i,j})$

4.2.7 *Hashing*

GIMLI-HASH initializes a 48-byte GIMLI state to all-zero before reading sequentially through a variable-length input as a series of 16-byte input blocks.

Each full 16-byte input block is handled (*absorbed*) as follows:

- XOR the block into the first 16 bytes of the state (i. e., the top row of 4 words).
- Apply the GIMLI permutation.

The input ends with exactly one final non-full (empty or partial) block, having b bytes where $0 \leq b \leq 15$. This final block is handled as follows:

- XOR the block into the first b bytes of the state.
- XOR `0x01` into the next byte of the state, position b.
- XOR `0x01` into the last byte of the state, position 47.
- Apply the GIMLI permutation.

After the input is fully processed, a 32-byte hash output is obtained as follows:

- Output (*squeeze*) the first 16 bytes of the state.
- Apply the GIMLI permutation.
- Output (*squeeze*) the first 16 bytes of the state.

Figure 4.4 gives an intuition of the sponge construction used in GIMLI-HASH, additionally a complete description is given in Algorithm 4, and a C implementation is provided in Section 4.B. Note that we make use of two functions `touint32()` and `tobytes()`. The former converts 4 bytes to a 32-bit unsigned integer in little-endian, while `tobytes()` converts a 32-bit unsigned integer to 4 bytes. Further, we use `pad`(m) to denote the padding of the message to full blocks.

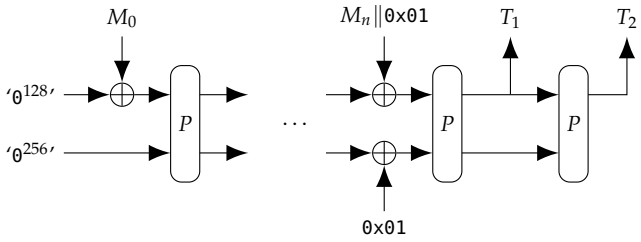


Figure 4.4: GIMLI-HASH sponge construction (simplified) where P are function calls to GIMLI.

Algorithm 2. The ABSORB function.

```
Input:  S = (s_{i,j}) ∈ W^{3×4}, m ∈ F_2^{8×16}
Output: ABSORB(S, m) = (s_{i,j}) ∈ W^{3×4}
  for i from 0 to 3 inclusive do
      s_{0,i} ← s_{0,i} ⊕ touint32(m_{4i}, ..., m_{4i+3})
  end for
  s ← GIMLI(s)
  return s
```

Algorithm 3. The SQUEEZE function.

```
Input:  S = (s_{i,j}) ∈ W^{3×4}
Output: SQUEEZE(S, m) = h ∈ F_2^{8×16}
  h ← tobytes(s_{0,0})||tobytes(s_{0,1})||tobytes(s_{0,2})||tobytes(s_{0,3})
  return h
```

Algorithm 4. The GIMLI-HASH function.

```
Input:  M = F_2^*
Output: GIMLI-HASH (M) = h ∈ F_2^{256}
  S ← '0^{384}'
  m_1, ..., m_t ← pad(M)
  for i from 0 to t inclusive do
      if i = t then
          s_{2,3} ← s_{2,3} ⊕ 0x01000000
      end if
      S ← ABSORB(S, m_i)
  end for
  h ← SQUEEZE(S)
  S ← GIMLI(S)
  h ← h||SQUEEZE(S)
  return h
```

4.2.8 *Authenticated encryption*

After initializing the state to a 16-byte nonce followed by a 32-byte key and applying the GIMLI permutation, GIMLI-CIPHER processes the additional data in the same way as GIMLI-HASH. The message is processed in a similar fashion with the exception that after each absorption of a block, the modified first 16 bytes of the state are produced as cipher text. Once the last non-full block is processed; the 16-byte authentication tag is generated from the first 16 bytes of the state.

See Figure 4.5 for an intuition of the duplex mode used in the encryption process of GIMLI-CIPHER. Algorithm 5 describes an algorithm for authenticated encryption, and Algorithm 6 for an algorithm for verified decryption. We also provide a C reference implementations in Sections 4.C and 4.D

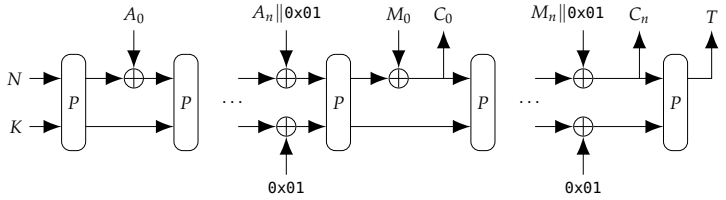


Figure 4.5: GIMLI-CIPHER duplex construction (encryption, simplified) where P are function calls to GIMLI.

ALGORITHM 5. The GIMLI-CIPHER AEAD encryption process.

Input: $M = \mathbb{F}_2^*$, $A = \mathbb{F}_2^*$, $N \in \mathbb{F}_2^{8\times 16}$, $K \in \mathbb{F}_2^{8\times 32}$
Output: GIMLI-CIPHER-ENCRYPT$(M, A, N, K) = C \in \mathbb{F}_2^*, T \in \mathbb{F}_2^{128}$

▷*Initialization*

$\mathcal{S} \leftarrow$ '0^{384}'
for i from 0 to 3 *inclusive* do
  $s_{0,i} \leftarrow$ touint32$(N_{4i}\| \ldots \|N_{4i+3})$
  $s_{1,i} \leftarrow$ touint32$(K_{4i}\| \ldots \|K_{4i+3})$
  $s_{2,i} \leftarrow$ touint32$(K_{16+4i}\| \ldots \|K_{16+4i+3})$
end for
$\mathcal{S} \leftarrow$ GIMLI$(\mathcal{S})$

▷*Processing AD*

$a_1, \ldots, a_q \leftarrow$ pad(A)
for i from 1 to q *inclusive* do
  if $i = q$ then
    $s_{2,3} \leftarrow s_{2,3} \oplus$ 0x01000000
  end if
  $\mathcal{S} \leftarrow$ ABSORB$(\mathcal{S}, a_i)$
end for

▷*Processing Plaintext*

$m_1, \ldots, m_t \leftarrow$ pad(M)
for i from 1 to t *inclusive* do
  $k_i \leftarrow$ SQUEEZE$(\mathcal{S})$
  $c_i \leftarrow k_i \oplus mi$
  if $i = t$ then
    $s_{2,3} \leftarrow s_{2,3} \oplus$ 0x01000000
  end if
  $\mathcal{S} \leftarrow$ ABSORB$(\mathcal{S}, a_i)$
end for
$C \leftarrow c_1\| \ldots \|c_t$

▷*Generating Tag*

$T \leftarrow$ SQUEEZE(s)
return C, T

ALGORITHM 6. The GIMLI-CIPHER AEAD decryption process.

Input: $C = \mathbb{F}_2^*$, $T \in \mathbb{F}_2^{128}$, $A = \mathbb{F}_2^*$, $N \in \mathbb{F}_2^{8\times16}$, $K \in \mathbb{F}_2^{8\times32}$
Output: GIMLI-CIPHER-DECRYPT$(C, T, A, N, K) = M \in \mathbb{F}_2^*$

▷*Initialization*

$\mathcal{S} \leftarrow$ '0^{384}'
for i from 0 to 3 *inclusive* do
  $s_{0,i} \leftarrow$ touint32$(N_{4i} \| \ldots \| N_{4i+3})$
  $s_{1,i} \leftarrow$ touint32$(K_{4i} \| \ldots \| K_{4i+3})$
  $s_{2,i} \leftarrow$ touint32$(K_{16+4i} \| \ldots \| K_{16+4i+3})$
end for
$s \leftarrow$ GIMLI(s)

▷*Processing AD*

$a_1, \ldots, a_q \leftarrow$ pad(A)
for i from 1 to q *inclusive* do
  if $i = q$ then
    $s_{2,3} \leftarrow s_{2,3} \oplus$ 0x01000000
  end if
  $\mathcal{S} \leftarrow$ ABSORB$(\mathcal{S}, a_i)$
end for

▷*Processing Ciphertext*

$c_1, \ldots, c_t \leftarrow$ pad(C)
for i from 1 to t *inclusive* do
  $k_i \leftarrow$ SQUEEZE$(\mathcal{S})$
  $m_i \leftarrow k_i \oplus ci$
  if $i = t$ then
    $s_{2,3} \leftarrow s_{2,3} \oplus$ 0x01000000
  end if
  $\mathcal{S} \leftarrow$ ABSORB$(\mathcal{S}, m_i)$
end for
$M \leftarrow m_1 \| \ldots \| m_t$

▷*Verifying Tag*

$T' \leftarrow$ SQUEEZE(s)
if $T' = T$ then
  return M
else
  return $\perp$
end if

Note that the authenticated cipher and the hash function use the same GIMLI permutation and a very similar approach to process data. As a result, on a device providing authenticated encryption, verified decryption, and hashing, the hardware area for this permutation (including computations and storage) is entirely shared. Similar comment applies to a software implementation.

4.3 UNDERSTANDING THE GIMLI DESIGN

This section explains how we arrived at the GIMLI design presented in Section 4.2.

We started from the well-known goal of designing one unified cryptographic primitive suitable for many different applications: collision-resistant hashing, preimage-resistant hashing, message authentication, message encryption, etc. We found no reason to question the "new conventional wisdom" that a permutation is a better unified primitive than a block cipher. Like KECCAK, ASCON [Dob+16b], etc., we evaluate performance only in the forward direction, and we consider only forward modes; modes that also use the inverse permutation require extra hardware area and do not seem to offer any noticeable advantages.

Where GIMLI departs from previous designs is in its objective of being a single primitive that performs well on every common platform. We do not insist on beating all previous primitives on all platforms simultaneously, but we do insist on coming reasonably close. Each platform has its own hazards that create poor performance for many primitives; what GIMLI shows is that all of these hazards can be avoided simultaneously.

4.3.1 *Vectorization*

On common Intel server CPUs, vector instructions are by far the most efficient arithmetic/logic instructions. As a concrete example, the 12-round CHACHA12 stream cipher has run at practically the same speed as 12-round AES-192 on several generations of Intel CPUs (e. g., 1.7 cycles/byte on Westmere; 1.5 cycles/byte on Ivy Bridge; 0.8 cycles/byte on Skylake), despite AES hardware support, because CHACHA12 takes advantage of the vector hardware on the same CPUs. Vectorization is attractive for CPU designers because the overhead of fetching and decoding an instruction is amortized across several data items.

Any permutation built from (e. g.,) common 32-bit operations can take advantage of a $32b$-bit vector unit if the permutation is applied to b blocks in parallel. Many modes of use of a permutation support this type of vectorization. But this type of vectorization creates two performance problems. First, if b parallel blocks do not fit into vector registers, then there is significant overhead for loads and stores; vectorized KECCAK implementations suffer exactly this problem. Second, a large b is wasted in applications where messages are short.

GIMLI, like SALSA and CHACHA, views its state as consisting of 128-bit rows that naturally fit into 128-bit vector registers. Each row consists of a vector of $128/w$ entries, each entry being a w-bit word, where w is optimized below. Most of the GIMLI operations are applied to every column in parallel, so the operations naturally vectorize. Taking

advantage of 256-bit or 512-bit vector registers requires handling only 2 or 4 blocks in parallel.

4.3.2 *Logic operations and shifts*

GIMLI's design uses only bitwise operations on w-bit words: specifically, AND, OR, XOR, constant-distance left shifts, and constant-distance rotations.

There are tremendous hardware-latency advantages to being able to carry out w bit operations in parallel. Even when latency is not a concern, bitwise operations are much more energy-efficient than integer addition, which (when carried out serially) uses almost $5w$ bit operations for w-bit words. Avoiding additions also allows "interleaved" implementations as in KECCAK, ASCON, etc., saving time on software platforms with word sizes below w.

On platforms with w-bit words there is a software cost in avoiding additions. One way to quantify this cost is as follows. A typical ARX design is roughly balanced between addition, rotation, and XOR. NORX [AJN14b] replaces each addition $a + b$ with a similar bitwise operation $a \oplus b \oplus ((a \wedge b) \ll 1)$, so 3 instructions (add, rotate, XOR) are replaced with 6 instructions; on platforms with free shifts and rotations (such as the ARM Cortex-M4), 2 instructions are replaced with 4 instructions; on platforms where rotations need to be simulated by shifts (as in typical vector units), 5 instructions are replaced with 8 instructions. On top of this near-doubling in cost, the diffusion in the NORX operation is slightly slower than the diffusion in addition, increasing the number of rounds required for security.

The pattern of GIMLI operations improves upon NORX in three ways. First, GIMLI uses a third input c for $a \oplus b \oplus ((c \wedge b) \ll 1)$, removing the need for a separate XOR operation. Second, GIMLI uses only two rotations for three of these operations; overall GIMLI uses 19 instructions on typical vector units, not far behind the 15 instructions used by three ARX operations. Third, GIMLI varies the 1-bit shift distance, improving diffusion compared to NORX and possibly even compared to ARX.

We searched through many combinations of possible shift distances (and rotation distances) in GIMLI, applying a simple security model to each combination. Large shift distances throw away many nonlinear bits and, unsurprisingly, turned out to be suboptimal. The final GIMLI shift distances $(2, 1, 3$ on three 32-bit words) keep 93.75% of the nonlinear bits.

4.3.3 *32-bit words*

Taking $w = 32$ is an obvious choice for 32-bit CPUs. It also works well on common 64-bit CPUs, since those CPUs have fast instructions for,

e. g., vectorized 32-bit shifts. The 32-bit words can also be split into 16-bit words (with top and bottom bits, or more efficiently with odd and even bits as in "interleaved" KECCAK software), and further into 8-bit words.

Taking $w = 16$ or $w = 8$ would lose speed on 32-bit CPUs that do not have vectorized 16-bit or 8-bit shifts. Taking $w = 64$ would interfere with GIMLI's ability to work within a quarter-state for some time (see below), and we do not see a compensating advantage.

4.3.4 *State size*

On common 32-bit ARM microcontrollers, there are 14 easily usable integer registers, for a total of 448 bits. The 512-bit states in SALSA20, CHACHA, NORX, etc. produce significant load-store overhead, which GIMLI avoids by (1) limiting its state to 384 bits (three 128-bit vectors), i. e., 12 registers, and (2) fitting temporary variables into just 2 registers.

Limiting the state to 256 bits would provide some benefit in hardware area, but would produce considerable slowdowns across platforms to maintain an acceptable level of security. For example, 256-bit sponge-based hashing at a 2^{100} security level would be able to absorb only 56 message bits (22% of the state) per permutation call, while 384-bit sponge-based hashing at the same security level is able to absorb 184 message bits (48% of the state) per permutation call, presumably gaining more than a factor of 2 in speed, even without accounting for the diffusion benefits of a larger state. It is also not clear whether a 256-bit state size leaves an adequate long-term security margin against multi-user attacks (see [FJM14]) and quantum attacks; more complicated modes can achieve high security levels using small states, but this damages efficiency.

One of the SHA-3 requirements was 2^{512} preimage security. For sponge-based hashing this requires at least a 1024-bit permutation, or an even larger permutation for efficiency, such as KECCAK's 1600-bit permutation. This requirement was based entirely on matching SHA-512, not on any credible assertion that 2^{512} preimage security will ever have any real-world value. GIMLI is designed for useful security levels, so it is much more comparable to, e. g., 512-bit SALSA20, 400-bit KECCAK-$p[400, n_r]$ (which reduces KECCAK's 64-bit lanes to 16-bit lanes), 384-bit C-QUARK [AKM12], 384-bit SPONGENT-256/256/128 [Bog+11], 320-bit ASCON, and 288-bit PHOTON-256/32/32 [GPP11].

4.3.5 *Working locally*

On the popular low-end ARM Cortex-M0 microcontroller, many instructions can access only 8 of the 14 32-bit registers. Working with more than 256 bits at a time incurs overhead to move data around. Similar comments apply to the 8-bit AVR microcontroller.

GIMLI performs many operations on the left half of its state, and separately performs many operations on the right half of its state. Each half fits into 6 32-bit registers, plus 2 temporary registers.

It is of course necessary for these 192-bit halves to communicate, but this communication does not need to be frequent. The only communication is *Big-Swap*, which happens only once every 4 rounds, so we can work on the same half-state for several rounds.

At a smaller scale, GIMLI performs a considerable number of operations within each column (i. e., each 96-bit quarter-state) before the columns communicate. Communication among columns happens only once every 2 rounds. This locality is intended to reduce wire lengths in unrolled hardware, allowing faster clocks.

4.3.6 *Parallelization*

Like KECCAK and ASCON, GIMLI has degree just 2 in each round. This means that, during an update of the entire state, all nonlinear operations are carried out in parallel: a nonlinear operation never feeds into another nonlinear operation.

This feature is often advertised as simplifying and accelerating masked implementations. The parallelism also has important performance benefits even if side channels are not a concern.

Consider, for example, software using 128-bit vector instructions to apply SALSA20 to a single 512-bit block. SALSA20 chains its 128-bit vector operations: an addition feeds into a rotation, which feeds into an XOR, which feeds into the next addition, etc. The only parallelism possible here is between the two shifts inside a shift-shift-OR implementation of the rotation. A typical vector unit allows more instructions to be carried out in parallel, but SALSA20 is unable to take advantage of this. Similar comments apply to BLAKE [Aum+14] and CHACHA20.

The basic NORX operation $a \oplus b \oplus ((a \wedge b) \ll 1)$ is only slightly better, depth 3 for 4 instructions. GIMLI has much more internal parallelism: on average approximately 4 instructions are ready at each moment.

Parallel operations provide slightly slower forward diffusion than serial operations, but experience shows that this costs only a few rounds. GIMLI has very fast backward diffusion.

4.3.7 *Compactness*

GIMLI is intentionally very simple, repeating a small number of operations again and again. This gives implementors the flexibility to create very small "rolled" designs, using very little area in hardware and very little code in software; or to unroll for higher throughput.

This simplicity creates three directions of symmetries that need to be broken. GIMLI is like Keccak in that it breaks all symmetries within

the permutation, rather than (as in SALSA, CHACHA, etc.) relying on attention from the mode designer to break symmetries. GIMLI puts more effort than KECCAK into reducing the total cost of asymmetric operations.

The first symmetry is that rotating each input word by any constant number of bits produces a near-rotation of each output word by the same number of bits; "near" accounts for a few bits lost from shifts. *Occasionally* (after rounds 24, 20, 16, etc.) GIMLI adds an asymmetric constant to entry 0 of the first row. This constant has many bits set (it is essentially the golden ratio `0x9e3779b9`, as used in TEA), and is not close to any of its nontrivial rotations (never fewer than 12 bits different), so a trail applying this symmetry would have to cancel many bits.

The second symmetry is that each round is identical, potentially allowing slide attacks. This is much more of an issue for small blocks (as in, e. g., 128-bit block ciphers) than for large blocks (such as GIMLI's 384-bit block), but GIMLI nevertheless incorporates the round number r into the constant mentioned above. Specifically, the constant is `0x93e77900` $\oplus\, r$. The implementor can also use `0x93e77900` $+\, r$ since r fits into a byte, or can have r count from `0x93e77918` down to `0x93e77900`.

The third symmetry is that permuting the four input columns means permuting the four output columns; this is a direct effect of vectorization. Occasionally (after rounds 24, 20, 16, etc.) GIMLI swaps entries $0, 1$ in the first row, and swaps entries $2, 3$ in the first row, reducing the symmetry group to 8 permutations (exchanging or preserving $0, 1$, exchanging or preserving $2, 3$, and exchanging or preserving the halves). Occasionally (after rounds 22, 18, 14, etc.) GIMLI swaps the two halves of the first row, reducing the symmetry group to 4 permutations (0123, 1032, 2301, 3210). The same constant distinguishes these 4 permutations.

We also explored linear layers slightly more expensive than these swaps. We carried out fairly detailed security evaluations of GIMLI-MDS (replacing a, b, c, d with $s \oplus a, s \oplus b, s \oplus c, s \oplus d$ where $s = a \oplus b \oplus c \oplus d$), GIMLI-SPARX (as in [Din+16a]), and GIMLI-SHUFFLE (with the swaps as above). We found some advantages in GIMLI-MDS and GIMLI-SPARX in proving security against various types of attacks, but it is not clear that these advantages outweigh the costs, so we opted for GIMLI-SHUFFLE as the final GIMLI.

4.3.8 *Inside the SP-box: choice of words and rotation distances*

The bottom bit of the T-function adds y to z and then adds x to y. We could instead add x to y and then add the new y to z, but this would be contrary to our goal of parallelism; see above.

After the T-function we exchange the roles of x and z, so that the next SP-box provides diffusion in the opposite direction. The shifted

parts of the T-function already provide diffusion in both directions, but this diffusion is not quite as fast, since the shifts throw away some bits.

We originally described rotations as taking place after the T-function, but this is equivalent to rotation taking place before the T-function (except for a rotation of the input and output of the entire permutation). Starting with rotation saves some instructions outside the main loop on platforms with rotated-input instructions; also, some applications reuse portions of inputs across multiple permutation calls, and can cache rotations of those portions. These are minor advantages but there do not seem to be any disadvantages.

Rotating all three of x, y, z adds noticeable software cost and is almost equivalent to rotating only two: it merely affects which bits are discarded by shifts. So, as mentioned above, we rotate only two. In a preliminary GIMLI design we rotated y and z, but we found that rotating x and y improves security by 1 round against our best integral attacks; see below.

This leaves two choices: the rotation distance for x and the rotation distance for y. We found very little security difference between, e. g., $(24, 9)$ and $(26, 9)$, while there is a noticeable speed difference on various software platforms. We decided against "aligned" options such as $(24, 8)$ and $(16, 8)$, although it seems possible that any security difference would be outweighed by further speedups.

4.3.9 *Bijectivity of Gimli*

The bijectivity of the SP-box is not easy to see. If we exclude the swapping and the rotations (which are trivially bijective), we can unroll SP over the first bits:

$$f_0 = \begin{cases} x'_0 \leftarrow x_0 \\ y'_0 \leftarrow y_0 \oplus x_0 \\ z'_0 \leftarrow z_0 \oplus y_0 \end{cases}$$

$$f_1 = \begin{cases} x'_1 \leftarrow x_1 \oplus z_0 \\ y'_1 \leftarrow y_1 \oplus x_1 \oplus (x_0 \vee z_0) \\ z'_1 \leftarrow z_1 \oplus y_1 \end{cases}$$

$$f_2 = \begin{cases} x'_2 \leftarrow x_2 \oplus z_1 \oplus (y_0 \wedge z_0) \\ y'_2 \leftarrow y_2 \oplus x_2 \oplus (x_1 \vee z_1) \\ z'_2 \leftarrow z_2 \oplus y_2 \end{cases}$$

and

$$f_n = \begin{cases} x'_n \leftarrow x_n \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \\ y'_n \leftarrow y_n \oplus x_n \oplus (x_{n-1} \vee z_{n-1}) \\ z'_n \leftarrow z_n \oplus y_n \oplus (x_{n-3} \wedge z_{n-3}) \end{cases}$$

Thus:

$$f_0^{-1} = \begin{cases} x_0 & \leftarrow x'_0 \\ y_0 & \leftarrow y'_0 \oplus x'_0 \\ z_0 & \leftarrow z'_0 \oplus y'_0 \oplus x'_0 \end{cases}$$

$$f_1^{-1} = \begin{cases} x_1 & \leftarrow x'_1 \oplus z_0 \\ y_1 & \leftarrow y'_1 \oplus x'_1 \oplus z_0 \oplus (x_0 \vee z_0) \\ z_1 & \leftarrow z'_1 \oplus y'_1 \oplus x'_1 \oplus z_0 \oplus (x_0 \vee z_0) \end{cases}$$

$$f_2^{-1} = \begin{cases} x_2 & \leftarrow x'_2 \oplus z_1 \oplus (y_0 \wedge z_0) \\ y_2 & \leftarrow y'_2 \oplus x'_2 \oplus z_1 \oplus (y_0 \wedge z_0) \oplus (x_1 \vee z_1) \\ z_2 & \leftarrow z'_2 \oplus y'_2 \oplus x'_2 \oplus z_1 \oplus (y_0 \wedge z_0) \oplus (x_1 \vee z_1) \end{cases}$$

and

$$f_n^{-1} = \begin{cases} x_n \leftarrow x'_n \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \\ y_n \leftarrow y'_n \oplus x_n \oplus (x_{n-1} \vee z_{n-1}) \\ z_n \leftarrow z'_n \oplus y_n \oplus (x_{n-3} \wedge z_{n-3}) \end{cases}$$

SP^{-1} is fully defined by recurrence. SP is therefore bijective.

4.3.10 *Application to hashing*

Gimli-Hash uses the well-known sponge mode [Ber+11a; Ber+08a], the simplest way to hash using a permutation. A generic sponge with a 16-byte rate and a 32-byte capacity has 2^{128} security against a broad class of attacks.

By default, Gimli-Hash provides a fixed-length output of 32 bytes (the concatenation of two 16-byte blocks). However, Gimli-Hash can be used as an eXtendable Output Function (XOF); for example to generate n bytes of output,

- concatenate $\lceil n/16 \rceil$ blocks of 16 bytes, where each block is obtained by extracting the first 16 bytes of the state and then applying the Gimli permutation, and then
- truncate the resulting $16\lceil n/16 \rceil$ bytes to n bytes.

Note that GIMLI-HASH applies the GIMLI permutation with one empty input block, if the input length is a multiple of 16. The seemingly obvious alternative would be to have a 3-way fork after each block, e.g.:

- XOR `0x00` into the capacity part after each non-final block,
- XOR `0x01` into the capacity part after a full final block (i. e., if the size of the final block is exactly 16 bytes), and
- XOR `0x02` into the capacity part after each partial final block (with an extra `0x01` at the end of the block).

This three-way fork saves one call of the permutation if the message length is a multiple of 16. However, the 2-way fork that we use has a performance benefit for lightweight applications.

Imagine a tiny device reading one byte (or even one bit) at a time, and at some point having the read instead say "end of data". With the 2-way fork, the device can handle each byte as follows:

- XOR the byte into the state at the current position,
- increase the current position, and
- if the current position would exceed the end of the block, apply the permutation and set the current position back to the first byte.

Whenever the device receives an "end of data", it can immediately XOR `0x01` into the state at the current position and apply the permutation.

With a 3-way fork, the device instead must delay calling the permutation at the end of each block until it knows whether the data is finished or not. If another byte arrives, the device must buffer that byte, perform the permutation, and then XOR that byte into the block. This complicates the handling of every block.

We conclude that the two-way fork in GIMLI-HASH is better suited for lightweight cryptosystems than the three-way fork, even though the three-way fork does save one application of the GIMLI permutation for 1/16 of all message lengths.

4.3.11 *Application to Authenticated Encryption*

GIMLI-CIPHER uses the well-known duplex mode [Ber+11b], the simplest way to encrypt using a permutation. Duplexing reads an input the same way as sponge hashing: each 16-byte message block m is XORed into the first block x of the state, changing this block of the state to $m \oplus x$. Duplexing also outputs $m \oplus x$ as a block of ciphertext.

We opted for a 256-bit key. This does not mean that we endorse the pursuit of (e. g.,) a 2^{224} security level; it means that we want to reduce concerns about multi-target attacks, quantum attacks, etc.

NIST has recommended that 256-bit keys be accompanied by a 2^{224} single-key pre-quantum security level. We have considered various ways to accommodate this recommendation. For example, one can add 16 of the key bytes (which can be shared with the existing key bytes, as in single-key Even–Mansour) into each 16-byte ciphertext block.

However, this requires the state storing the key to be accessed for each block, rather than just at the beginning of processing a message. In the absence of any explanation of any real-world relevance of security levels above 2^{128}, we have opted to avoid this complication.

We have also considered a mode called "Beetle", which is argued in [Cha+18] to achieve quantitatively higher security than duplexing. Beetle uses the key only at the beginning, and it involves only slightly more computation than duplexing:

- View the plaintext block as two halves: (m_0, m_1).
- View the first state block as (x_0, x_1).
- Output the ciphertext block $(m_0 \oplus x_0, m_1 \oplus x_1)$, as in duplexing.
- Replace the first state block with $(m_0 \oplus x_1, m_1 \oplus x_1 \oplus x_0)$.

However, in environments that communicate (say) 1 bit at a time, it is not clear how to fit the Beetle mode into as little space as the duplex mode. The duplex mode allows each plaintext bit to be read, added into the state, output as a ciphertext bit with no further latency, and then forgotten, with a small (7-bit) counter keeping track of the position inside the block.

4.4 SECURITY ANALYSIS

4.4.1 *Diffusion*

As a first step in understanding the security of reduced-round GIMLI, we consider the following two minimum security requirements:

- the number of rounds required to show the avalanche effect for each bit of the state;
- the number of rounds required to reach a *state full of* '1' starting from a state where only one bit is set. In this experiment we replace every XOR and every AND by OR.

Given the input size of the SP-box, we verify the first criterion with the Monte-Carlo method. We generate random states and flip each bit once. We can then count the number of bits flipped after a defined number of rounds. Experiments show that 10 rounds are required for each bit to change on the average half of the state (see Table 4.12 in Section 4.I).

As for the second criterion, we replace the T-function in the SP-box by the following operations:

$$\begin{aligned} x' &\leftarrow x \vee (z \ll 1) \vee ((y \vee z) \ll 2) \\ y' &\leftarrow y \vee x \vee ((x \vee z) \ll 1) \\ z' &\leftarrow z \vee y \vee ((x \vee y) \ll 3) \end{aligned}$$

By testing the 384 bit positions, we prove that a maximum of 8 rounds are required to fill up the state.

4.4.2 *Differential Cryptanalysis*

To study GIMLI's resistance against differential cryptanalysis we use the same method as has been used for NORX [AJN14a] and SIMON [KLT15] by using a tool-assisted approach to find the optimal differential trails for a reduced number of rounds. In order to enable this approach we first need to define the valid transitions of differences through the GIMLI round function.

The non-linear part of the round function shares similarities with the NORX round function, but we need to take into account the dependencies between the three lanes to get a correct description of the differential behavior of GIMLI. In order to simplify the description we will look at the following function which only covers the non-linear part of GIMLI:

$$f(x,y,z): \quad \begin{aligned} x' &\leftarrow y \wedge z \\ y' &\leftarrow x \vee z \\ z' &\leftarrow x \wedge y \end{aligned} \tag{4.1}$$

where $x, y, z \in \mathcal{W}$. For the GIMLI SP-box we only have to apply some additional linear functions which behave deterministically with respect to the propagation of differences. In the following we denote $(\Delta_x, \Delta_y, \Delta_z)$ as the input difference and $(\Delta_{x'}, \Delta_{y'}, \Delta_{z'})$ as the output difference.

LEMMA 4.4.1 (DIFFERENTIAL PROBABILITY). *For each possible differential through f it holds that*

$$\begin{aligned} \Delta_{x'} \wedge \neg(\Delta_y \vee \Delta_z) &= 0 \\ \Delta_{y'} \wedge \neg(\Delta_x \vee \Delta_z) &= 0 \\ \Delta_{z'} \wedge \neg(\Delta_x \vee \Delta_y) &= 0 \\ (\Delta_x \wedge \Delta_y \wedge \neg\Delta_z) \wedge \neg(\Delta_{x'} \oplus \Delta_{y'}) &= 0 \\ (\Delta_x \wedge \neg\Delta_y \wedge \Delta_z) \wedge (\Delta_{x'} \oplus \Delta_{z'}) &= 0 \\ (\neg\Delta_x \wedge \Delta_y \wedge \Delta_z) \wedge \neg(\Delta_{y'} \oplus \Delta_{z'}) &= 0 \\ (\Delta_x \wedge \Delta_y \wedge \Delta_z) \wedge \neg(\Delta_{x'} \oplus \Delta_{y'} \oplus \Delta_{z'}) &= 0. \end{aligned} \tag{4.2}$$

The differential probability *of* $((\Delta_x, \Delta_y, \Delta_z) \Rightarrow (\Delta_{x'}, \Delta_{y'}, \Delta_{z'}))$ *is given by*

$$\mathrm{DP}((\Delta_x, \Delta_y, \Delta_z) \Rightarrow (\Delta_{x'}, \Delta_{y'}, \Delta_{z'})) = 2^{-2 \cdot \boldsymbol{hw}(\Delta_x \vee \Delta_y \vee \Delta_z)}. \tag{4.3}$$

Proof of Lemma 4.4.1. We want to show how to compute the set of valid differentials for a given input difference

$$\begin{aligned} &\{(\Delta_{x'}, \Delta_{y'}, \Delta_{z'}): \\ &f(x,y,z) \oplus f(x \oplus \Delta_x, y \oplus \Delta_y, z \oplus \Delta_z) = (\Delta_{x'}, \Delta_{y'}, \Delta_{z'})\}. \end{aligned} \tag{4.4}$$

It is sufficient to look at the case where $\mathcal{W}$ is $\mathbb{F}_2$ as there is no interaction between different coordinates in f. The output differences for f are given by

$$\begin{aligned}\Delta_{x'} &= (y \wedge z) \oplus (y \oplus \Delta_y \wedge z \oplus \Delta_z)\\ \Delta_{y'} &= (x \vee z) \oplus (x \oplus \Delta_x \vee z \oplus \Delta_z)\\ \Delta_{z'} &= (x \wedge y) \oplus (x \oplus \Delta_x \wedge y \oplus \Delta_y).\end{aligned} \tag{4.5}$$

If the input difference $(\Delta_x, \Delta_y, \Delta_z) = (0,0,0)$, then the output difference is clearly $(0,0,0)$ as well. We can split the remaining cases in three groups.

CASE $(\Delta_x, \Delta_y, \Delta_z) = (1,0,0)$.
This simplifies Equation 4.5 to

$$\begin{aligned}\Delta_{x'} &= (y \wedge z) \oplus (y \wedge z) = 0\\ \Delta_{y'} &= (x \vee z) \oplus (\neg x \vee z) = -z\\ \Delta_{z'} &= (x \wedge y) \oplus (\neg x \wedge y) = y,\end{aligned} \tag{4.6}$$

and gives us the set of possible output differences

$$(\Delta_{x'}, \Delta_{y'}, \Delta_{z'}) \in \{(0,0,0), (0,0,1), (0,1,0), (0,1,1)\}. \tag{4.7}$$

Similarly, we can find the differentials for the other cases with a single bit difference which gives us the first three conditions in Lemma 4.4.1.

CASE $(\Delta_x, \Delta_y, \Delta_z) = (1,1,0)$.
This simplifies Equation 4.5 to

$$\begin{aligned}\Delta_{x'} &= (y \wedge z) \oplus (\neg y \wedge z) = z\\ \Delta_{y'} &= (x \vee z) \oplus (\neg x \vee z) = -z\\ \Delta_{z'} &= (x \wedge y) \oplus (\neg x \wedge \neg y) = \neg(x \oplus y),\end{aligned} \tag{4.8}$$

giving the set of possible output differences

$$(\Delta_{x'}, \Delta_{y'}, \Delta_{z'}) \in \{(0,1,0), (0,1,1), (1,0,0), (1,0,1)\}. \tag{4.9}$$

Again we can derive the other two cases in a similar way, giving us conditions 4-6 in Lemma 4.4.1.

CASE $(\Delta_x, \Delta_y, \Delta_z) = (1,1,1)$.
This simplifies Equation 4.5 to

$$\begin{aligned}\Delta_{x'} &= (y \wedge z) \oplus (\neg y \wedge \neg z) = \neg(y \oplus z)\\ \Delta_{y'} &= (x \vee z) \oplus (\neg x \vee \neg z) = \neg(x \oplus y)\\ \Delta_{z'} &= (x \wedge y) \oplus (\neg x \wedge \neg y) = \neg(x \oplus y),\end{aligned} \tag{4.10}$$

giving the set of possible output differences

$$(\Delta_{x'}, \Delta_{y'}, \Delta_{z'}) \in \{(0,0,1), (0,1,0), (1,0,0), (1,1,1)\}. \tag{4.11}$$

This corresponds to the last condition in Lemma 4.4.1.

As in all but the $(0,0,0)$ cases, the size of the set of possible output differences is 4 the probability of any differential transition is 2^{-2}. □

We can then use the conditions of Lemma 4.4.1 together with the linear transformations to describe how differences propagate through the GIMLI round functions. For computing the differential probability over multiple rounds we assume that the rounds are independent. Using this model we then search for the optimal differential trails with the SAT/SMT-based approach [AJN14a; KLT15].

We are able to find the optimal differential trails up to 8 rounds of GIMLI (see Table 4.1). After more rounds this approach failed to find any solution in a reasonable amount of time. The 8-round differential trail is given in Table 4.3.

Table 4.1: The optimal differential trails for a reduced number of rounds of GIMLI.

Rounds	1	2	3	4	5	6	7	8
Weight	0	0	2	6	12	22	36	52

In order to cover more rounds of GIMLI we restrict our search to a *good* starting difference and expand it in both directions. As the probability of a differential trail quickly decreases with the Hamming weight of the state, it is likely that any high probability trail will contain some rounds with very low Hamming weight. In Table 4.2, we show the results when starting from a single bit difference in any of the words. Interestingly, the best trails match the optimal differential trails up to 8 rounds given in Table 4.1.

Table 4.2: The optimal differential trails when expanding from a single bit difference in any of the words.

Rounds	1	2	3	4	5	6	7	8	9
$r=0$	0	2	6	14	28	58	102		
$r=1$	0	0	2	6	12	26	48	88	
$r=2$	-	0	2	6	12	22	36	66	110
$r=3$	-	-	8	10	14	32	36	52	74
$r=4$	-	-	-	26	28	32	38	52	74

Using the optimal differential for 7 rounds we can construct a 12-round differential trail with probability 2^{-188} (see Table 4.4 in **??**). If we look at the corresponding differential, this means we do not care about any intermediate differences; many trails might contribute to the probability. In the case of our 12-round trail we find 15800 trails with probability 2^{-188} and 20933 trails with probability 2^{-190}

contributing to the differential. Therefore, we estimate the probability of the differential to be $\approx 2^{-173.63}$.

Table 4.3: Optimal differential trail for 8-round GIMLI.

Round	$s_{*,0}$	$s_{*,1}$	$s_{*,2}$	$s_{*,3}$	Weight
0	0x80404180	0x00020100	-	-	18
	0x80002080	-	-	-	
	0x80002080	0x80010080	-	-	
1	0x80800100	-	-	-	8
	0x80400000	-	-	-	
	0x80400080	-	-	-	
2	0x80000000	-	-	-	0
	0x80000000	-	-	-	
	0x80000000	-	-	-	
3	-	-	-	-	0
	-	-	-	-	
	0x80000000	-	-	-	
4	0x00800000	-	-	-	2
	-	-	-	-	
	-	-	-	-	
5	-	-	-	-	4
	0x00000001	-	-	-	
	0x00800000	-	-	-	
6	0x01008000	-	-	-	6
	0x00000200	-	-	-	
	0x01000000	-	-	-	
7	-	-	-	-	14
	0x01040002	-	-	-	
	0x03008000	-	-	-	
8	0x02020480	-	-	-	-
	0x0a00040e	-	0x06000c00	-	
	0x06010000	-	0x00010002	-	

Table 4.4: A 12-round differential trail for GIMLI with probability 2^{-188} expanding the optimal 7-round differential trail.

Round	$s_{*,0}$	$s_{*,1}$	$s_{*,2}$	$s_{*,3}$	Weight
0	0x04010100	0x80010380	0x06010100	0x80100C00	46
	-	0x40010180	0x02000000	0x40100400	
	0x02008080	0x40010180	0x03018080	0x40104400	
1	-	0x80020080	-	0x80210180	24
	-	0x00060080	-	0x40200080	
	-	0x00070480	-	0x00318400	
2	-	0x00003100	-	0x80401180	20
	-	0x00000100	-	0x80000180	
	-	0x80000980	-	0x80000980	
3	-	-	-	0x80800100	8
	-	-	-	0x80400000	
	-	-	-	0x80400080	
4	-	-	-	0x80000000	0
	-	-	-	0x80000000	
	-	-	-	0x80000000	
5	-	-	-	-	0
	-	-	-	-	
	-	-	-	0x80000000	
6	-	-	-	0x00800000	2
	-	-	-	-	
	-	-	-	-	
7	-	-	-	-	4
	-	-	-	0x00000001	
	-	-	-	0x00800000	
8	-	-	-	0x01008000	6
	-	-	-	0x00000200	
	-	-	-	0x01000000	
9	-	-	0x00010002	-	14
	-	-	-	0x01040002	
	-	-	-	0x03008000	
10	-	-	-	0x020A0480	24
	-	-	0x02000400	0x0A000402	
	-	-	0x00010002	0x0A010000	
11	0x02020104	0x02000100	-	-	40
	-	-	0x00080004	0x14010430	
	-	-	0x00020004	0x1E081480	
12	-	-	0x00000A00	0xB00A0910	-
	0x04020804	0x00020004	0x10001800	0x02186078	
	0x02020104	0x02000100	0x00040008	0x3C102900	

4.4.3 *Algebraic Degree and Integral Attacks*

Since the algebraic degree of the round function of GIMLI is only 2, it is important to check how the degree increases by iterating the round function. We use the (bit-based) division property [Tod15; TM16] to evaluate the algebraic degree, and the propagation search is assisted by mixed integer linear programming (MILP) [Xia+16].

The division property is normally used to search for integral distinguishers. Evaluation of the algebraic degree, which we use in this chapter, is kind of a reverse use of the division property. Assume that the MILP model $\mathcal{M}$ in which the propagation rules of the division property for GIMLI are described, and $\vec{x}$ and $\vec{y}$ denote MILP variables corresponding to input and output of GIMLI, respectively. In the normal use of the division property, $\vec{x}$ has a specific value. To be precise, $x_i = 1$ when the ith bit of the input is active, and $x_i = 0$ otherwise. Then, we check the feasibility that $\vec{y} = \vec{e}_j$, where $\vec{e}_j$ is 384-dimensional unit vector whose jth element is 1. If it is impossible then the jth bit is balanced.

In the reverse use, we constrain $\vec{y}$ and maximize $\sum_{i=1}^{384} x_i$ by MILP. For example, we constrain $\sum_{i=1}^{384} y_i = 1$ and maximize $\sum_{i=1}^{384} x_i$ by using MILP. Suppose the maximized value is d in r-round GIMLI. Then, in other words, if $\sum_{i=1}^{384} x_i = d + 1$, it is impossible that $\sum_{i=1}^{384} y_i = 1$. From this it follows that the algebraic degree of r-round GIMLI is at most d. If we focus on a specific bit in the output, e. g., the jth bit, we constrain $\vec{y} = \vec{e}_j$ and maximize $\sum_{i=1}^{384} x_i$ by using MILP. Moreover, if the algebraic degree involving active bits chosen by attackers is evaluated, we maximize $\sum_{i \in S} x_i$, where S is chosen by attackers. This strategy allows us to efficiently evaluate the algebraic degree in several scenarios.

We first evaluated the upper bound of the algebraic degree on r-round GIMLI, and the result is summarized in Table 4.5.

Table 4.5: Algebraic degree on r-round GIMLI.

Rounds	1	2	3	4	5	6	7	8	9
Algebraic degree	2	4	8	16	29	52	95	163	266

When we focus on only one bit in the output of r-round GIMLI, the increase of the degree is slower than the general case. Especially, the algebraic degree of z_0 in each 96-bit value is lower than other bits because z_0 in rth round is the same as x_6 in $(r-1)$th round. All bits except for z_0 are mixed by at least two bits in $(r-1)$th round. Therefore, we next evaluate the upper bound of the algebraic degree on four z_0 in r-round GIMLI, and the result is summarized as follows.

In integral attacks, a part of the input is chosen as active bits and the other part is chosen as constant bits. Then, we have to evaluate the

Table 4.6: Algebraic degree on r-round GIMLI.

Rounds	1	2	3	4	5	6	7	8	9	10	11
Algebraic degree	1	2	4	8	15	27	48	88	153	254	367

algebraic degree involving active bits. From the structure of the round function of GIMLI, the algebraic degree will be small when 96 entire bits in each column are active. We evaluated two cases: the algebraic degree involving $s_{i,0}$ is evaluated in the first case, and the algebraic degree involving $s_{i,0}$ and $s_{i,1}$ is evaluated in the second case. Moreover, all z_0 in 4 columns are evaluated, and the following table summarizes the upper bound of the algebraic degree in the weakest column in every round.

Table 4.7: Upper bound of the algebraic degree in the weakest column in every round.

Rounds		3	4	5	6	7	8	9	10	11	12	13	14
active	0	0	0	4	8	15	28	58	89	95	96	96	96
columns	0 and 1	0	0	7	15	30	47	97	153	190	191	191	192

The above result implies that GIMLI has an 11-round integral distinguisher when 96 bits in $s_{i,0}$ are active and the others are constant. Moreover, when 192 bits in $s_{i,0}$ and $s_{i,1}$ are active and the others are constant, GIMLI has a 13-round integral distinguisher.

4.5 IMPLEMENTATIONS

This section reports the performance of GIMLI for several target platforms. See Tables 4.8 and 4.9 for cross-platform overviews of software performance.

Table 4.8: AVR performance comparison with various permutations. "Hashing 500 bytes": AVR cycles for comparability with [Bal+12]

Algorithm	Cycles	ROM Bytes	RAM Bytes
SPONGENT [Bal+12]	25 464 000	364	101
KECCAK-f[400] [Bal+12]	1 313 000	608	96
GIMLI-HASH small	805 110	778	44
GIMLI-HASH fast	362 712	19 218	45

[m]: our measurement, [b]: no data

Table 4.9: "Permutation": Cycles/byte for permutation on all platforms. AEAD timings from [BL] are scaled to estimate permutation timings.

Permutation	Cycles/B	ROM Bytes	RAM Bytes
AVR ATmega			
Gimli *small*	413	778	44
ChaCha20 [Wea16]	238	– [b]	132
Salsa20 [HS13]	216	1 750	266
Gimli *fast*	213	19 218	45
AES-128 [Poe03] small	171	1 570	– [b]
AES-128 [Poe03] fast	155	3 098	– [b]
arm Cortex-M0			
Gimli	49	4 730	64
ChaCha20 [SN16]	40	– [b]	– [b]
Chaskey [Mou+14]	17	414	– [b]
arm Cortex-M3/M4			
Spongent [Bog+11; SG15] ([m])	129 486	1 180	– [b]
Ascon [Dob+16b] (opt32 [m])	196	– [b]	– [b]
Keccak-$f[400]$ [AK16]	106	540	– [b]
AES-128 [SS a]	34	3 216	72
Gimli	21	3 972	44
ChaCha20 [HRS16]	13	2 868	8
Chaskey [Mou+14]	7	908	– [b]
arm Cortex-A8			
Keccak-$f[400]$ (KetjeSR) [BL]	37.52	– [b]	– [b]
Ascon [BL]	25.54	– [b]	– [b]
AES-128 [BL] many blocks	19.25	– [b]	– [b]
Gimli single block	8.73	480	– [b]
ChaCha20 [BL] multiple blocks	6.25	– [b]	– [b]
Salsa20 [BL] multiple blocks	5.48	– [b]	– [b]
Intel Haswell			
Gimli single block	4.46	252	– [b]
NORX-32-4-1 [BL] single block	2.84	– [b]	– [b]
Gimli two blocks	2.33	724	– [b]
Gimli four blocks	1.77	1227	– [b]
Salsa20 [BL] eight blocks	1.38	– [b]	– [b]
ChaCha20 [BL] eight blocks	1.20	– [b]	– [b]
AES-128 [BL] many blocks	0.85	– [b]	– [b]

4.5.1 *FPGA & ASIC*

We designed and evaluated three main architectures to address different hardware applications. These different architectures are a trade-off between resources, maximum operational frequency and number of cycles necessary to perform the full permutation. Even with these differences, all three architectures share a common simple communication interface which can be expanded to offer different operation modes. All this was done in VHDL and tested in ModelSim for behavioral results, synthesized, and tested for FPGAs with Xilinx ISE 14.7.

In 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 4.6, 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 case of 12 or 24 combinational rounds, optimized architectures for these cases were done, in order to have better results. The rounds themselves are computed as shown in Figure 4.7. 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.

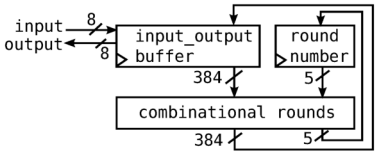


Figure 4.6: Round-based architecture.

Besides the round and the optimized half and full combinational architectures, the other one is a serial-based architecture illustrated in Figure 4.8. The serial-based architecture performs one SP-box application per cycle, through a circular-shift-based architecture, therefore taking in total 4 cycles. In case of the linear layer, it is still executed in one cycle in parallel. The reason of not being done in a serial based manner, is because the cost for the parallel version is very low.

All hardware results are shown in Table 4.10.

In case of FPGAs the lowest latency is the one with 4 combinational rounds in one cycle, and the one with best Resources$\times$Time/State is the one with 2 combinational rounds. For ASICs the results change as the lowest latency is the one with full combinational setting, and the

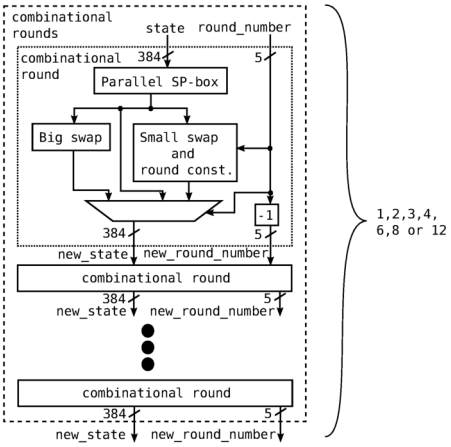


Figure 4.7: Combinational round in round-based architecture.

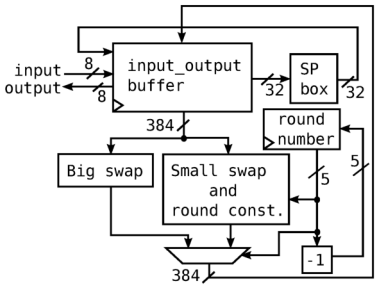


Figure 4.8: Serial-based architecture.

one with best Resources×Time/State is the one with 8 combinational rounds for 180nm and 4 combinational rounds for 28nm. This difference illustrates that each technology can give different results, making it difficult to compare results on different technology.

Hardware variants that do 2 or 4 rounds in one cycle appear to be attractive 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, we synthesized all permutations 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.

Table 4.10: Hardware results for GIMLI and competitors.
Gates Equivalent(GE). Slice(S). LUT(L). Flip-Flop(F).
* Could not finish the place and route, [s]: serial

Perm.	State size	Vers.	Cycl.	Resources	Period (ns)	Time (ns)	Res.×T. ÷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	[s]	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	[s]	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	[s]	108	3846	11.2300	1213	12146.0

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-$p[400, n_r]$ 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 much lower product of time and logic resources.

4.5.2 *SP-box in assembly*

We now turn our attention to software. Subsequent subsections explain how to optimize GIMLI for various illustrative examples of CPUs. As a starting point, we show in Code 4.5.2 how to apply the GIMLI SP-box to three 32-bit registers x, y, z using just two temporary registers u, v.

```
# Rotate          # Compute x      # Compute y      # Compute z
x ← x ⋘ 24        v ← z ≪ 1        v ← y            u ← u ∧ v
y ← y ⋘ 9         x ← y ∧ z        y ← u ∨ z        u ← u ≪ 3
u ← x             x ← x ≪ 2        y ← y ≪ 1        v ← v ⊕ u
                  x ← x ⊕ v        y ← y ⊕ u        z ← v ⊕ z
                  x ← x ⊕ u        y ← y ⊕ v
```

Code 4.5.2: SP-box assembly instructions

4.5.3 *8-bit microcontroller: AVR ATmega*

The AVR architecture provides 32 8-bit registers (256 bits). This does not allow the full 384-bit GIMLI state to stay in the registers: we are forced to use loads and stores in the main loop.

To minimize the overhead for loads and stores, we work on a half-state (two columns) for as long as possible. For example, we focus on the left half-state for rounds 21, 20, 19, 18, 17, 16, 15, 14. Before doing this, we focus on the right half-state through the end of round 18, so that the *Big-Swap* at the end of round 18 can feed 2 words (64 bits) from the right half-state into the left half-state. See Figure 4.9 for the exact order of computation.

A half-state requires a total of 24 registers (6 words), leaving us with 8 registers (2 words) to use as temporaries. We can therefore use the same order of operations as defined in Code 4.5.2 for each SP-box. In a stretch of 8 rounds on a half-state (16 SP-boxes) there are just a few loads and stores.

We provide two implementations of this construction. One is fully unrolled and optimized for speed: it runs in just 10 264 cycles, using 19 218 bytes of ROM. The other is optimized for size: it uses just 778 bytes of ROM and runs in 23 670 cycles. Each implementation requires about the same amount of stack, namely 45 bytes.

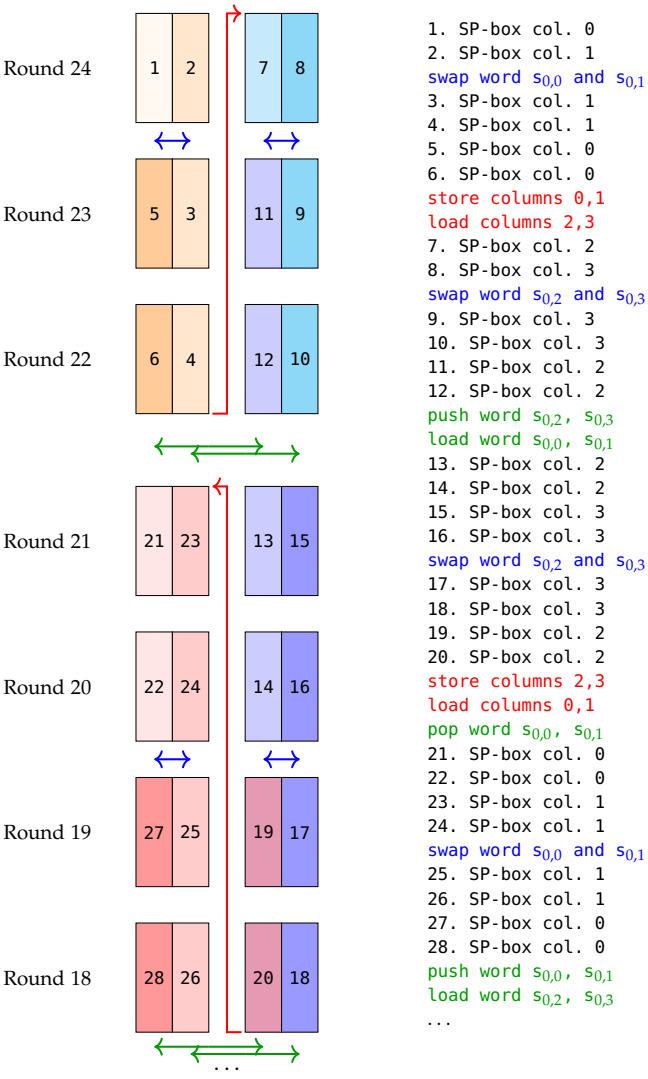


Figure 4.9: Computation order on AVR.

4.5.4 *32-bit low-end embedded microcontroller:* ARM *Cortex-M0*

ARM Cortex-M0 comes with 16 32-bit registers, of which we can use 14 for our computations. However, `orr`, `eor`, `and`-like instructions can only be used on the lower registers (`r0` to `r7`). This forces us to use the same computation layout as in the AVR implementation. We split the state into two halves: one in the lower registers, one in the higher ones. Then we can operate on each during multiple rounds before exchanging them.

4.5.5 *32-bit high-end embedded microcontroller:* ARM *Cortex-M3*

We focus here on the ARM Cortex-M3 microprocessor, which implements the ARMv7-M architecture. There is a higher-end microcontroller, the Cortex-M4, implementing the ARMv7E-M architecture; but our GIMLI software does not make use of any of the DSP, (optional) floating-point, or additional saturated instructions added in this architecture.

The Cortex-M3 features 16 32-bit registers `r0` to `r15`, with one register used as program counter and one as stack pointer, leaving 14 registers for free use. As the GIMLI state fits into 12 registers, and we need only 2 registers for temporary values, we compute the GIMLI permutation without requiring any load or store instructions beyond the initial loads of the input and the final stores of the output.

One particularly interesting feature of various ARM instruction sets including the ARMv7-M instruction set are free shifts and rotates as part of arithmetic instructions. More specifically, all bit-logical operations allow one of the inputs to be shifted or rotated by an arbitrary fixed distance for free. This was used, e.g., in [SYY12, Sec. 3.1] to eliminate all rotation instructions in an unrolled implementation of BLAKE. For GIMLI this feature gives us the non-cyclic shifts by 1, 2, 3 and the rotation by 9 for free. We have not found a way to eliminate the rotation by 24. Each SP-box evaluation thus uses 10 instructions: namely, 9 bit-logical operations (6 XORs, 2 ANDs, and 1 OR) and one rotation.

From these considerations we can derive a lower bound on the amount of cycles required for the GIMLI permutation: Each round performs 4 SP-box evaluations (one on each of the columns of the state), each using 10 instructions, for a total of 40 instructions. In 24 rounds we thus end up with $24 \cdot 40 = 960$ instructions from the SP-boxes, plus 6 XORs for the addition of round constants. This gives us a lower bound of 966 cycles for the GIMLI permutation, assuming an unrolled implementation in which all *Big-Swap* and *Small-Swap* operations are handled through (free) renaming of registers. Our implementation for the M3 uses such a fully unrolled approach and takes 1 047 cycles. This discrepancy with the theoretical bound is the result of the loads and stores of the state.

4.5.6 *32-bit smartphone CPU: ARM Cortex-A8 with NEON*

We focus on a Cortex-A8 for comparability with the highly optimized Salsa20 results of [BS12]. As a future optimization target we suggest a newer Cortex-A7 CPU core, which according to ARM has appeared in more than a billion chips. Since our Gimli software uses almost purely vector instructions (unlike [BS12], which mixes integer instructions with vector instructions), we expect it to perform similarly on the Cortex-A7 and the Cortex-A8.

The Gimli state fits naturally into three 128-bit NEON vector registers, one row per vector. The T-function inside the Gimli SP-box is an obvious match for the NEON vector instructions: two ANDs, one OR, four shifts, and six XORs. The rotation by 9 uses three vector instructions. The rotation by 24 uses two 64-bit vector instructions, namely permutations of byte positions (`vtbl`) using a precomputed 8-byte permutation. The four SP-boxes in a round use 18 vector instructions overall.

A straightforward 4-round-unrolled assembly implementation uses just 77 instructions for the main loop: 72 for the SP-boxes, 1 `vrev64.i32` for *Small-Swap*, 1 to load the round constant from a precomputed 96-byte table, 1 to XOR the round constant, and 2 for loop control (which would be reduced by further unrolling). We handle *Big-Swap* implicitly through the choice of registers in two `vtbl` instructions, rather than using an extra `vswp` instruction. Outside the main loop we use just 9 instructions, plus 3 instructions to collect timing information and 20 bytes of alignment, for 480 bytes of code overall.

The lower bound for arithmetic is $65 \cdot 6 = 390$ cycles: 16 arithmetic cycles for each of the 24 rounds, and 6 extra for the round constants. The Cortex-A8 can overlap permutations with arithmetic. With moderate instruction-scheduling effort we achieved 419 cycles, just 8.73 cycles/byte. For comparison, [BS12] says that a "straightforward NEON implementation" of the inner loop of Salsa20 "cannot do better than 11.25 cycles/byte" (720 cycles for 64 bytes), plus approximately 1 cycle/byte overhead. [BS12] does better than this only by handling multiple blocks in parallel: 880 cycles for 192 bytes, plus the same overhead.

4.5.7 *64-bit server CPU: Intel Haswell*

Intel's server/desktop/laptop CPUs have had 128-bit vectorized integer instructions ("SSE2") starting with the Pentium 4 in 2001, and 256-bit vectorized integer instructions ("AVX2") starting with the Haswell in 2013. In each case the vector registers appeared in CPUs a few years earlier supporting vectorized floating-point instructions ("Streaming SIMD Extensions (SSE)" and "Advanced Vector Extensions (AVX)"), including full-width bitwise logic operations, but not including shifts. The vectorized integer instructions include shifts but not rotations.

Intel has experimented with 512-bit vector instructions in coprocessors such as Knights Corner and Knights Landing and later released in September 2019 the Ice Lake architecture, supporting vectorized rotations and three-input logical operations. However, we focus here on CPUs that were commonly available from Intel and AMD in 2017.

Our implementation strategy for these CPUs is similar to our implementation strategy for NEON: again the state fits naturally into three 128-bit vector registers, with GIMLI instructions easily translating into the CPU's vector instructions. The cycle counts on Haswell are better than the cycle counts for the Cortex-A8 since each Haswell core has multiple vector units. We save another factor of almost 2 for 2-way-parallel modes, since 2 parallel copies of the state fit naturally into three 256-bit vector registers. As with the Cortex-A8, we outperform SALSA20 and CHACHA20 for short messages.

4.6 CONCLUSION: NIST-LWC AND THIRD PARTY CRYPTANALYSIS.

In July 2021, the NIST officially published their report of on the second round candidates for LWC [Tur+21]. It provides a selection of 10 finalists which GIMLI is not part of. This absence is easily explained by several weaknesses discovered by multiple in-depth analyses of the design. While none of these results invalidate any security claim of GIMLI-HASH or GIMLI-CIPHER, they raised sufficient concerns about the security of the GIMLI construction.

We now briefly summarize results published on the security of the GIMLI permutation, GIMLI-HASH, and GIMLI-CIPHER since the submission to NIST LWC. The state-of-the-art attacks on round-reduced GIMLI-CIPHER and GIMLI-HASH are summarized in Table 4.11.

Table 4.11: Summary of attacks on GIMLI-CIPHER and GIMLI-HASH.

Scheme	Attack	Rounds	Time	Memory	Data	reference
GIMLI-HASH	Collision	12	2^{96}	negl.		[Fló+20]
	Collision	6	$2^{91.4}$	negl.		[Fló+20]
	Collision	6	2^{113}	negl.		[ZDW19]
	Collision	6	2^{64}	2^{64}		[LIM20a]
	Preimages	5	2^{96}	$2^{65.6}$		[LIM20a]
GIMLI-CIPHER	State recovery	9	2^{190}	2^{129}	4	[LIM20a]
	State recovery	5	2^{128}	2^{126}	4	[LIM20a]

More specific results are as follows:

- In [LIM20a; LIM20b; LIM19] the authors present various new results on GIMLI. This includes distinguishers, collision and preimage attacks on round-reduced variants, and a new technique to find and verify differential characteristics for GIMLI. It

is shown that several previous differential attacks used invalid characteristics. Based on these results the authors propose several round-reduced attacks on GIMLI-HASH and GIMLI-CIPHER (see Table 4.11).

- In [Fló+20] the authors provide several new distinguishers on the GIMLI permutation and round-reduced attacks on GIMLI-HASH. The best attacks on GIMLI-HASH are included in Table 4.11.
- In [ZDW19] the authors present a 6-round collision attack on GIMLI based on the differential properties of the GIMLI permutation. See Table 4.11.
- In [Bak+20] the authors present distinguishers based on deep learning for 8 rounds of GIMLI-HASH and GIMLI-CIPHER. These distinguishers are slower than the generic distinguisher described above; additionally, contrary to most classical attacks, it is not possible to extend their model to cover further rounds.

APPENDIX OF CHAPTER 4

In the following, we provide C and hacspec[1] implementations of Gimli, Gimli-Hash and Gimli-Cipher. Additionally, we report on the avalanche criterion analysis, differential trails, and benchmarks.

4.A THE GIMLI PERMUTATION IN C

```c
#include <stdint.h>

uint32_t rotate(uint32_t x, int bits)
{
  if (bits == 0) return x;
  return (x << bits) | (x >> (32 - bits));
}

extern void gimli(uint32_t *state)
{
  int round;
  int column;
  uint32_t x;
  uint32_t y;
  uint32_t z;

  for (round = 24; round > 0; --round)
  {
    for (column = 0; column < 4; ++column)
    {
      x = rotate(state[    column], 24);
      y = rotate(state[4 + column],  9);
      z =        state[8 + column];

      state[8 + column] = x ^ (z << 1) ^ ((y&z) << 2);
      state[4 + column] = y ^ x        ^ ((x|z) << 1);
      state[column]     = z ^ y        ^ ((x&y) << 3);
    }

    if ((round & 3) == 0) { // small swap: pattern s...s...s...
      x = state[0]; state[0] = state[1]; state[1] = x;
      x = state[2]; state[2] = state[3]; state[3] = x;
    }
    if ((round & 3) == 2) { // big swap: pattern ..S...S...S.
      x = state[0]; state[0] = state[2]; state[2] = x;
      x = state[1]; state[1] = state[3]; state[3] = x;
    }

    if ((round & 3) == 0) { // add constant: pattern c...c...c...
      state[0] ^= (0x9e377900 | round);
    }
  }
}
```

Code 4.1: C reference implementation of Gimli

1 https://hacspec.github.io/

4.B GIMLI-HASH IN C

```c
#include <stdint.h>
#include <string.h>

int gimli_hash( unsigned char *out,
                const unsigned char *in,
                unsigned long long inlen)
{
    uint32_t state[12];
    uint8_t* state_8 = (uint8_t*)state;
    uint64_t i;

    // === Initialize the state ===
    memset(state,0,sizeof(state));

    // === Absorb all the input blocks ===
    while (inlen >= 16) {
        for(i=0;i<16;++i) state_8[i] ^= in[i];
        in += 16;
        inlen -= 16;
        gimli(state);
    }
    for (i=0;i<inlen;++i) state_8[i] ^= in[i];
    // === Do the padding and switch to the squeezing phase ===
    state_8[i] ^= 1;
    state_8[47] ^= 1;
    gimli(state);

    // === Squeeze out all the output blocks ===
    for (i=0;i<16;++i) out[i] = state_8[i];
    out += 16;
    gimli(state);
    for (i=0;i<16;++i) out[i] = state_8[i];

    return 0;
}
```

4.C ENCRYPTION FUNCTION OF GIMLI-CIPHER IN C

```c
#include <stdint.h>
#include <string.h>

int gimli_aead_encrypt( unsigned char *c,
                        unsigned long long *clen,
                        const unsigned char *m,
                        unsigned long long mlen,
                        const unsigned char *ad,
                        unsigned long long adlen,
                        const unsigned char *nsec,
                        const unsigned char *npub,
                        const unsigned char *k)
{
    (void)nsec;
    uint32_t state[12];
    uint8_t *const state_8 = (uint8_t *)state;
    unsigned long long i;

    // === Initialize the state ===
    memcpy(state_8, npub, 16);
    memcpy(state_8 + 16, k, 32);
    gimli(state);

    *clen = mlen + 16;

    // === Processing AD ===
    while (adlen >= 16) {
        for (i = 0; i < 16; ++i) state_8[i] ^= ad[i];
        ad += 16;
        adlen -= 16;
        gimli(state);
    }
    for (i = 0; i < adlen; ++i) state_8[i] ^= ad[i];
    state_8[i] ^= 1;
    state_8[47] ^= 1;
    gimli(state);

    // === Processing Plaintext ===
    while (mlen >= 16) {
        for (i = 0; i < 16; ++i) state_8[i] ^= m[i];
        for (i = 0; i < 16; ++i) c[i] = state_8[i];
        c += 16;
        m += 16;
        mlen -= 16;
        gimli(state);
    }
    for (i = 0; i < mlen; ++i) state_8[i] ^= m[i];
    for (i = 0; i < mlen; ++i) c[i] = state_8[i];
    c += mlen;
    state_8[i] ^= 1;
    state_8[47] ^= 1;
    gimli(state);

    // === Generate Tag ===
    for (i = 0; i < 16; ++i) c[i] = state_8[i];

    return 0;
}
```

4.D DECRYPTION FUNCTION OF GIMLI-CIPHER IN C

```c
#include <stdint.h>
#include <string.h>

int gimli_aead_decrypt( unsigned char *m,
                        unsigned long long *mlen,
                        unsigned char *nsec,
                        const unsigned char *c,
                        unsigned long long clen,
                        const unsigned char *ad,
                        unsigned long long adlen,
                        const unsigned char *npub,
                        const unsigned char *k)
{
    (void)nsec;
    uint32_t state[12];
    uint8_t *const state_8 = (uint8_t *)state;
    uint32_t result;
    unsigned long long i;
    unsigned long long tlen;

    // === Ciphertext length is not smaller than 16 bytes: Tag ===
    if (clen < 16)
        return -1;

    // === Message length ===
    tlen = clen - 16;
    *mlen = tlen;

    // === Initialize the state ===
    memcpy(state_8, npub, 16);
    memcpy(state_8 + 16, k, 32);
    gimli(state);

    // === Processing AD ===
    while (adlen >= 16) {
        for (i = 0; i < 16; ++i) state_8[i] ^= ad[i];
        gimli(state);
        ad += 16;
        adlen -= 16;
    }
    for (i = 0; i < adlen; ++i) state_8[i] ^= ad[i];
    state_8[i] ^= 1;
    state_8[47] ^= 1;
    gimli(state);

    // === Processing Ciphertext ===
    while (tlen >= 16) {
        for (i = 0; i < 16; ++i) m[i] = state_8[i] ^ c[i];
        for (i = 0; i < 16; ++i) state_8[i] = c[i];
        gimli(state);
        c += 16;
        m += 16;
        tlen -= 16;
    }
    for (i = 0; i < tlen; ++i) m[i] = state_8[i] ^ c[i];
    for (i = 0; i < tlen; ++i) state_8[i] = c[i];
    c += tlen;
    m += tlen;
    state_8[tlen] ^= 1;
    state_8[47] ^= 1;
    gimli(state);
```

```
    // === Verify Tag ===
    result = 0;
    for (i = 0; i < 16; ++i) result |= c[i] ^ state_8[i];
    result -= 1;
    result = ((int32_t)result) >> 16;

    // === Mask Plaintext ===
    tlen = *mlen;
    m -= tlen;
    for (i = 0; i < tlen; ++i) m[i] &= result;

    return ~result;
}
```

4.E THE GIMLI PERMUTATION IN HACSPEC

```
array!(State, 12, U32, type_for_indexes: StateIdx);

fn swap(mut s: State, i: StateIdx, j: StateIdx) -> State {
  let tmp = s[i];
  s[i] = s[j];
  s[j] = tmp;
  s
}

fn gimli_round(mut s: State, r: u32) -> State {
  for col in 0usize..4 {
    let x = s[col].rotate_left(24);
    let y = s[col + 4].rotate_left(9);
    let z = s[col + 8];
    s[col + 8] = x ^ (z << 1) ^ ((y & z) << 2);
    s[col + 4] = y ^ x ^ ((x | z) << 1);
    s[col] = z ^ y ^ ((x & y) << 3);
  }

  if (r & 3u32) == 0u32 {
    s = swap(s, 0, 1);
    s = swap(s, 2, 3);
  }

  if (r & 3u32) == 2u32 {
    s = swap(s, 0, 2);
    s = swap(s, 1, 3);
  }

  if (r & 3u32) == 0u32 {
    s[0] = s[0] ^ (U32(0x9e377900u32) | U32(r))
  }
  s
}

pub fn gimli(mut s: State) -> State {
  for rnd in 0..24 {
    let rnd = (24 - rnd) as u32;
    s = gimli_round(s, rnd);
  }
  s
}
```

Code 4.2: Hacspec reference implementation of GIMLI

4.F GIMLI-HASH IN HACSPEC

```
bytes!(Block, 16);
bytes!(Digest, 32);

fn absorb_block(input_block: Block, mut s: State) -> State {
  let input_bytes = input_block.to_le_U32s();
  s[0] = s[0] ^ input_bytes[0];
  s[1] = s[1] ^ input_bytes[1];
  s[2] = s[2] ^ input_bytes[2];
  s[3] = s[3] ^ input_bytes[3];
  gimli(s)
}

fn squeeze_block(s: State) -> Block {
  let mut block = Block::new();
  for i in 0..4 {
    // XXX: Rust can't figure out the type here for some reason.
    let s_i: U32 = s[i];
    let s_i_bytes = s_i.to_le_bytes();
    block[4 * i] = s_i_bytes[0];
    block[4 * i + 1] = s_i_bytes[1];
    block[4 * i + 2] = s_i_bytes[2];
    block[4 * i + 3] = s_i_bytes[3];
  }
  block
}

fn gimli_hash_state(input: &ByteSeq, mut s: State) -> State {
  let rate = Block::length();
  for i in 0..input.num_chunks(rate) {
    let (block_len, input_block) = input.get_chunk(rate, i);
    if block_len == rate {
      // Absorb full blocks
      let full_block = Block::from_seq(&input_block);
      s = absorb_block(full_block, s);
    } else {
      // Absorb last incomplete block
      // Note that this would work in all
      // other cases as well, but the above is safer.
      let input_block_padded = Block::new();
      let mut input_block_padded =
        input_block_padded.update_start(&input_block);
      input_block_padded[block_len] = U8(1u8);

      // XOR in capacity part
      s[11] = s[11] ^ U32(0x01000000u32);
      s = absorb_block(input_block_padded, s);
    }
  }

  s
}

pub fn gimli_hash(input_bytes: &ByteSeq) -> Digest {
  let s = State::new();
  let s = gimli_hash_state(input_bytes, s);
  let output = Digest::new();
  let output = output.update_start(&squeeze_block(s));
  let s = gimli(s);
  output.update(Block::length(), &squeeze_block(s))
}
```

4.G ENCRYPTION FUNCTION OF GIMLI-CIPHER IN HACSPEC

```
bytes!(Nonce, 16);
bytes!(Key, 32);
bytes!(Tag, 16);

fn process_ad(ad: &ByteSeq, s: State) -> State {
  gimli_hash_state(ad, s)
}

fn process_msg(message: &ByteSeq, mut s: State) -> (State, ByteSeq) {
  let mut ciphertext = ByteSeq::new(message.len());

  let rate = Block::length();
  let num_chunks = message.num_chunks(rate);
  for i in 0..num_chunks {
    let key_block = squeeze_block(s);
    let (block_len, msg_block) = message.get_chunk(rate, i);

    // This pads the msg_block if necessary.
    let msg_block_padded = Block::new();
    let mut msg_block_padded = msg_block_padded.update_start(&msg_block);

    ciphertext = ciphertext.set_chunk(rate, i,
      // the slice_range cuts off the last block if it is padded
      &(msg_block_padded ^ key_block).slice_range(0..block_len),
    );
    if i == num_chunks - 1 {
      msg_block_padded[block_len] = msg_block_padded[block_len] ^ U8(1u8);
      s[11] = s[11] ^ U32(0x01000000u32); // s_2,3
    }
    s = absorb_block(msg_block_padded, s);
  }

  (s, ciphertext)
}

fn process_ct(ciphertext: &ByteSeq, mut s: State) -> (State, ByteSeq) {
  let mut message = ByteSeq::new(ciphertext.len());

  let rate = Block::length();
  let num_chunks = ciphertext.num_chunks(rate);
  for i in 0..num_chunks {
    let key_block = squeeze_block(s);
    let (block_len, ct_block) = ciphertext.get_chunk(rate, i);

    // This pads the ct_block if necessary.
    let ct_block_padded = Block::new();
    let ct_block_padded = ct_block_padded.update_start(&ct_block);
    let msg_block = ct_block_padded ^ key_block;
    // Zero pad the block if necessary
    // (replace bytes with zeros if block_len != rate).
    let mut msg_block = Block::from_slice_range(&msg_block, 0..block_len);

    // Slice_range cuts off the msg_block to the actual length.
    message = message.set_chunk(rate, i, &msg_block.slice_range(0..block_len));
    if i == num_chunks - 1 {
      msg_block[block_len] = msg_block[block_len] ^ U8(1u8);
      s[11] = s[11] ^ U32(0x01000000u32); // s_2,3
    }
    s = absorb_block(msg_block, s);
  }
```

```
  (s, message)
}

pub fn nonce_to_u32s(nonce: Nonce) -> Seq<U32> {
  let mut uints = Seq::<U32>::new(4);
  uints[0] = U32_from_le_bytes(
                U32Word::from_slice_range(&nonce, 0..4));
  uints[1] = U32_from_le_bytes(
                U32Word::from_slice_range(&nonce, 4..8));
  uints[2] = U32_from_le_bytes(
                U32Word::from_slice_range(&nonce, 8..12));
  uints[3] = U32_from_le_bytes(
                U32Word::from_slice_range(&nonce, 12..16));
  uints
}

pub fn key_to_u32s(key: Key) -> Seq<U32> {
  let mut uints = Seq::<U32>::new(8);
  uints[0] = U32_from_le_bytes(
                U32Word::from_slice_range(&key, 0..4));
  uints[1] = U32_from_le_bytes(
                U32Word::from_slice_range(&key, 4..8));
  uints[2] = U32_from_le_bytes(
                U32Word::from_slice_range(&key, 8..12));
  uints[3] = U32_from_le_bytes(
                U32Word::from_slice_range(&key, 12..16));
  uints[4] = U32_from_le_bytes(
                U32Word::from_slice_range(&key, 16..20));
  uints[5] = U32_from_le_bytes(
                U32Word::from_slice_range(&key, 20..24));
  uints[6] = U32_from_le_bytes(
                U32Word::from_slice_range(&key, 24..28));
  uints[7] = U32_from_le_bytes(
                U32Word::from_slice_range(&key, 28..32));
  uints
}

pub fn gimli_aead_encrypt(
  message: &ByteSeq,
  ad: &ByteSeq,
  nonce: Nonce,
  key: Key,
) -> (ByteSeq, Tag) {
  // Add nonce and key to state
  let s = State::from_seq(
            &nonce_to_u32s(nonce).concat(&key_to_u32s(key)));
  let s = gimli(s);

  let s = process_ad(ad, s);
  let (s, ciphertext) = process_msg(message, s);

  let tag = squeeze_block(s);
  let tag = Tag::from_seq(&tag);

  (ciphertext, tag)
}
```

4.H DECRYPTION FUNCTION OF GIMLI-CIPHER IN HACSPEC

```
pub fn gimli_aead_decrypt(
  ciphertext: &ByteSeq,
  ad: &ByteSeq,
  tag: Tag,
  nonce: Nonce,
  key: Key,
) -> ByteSeq {
  // Add nonce and key to state
  let s = State::from_seq(
            &nonce_to_u32s(nonce).concat(&key_to_u32s(key)));
  let s = gimli(s);

  let s = process_ad(ad, s);
  let (s, message) = process_ct(ciphertext, s);

  let my_tag = squeeze_block(s);
  let my_tag = Tag::from_seq(&my_tag);

  let mut out = ByteSeq::new(0);
  if my_tag.equal(tag) {
    out = message;
  };

  out
}
```

4.1 AVALANCHE CRITERION

The following table shows the average number of flipped bits after 10 rounds if the bit at the *index* position is flipped. Sampling has been done over 1024 independent random inputs.

Table 4.12: average number bit flipped and standard deviation.
format: *bit index* $(\bar{x}, \sigma)$

	$s_{0,0}$		$s_{1,0}$		$s_{2,0}$
000	(192.3, 9.6)	032	(192.5, 9.5)	064	(191.8, 9.9)
001	(191.8, 9.8)	033	(192.2, 9.8)	065	(192.8, 9.9)
002	(191.8, 9.8)	034	(192.0, 10.2)	066	(191.7, 9.5)
003	(192.3, 9.6)	035	(191.7, 9.7)	067	(191.5, 9.6)
004	(192.1, 9.8)	036	(192.4, 9.6)	068	(192.0, 10.0)
005	(191.7, 9.9)	037	(191.3, 9.7)	069	(192.0, 10.1)
006	(192.1, 9.9)	038	(191.8, 9.9)	070	(192.0, 9.5)
007	(191.9, 9.8)	039	(192.2, 9.8)	071	(191.2, 9.8)
008	(191.7, 9.8)	040	(192.2, 9.9)	072	(192.2, 9.9)
009	(192.1, 9.8)	041	(192.6, 10.0)	073	(191.7, 9.6)
010	(191.8, 10.1)	042	(192.1, 9.9)	074	(192.2, 9.9)
011	(191.7, 9.9)	043	(192.7, 9.9)	075	(191.8, 9.7)
012	(191.9, 9.8)	044	(191.9, 9.8)	076	(191.9, 9.9)
013	(191.7, 9.3)	045	(192.1, 9.4)	077	(192.4, 9.5)
014	(192.2, 9.6)	046	(192.5, 9.7)	078	(191.9, 9.6)
015	(192.4, 9.5)	047	(192.5, 9.7)	079	(192.0, 10.2)
016	(191.9, 9.7)	048	(192.3, 9.9)	080	(191.8, 9.7)
017	(191.9, 9.7)	049	(192.0, 9.6)	081	(192.7, 9.6)
018	(191.5, 9.7)	050	(191.8, 9.9)	082	(192.2, 9.8)
019	(191.8, 9.6)	051	(191.5, 9.7)	083	(191.9, 9.9)
020	(191.9, 9.7)	052	(192.0, 10.1)	084	(192.5, 9.9)
021	(192.0, 9.8)	053	(192.0, 9.8)	085	(192.1, 9.9)
022	(192.1, 9.7)	054	(191.6, 9.8)	086	(192.2, 9.6)
023	(191.8, 10.2)	055	(192.3, 9.9)	087	(191.6, 9.9)
024	(191.9, 10.0)	056	(191.9, 9.6)	088	(191.6, 9.7)
025	(192.1, 9.9)	057	(192.1, 9.5)	089	(192.4, 9.5)
026	(191.8, 9.9)	058	(192.2, 10.3)	090	(192.5, 10.1)
027	(191.9, 10.1)	059	(192.1, 9.8)	091	(191.8, 9.8)
028	(192.0, 10.0)	060	(192.7, 10.1)	092	(192.2, 9.6)
029	(192.4, 9.9)	061	(192.0, 9.5)	093	(191.9, 10.1)
030	(192.0, 10.0)	062	(192.0, 9.8)	094	(192.3, 9.7)
031	(192.1, 10.0)	063	(191.6, 10.2)	095	(191.7, 9.7)

	$s_{0,1}$		$s_{1,1}$		$s_{2,1}$
096	(191.8, 9.7)	128	(192.5, 9.8)	160	(192.0, 9.8)
097	(191.3, 9.9)	129	(192.0, 10.1)	161	(192.0, 9.8)
098	(192.1, 10.1)	130	(191.9, 10.0)	162	(191.7, 9.7)
099	(191.7, 9.9)	131	(191.9, 10.0)	163	(191.6, 9.7)
100	(191.8, 9.8)	132	(192.1, 10.0)	164	(192.2, 9.9)
101	(191.7, 10.0)	133	(192.1, 9.7)	165	(192.1, 10.3)
102	(192.3, 10.0)	134	(192.0, 9.7)	166	(192.3, 10.1)
103	(191.8, 9.6)	135	(192.4, 9.4)	167	(192.0, 9.8)
104	(192.0, 9.4)	136	(192.4, 9.9)	168	(192.2, 9.8)
105	(191.8, 9.8)	137	(191.9, 9.8)	169	(192.1, 9.5)
106	(192.2, 10.0)	138	(191.9, 10.3)	170	(191.6, 9.6)
107	(192.3, 9.6)	139	(191.6, 9.7)	171	(192.2, 10.0)
108	(192.0, 9.7)	140	(191.8, 9.9)	172	(192.5, 10.1)
109	(191.9, 9.5)	141	(192.6, 9.8)	173	(192.2, 9.6)
110	(192.1, 9.6)	142	(191.6, 9.8)	174	(192.6, 9.9)
111	(192.5, 9.6)	143	(191.7, 9.8)	175	(192.3, 9.6)
112	(192.0, 9.6)	144	(192.0, 9.8)	176	(192.0, 9.8)
113	(191.9, 9.6)	145	(191.8, 9.6)	177	(192.4, 9.9)
114	(191.7, 9.5)	146	(191.7, 10.0)	178	(192.5, 9.6)
115	(192.4, 9.8)	147	(191.7, 9.9)	179	(191.5, 9.5)
116	(192.0, 9.7)	148	(191.7, 9.9)	180	(191.9, 9.7)
117	(191.8, 9.8)	149	(192.1, 9.7)	181	(192.4, 9.7)
118	(192.1, 9.6)	150	(191.7, 9.9)	182	(192.0, 9.9)
119	(192.4, 10.0)	151	(191.9, 10.0)	183	(191.5, 9.9)
120	(191.9, 10.0)	152	(191.9, 9.9)	184	(192.1, 9.8)
121	(191.6, 9.6)	153	(192.5, 10.1)	185	(191.8, 9.8)
122	(192.1, 9.6)	154	(192.2, 10.1)	186	(191.9, 9.7)
123	(191.6, 9.6)	155	(191.6, 9.9)	187	(192.1, 9.8)
124	(191.8, 9.6)	156	(191.9, 9.3)	188	(192.2, 9.9)
125	(191.6, 9.7)	157	(192.2, 9.8)	189	(192.2, 9.6)
126	(191.6, 9.8)	158	(192.1, 9.9)	190	(192.4, 9.8)
127	(192.2, 9.8)	159	(191.6, 9.5)	191	(192.8, 10.1)

	$s_{0,2}$		$s_{1,2}$		$s_{2,2}$
192	(192.0, 9.8)	224	(192.5, 9.8)	256	(192.2, 10.0)
193	(191.6, 9.9)	225	(191.5, 10.2)	257	(192.4, 9.7)
194	(191.9, 10.0)	226	(192.9, 9.8)	258	(191.9, 9.6)
195	(192.0, 9.6)	227	(191.5, 9.5)	259	(192.5, 9.7)
196	(191.5, 10.0)	228	(192.3, 9.8)	260	(191.9, 9.9)
197	(192.1, 9.9)	229	(192.2, 9.8)	261	(192.9, 9.5)
198	(191.9, 9.8)	230	(191.9, 9.7)	262	(192.4, 9.8)
199	(191.7, 9.4)	231	(191.9, 9.8)	263	(191.9, 10.0)
200	(192.0, 9.6)	232	(192.5, 10.2)	264	(191.9, 10.0)
201	(191.3, 9.8)	233	(192.0, 9.9)	265	(192.2, 9.6)
202	(191.5, 9.9)	234	(191.6, 10.0)	266	(191.9, 10.0)
203	(192.0, 9.9)	235	(192.1, 9.7)	267	(191.9, 10.0)
204	(191.8, 9.8)	236	(191.9, 9.4)	268	(191.9, 9.7)
205	(191.9, 9.9)	237	(192.1, 9.3)	269	(191.9, 9.6)
206	(192.2, 9.9)	238	(191.9, 9.8)	270	(192.2, 9.6)
207	(192.4, 9.8)	239	(192.2, 10.0)	271	(192.1, 9.7)
208	(191.7, 10.2)	240	(191.8, 9.7)	272	(191.7, 9.9)
209	(191.9, 9.7)	241	(191.6, 10.4)	273	(191.9, 9.8)
210	(192.0, 9.5)	242	(192.0, 10.0)	274	(192.4, 10.1)
211	(192.3, 10.0)	243	(192.0, 9.6)	275	(192.0, 9.7)
212	(192.3, 9.9)	244	(192.5, 9.5)	276	(192.3, 10.0)
213	(191.8, 9.4)	245	(192.3, 9.8)	277	(192.1, 9.9)
214	(192.3, 9.8)	246	(192.0, 9.7)	278	(192.3, 9.8)
215	(192.0, 10.2)	247	(192.3, 9.6)	279	(191.5, 10.0)
216	(191.8, 10.2)	248	(192.1, 10.2)	280	(192.0, 9.6)
217	(192.4, 9.8)	249	(192.0, 9.6)	281	(191.6, 9.8)
218	(192.3, 10.0)	250	(191.7, 9.7)	282	(192.2, 9.8)
219	(192.1, 9.7)	251	(192.3, 9.5)	283	(192.1, 9.9)
220	(192.1, 9.9)	252	(192.0, 9.7)	284	(191.5, 9.9)
221	(191.8, 10.0)	253	(192.4, 10.4)	285	(192.1, 9.7)
222	(192.6, 9.8)	254	(192.3, 9.6)	286	(191.9, 9.7)
223	(191.8, 10.0)	255	(192.3, 9.9)	287	(192.1, 9.9)

	$s_{0,3}$		$s_{1,3}$		$s_{2,3}$
288	(191.7, 9.6)	320	(192.2, 9.6)	352	(191.6, 9.7)
289	(192.3, 10.0)	321	(192.1, 9.8)	353	(192.3, 9.9)
290	(192.0, 9.8)	322	(191.6, 9.7)	354	(192.2, 9.7)
291	(192.2, 10.2)	323	(192.2, 9.4)	355	(191.7, 9.9)
292	(192.3, 9.5)	324	(192.0, 9.6)	356	(191.5, 9.8)
293	(191.8, 10.0)	325	(191.5, 9.7)	357	(192.3, 9.7)
294	(192.0, 9.7)	326	(192.5, 10.2)	358	(192.2, 9.8)
295	(192.5, 9.7)	327	(192.6, 10.0)	359	(191.7, 9.9)
296	(192.1, 9.7)	328	(192.0, 9.6)	360	(192.0, 10.0)
297	(192.1, 9.4)	329	(192.2, 9.9)	361	(192.2, 9.7)
298	(192.1, 9.8)	330	(192.0, 9.8)	362	(191.9, 9.5)
299	(191.8, 9.7)	331	(191.9, 9.9)	363	(191.9, 9.7)
300	(192.2, 9.5)	332	(192.1, 9.7)	364	(191.9, 10.1)
301	(192.3, 10.2)	333	(192.5, 9.9)	365	(191.9, 9.9)
302	(192.1, 9.7)	334	(191.9, 9.8)	366	(192.0, 9.9)
303	(191.9, 10.0)	335	(191.9, 9.6)	367	(192.0, 9.8)
304	(192.0, 10.2)	336	(192.3, 9.7)	368	(191.9, 9.5)
305	(191.9, 9.8)	337	(191.7, 9.6)	369	(191.9, 9.9)
306	(192.5, 9.5)	338	(192.0, 9.7)	370	(192.1, 10.0)
307	(191.9, 9.5)	339	(192.1, 10.2)	371	(191.9, 10.2)
308	(191.8, 9.8)	340	(192.0, 9.8)	372	(191.8, 9.8)
309	(192.4, 9.6)	341	(192.3, 9.6)	373	(191.9, 9.8)
310	(192.0, 9.8)	342	(192.3, 9.8)	374	(192.1, 10.1)
311	(191.5, 9.7)	343	(191.7, 9.6)	375	(192.2, 9.7)
312	(192.3, 10.0)	344	(192.4, 10.3)	376	(192.3, 9.9)
313	(191.8, 9.7)	345	(192.2, 9.9)	377	(192.3, 9.7)
314	(192.2, 10.2)	346	(192.2, 10.0)	378	(192.0, 9.8)
315	(192.4, 9.8)	347	(192.3, 9.9)	379	(191.4, 10.0)
316	(192.2, 9.9)	348	(191.8, 9.9)	380	(191.9, 9.9)
317	(192.3, 9.7)	349	(192.3, 9.3)	381	(191.8, 9.8)
318	(191.8, 9.5)	350	(192.4, 9.6)	382	(191.9, 9.7)
319	(192.2, 9.6)	351	(192.1, 9.8)	383	(191.0, 9.6)

5 ASSEMBLY OR OPTIMIZED C FOR LIGHTWEIGHT CRYPTOGRAPHY ON RISC-V?

After a short introduction, this chapter is structured as follows. Section 5.2 provides background information on the RISC-V 32-bit architecture and instruction set. We also give the necessary details on the platforms used for benchmarking. In Section 5.3, we briefly recall the selected algorithms and present our optimization strategies. Then, we describe the benchmark and discuss the achieved results in Section 5.4. Finally, in Section 5.6, we conclude the chapter.

5.1 INTRODUCTION

The RISC-V project, with roots in academia and research (University of California, Berkeley), has initiated a fundamental shift in the technical and business models for microprocessors. RISC-V [Wat+17], a royalty-free and open-source reduced-instruction-set architecture (ISA), provides a competitive advantage and the required degree of flexibility to develop secure microprocessors with addresses of 32-, 64-, and 128-bits in length.

CONTRIBUTION. We compare optimization at different levels of the round-2 NIST-LWC [SN15] candidates on the RISC-V architecture. To achieve this, we first present optimized RISC-V implementations of several cryptographic algorithms. Further, we study the impact of implementing these primitives on RISC-V in assembly compared to implementations in C. Based on this, general implementation strategies are derived and discussed.

RELATED WORK. Many aspects regarding the optimization of lightweight cryptographic algorithms have been studied in the literature. In [Mou15], generic security, efficiency, and some considerations for cryptographic design of lightweight constructions were explored. Cruz, Reis, Aranha, López, and Patil [Cru+16] discussed techniques for efficient and secure implementations of lightweight encryption on ARM devices. Also, the modular and reusable architecture of RISC-V facilitates a variety of designs for the implementation of accelerators, ranging from loosely [Wan+20] to tightly coupled designs [Alk+20; FSS20]. However, only few works focused on the optimization of cryptographic algorithms on the standard RISC-V instruction set. Stoffelen [Sto19] presented the first optimized assembly implementations of AES, ChaCha, and the Keccak-f[1600] permutation for the RISC-V

instruction set. In [Nis+19] the 32-second round finalists from the NIST-LWC were evaluated on RISC-V without further optimization.

AVAILABILITY OF IMPLEMENTATIONS. We place all software and hardware implementations described in this chapter into the public domain to maximize reusability of our results. They are available in the associated material of this thesis (Section 1.3) and directly at `https://github.com/AsmOptC-RiscV/Assembly-Optimized-C-RiscV`.

5.2 RISC-V

In Section 5.2.1 we describe in more details the RISC-V 32-bit architecture before detailing the associated instruction set (Section 5.2.2). We then discuss different approaches to execute code targeting the RISC-V platform (Section 5.2.3).

5.2.1 *Architecture*

The RISC-V architecture uses 32 32-bit registers numbered from `x0` through `x31`. To ease their use, they also have aliases. `zero` (`x0`) is hard-wired to the value 0; `ra` (`x1`) corresponds to the return address; `sp` (`x2`) to the stack pointer; `gp` (`x3`) to the global pointer; `tp` (`x4`) to the thread pointer. `a0`-`a7` (`x10`-`x17`) are function arguments with `a0` and `a1` also functioning as return values. `s0`-`s11` (`x8`-`x9`, `x18`-`x27`) are saved registers. Finally, `t0`-`t6` (`x5`-`x7`, `x28`-`x31`) are temporary registers.

The caller has the responsibility for the saved registers `s0`-`s11` while the callee is able to freely modify the arguments (`a0`-`a7`) and temporary registers (`t0`-`t6`).

Excluding the `zero`, `ra`, `sp`, `gp`, and `tp` registers, we are left with 27 freely usable 32-bit registers. This is about twice of what is available in the Cortex-M3 and Cortex-M4 architectures; and it enables us to easily take care of register allocation.

5.2.2 *Instruction set*

The RISC-V base instruction set contains a small number of instructions which we briefly describe here.

Bitwise and arithmetic instructions such as `add`, `addi`, `and`, `andi`, `or`, `ori`, `sub`, `xor`, `xori` take three register operands, or if postfixed by `i`, two registers and one 12-bit sign-extended immediate.

We soon notice missing instructions. e. g., `mov rd, rs` is implemented by taking advantage of the zero register as `add rd, zero, rs`. Similarly, the two's complement negation `neg rd, rs` is replaced by `sub rd, zero, rs` and the one's complement negation `not rd, rs` as `xori rd, rs, -1`. Subtract immediate (`subi`) is written as `addi` with a negative immediate.

The base ISA does not provide rotation instructions but only logical and arithmetic shifts: `sll`, `slli`, `srl`, `srli`, `sra`, and `srai`. Those instructions are read as `s`hift [`l`eft | `r`ight] [`l`ogical | `a`rithmetic].

The loading of constants is done with two instructions: `lui` and `addi`. Load upper immediate `lui` takes a 20-bit unsigned immediate and places it in the upper 20 bits of the destination register. The lowest 12 bits are filled with zeros.

```
.equ UART_BASE, 0x40003000

        lui a0,      %hi(UART_BASE)
        addi a0, a0, %lo(UART_BASE)
```

In order to `l`oad `w`ords, `h`alf-words (`u`nsigned), or `b`ytes (`u`nsigned) from memory, the instructions `lw`, `lh`, `lhu`, `lb`, `lbu` are used. Similarly `sw`, `sh`, `shu`, `sb`, `sbu` are available to `s`tore values into the memory. For example `lw a5, 8(a2)` will load into `a5` the word located at address `a2 + 8`. Note that the offset has to be a constant. Additionally, loads and stores of words have to be 32-bit aligned, e. g., `lw a5, 3(a2)` will fail.

Text labels are used as targets for branches, unconditional jumps and symbol offsets. They are added to the symbol table of the compiled module. Numeric labels are used for local references. When used in jumps and similar instructions, they are suffixed with 'f' for a forward reference or 'b' for a backwards reference.

```
loop:                          1:
        ...                            ...
        j loop                         j 1b

        j func                         j 2f
        ...                            ...
fun:                           2:
```

In addition to the `jal` and `jalr` unconditional jump —relative to the program counter or as an absolute address in a register— the instruction `beq`, `blt`, `bltu`, `bge`, `bgeu` are used for conditional jumps. They take three arguments, the first two are used in the comparison while the third one is the destination —label— encoded later as an offset relative to the program counter.

To perform our benchmarks we use the `csrr` instruction (control and status register) to read the 64-bit cycle-counter. On the RV32I architecture, it is split into two 32-bit words (`cycle` and `cycleh`).

5.2.3 *Executing code*

To write optimized code for a specific architecture, we need ways to measure improvements or regressions. Below, we describe 3 test platforms which allowed us to benchmark our code.

SIFIVE E31 CORE. We use a HiFive1[1] development board which contains the FE310-G000 SoC with an E31 core. The CPU implements the RV32IMAC instruction set. This corresponds to the RV32I base ISA with the extensions for multiplications, atomic instructions and compressed instructions.

It has to be noted that RISC-V does not specify how many cycles an instruction may take or the kind of memory used. As a result, benchmarks between different RISC-V cores have to be carefully compared.

The E31 runs at 320+ MHz and uses a 5-stage single-issue in-order pipeline. Additionally, it uses a 16KB, 2-way instruction cache. Fetching an instruction from the cache takes only 1 cycle. The execution of most instructions takes 1 cycle, with a few exceptions. For example, on a cache hit, load word (`lw`) takes 2 cycles, loads of half word (`lh`) and bytes (`lb`) have a 3-cycle latency. In the case of a cache miss, the latency is highly dependent on the flash controller's clock frequency. To prevent such unpredictability, we fill up the cache before any benchmarks.

The E31 comes with a 1-cycle-latency branch predictor. It uses a 28-entry branch target buffer (BTB), a 512-entry branch history table (BHT) for the direction of conditional branches, and a 6-entry return-address stack (RAS). A correctly predicted control-flow instruction results in no penalty while mispredictions incur a 3-cycle penalty.

The RISC-V specification requires a 64-bit cycle counter accessible via two CSR registers which we will use to benchmark code. Occasionally for unknown reasons measurements on the board may end up taking much longer than expected, we ignore these odd values.

VEXRISCV SIMULATOR. The VexRiscv simulator [Spib] is a 32-bit RISC-V CPU implementation written in SpinalHDL [Spia]. Although it is possible to load the core onto an FPGA; we use the Verilator simulator to emulate a core and flash binaries to it. This process allows us to have cycle counts and to evaluate how each algorithm is performing.

The core features the RV32IM instruction set. This corresponds to the base ISA with the extension for multiplications. We initialize the simulator with 256 KB of RAM and 128 KB of sRAM.

Similarly to the E31 core, the VexRiscv makes use of a 5-stage pipeline. The absence of a branch predictor and an instruction cache give a significant advantage to algorithms which have been unrolled either by hand or the compiler. This explains major cycle-counts differences in the execution of different implementations of a same algorithm.

RISCVOVPSIM SIMULATOR. Finally, as opposed to executing code on a board or on a fully simulated core, we use the Open Virtual

1 Discontinued, see sifive.com/boards, last accessed on March 4^{th}, 2021.

Platforms developed by Imperas Software, Ltd. Their RISC-V simulator [IS] uses Just-in-Time Code Morphing and executes RISC-V code on a Linux or Windows host computer.

This simulator implements the full instruction set and permits us to enable or disable specific extensions such as vector instructions or bit manipulations. The B extension gives us access to more advanced instructions such as rotations (`rori`, `roli`), packing (`pack`, `packu`), and many others.

Unfortunately, this approach simulates neither an execution pipeline, nor a cache. While it allows us to execute RISC-V binary files, the results may be biased towards some optimization practices, leading to significant differences between implementations as shown in our benchmarks (see Section 5.4).

5.3 OPTIMIZED ALGORITHMS

Optimized cryptographic implementation are usually written directly in assembly with the idea to prevent the compiler from introducing bugs or weaknesses, and to further improve the speed of the software. By making sure we do not branch on secret data and considering the small size of the RISC-V ISA, we trust the compiler to match our implementations.

We call "Optimized C" the translation of an assembly implementation back into C, making use of `uint32_t` such that the C code mimics the assembly instructions. The underlying idea is to have the compiler further optimize our code and take care of the register allocation.

We now describe the algorithms we optimized and some of the implementation strategies we used.

5.3.1 Gimli

Previously described in details in Chapter 4, Gimli [Ber+17a] is a lightweight scheme proposed by Bernstein, Kölbl, Lucks, Massolino, Mendel, Nawaz, Schneider, Schwabe, Standaert, Todo, and Viguier.

Its design puts an emphasis on cross-platform performance and simplicity. The code is compact and uses only logical operations and shifts. The absence of additions allows to "interleave" implementations for platform with different register size than 32 bits. An implementation for RISC-V-64 with the B extension would likely be using such strategy.

On RISC-V-32 we are able to get speed-ups on both Gimli-Hash and Gimli-Cipher by optimizing the underlying permutation Gimli. Instead of using the order of instructions presented in Chapter 4, more precisely in Code 4.5.2; we reschedule the order of instructions to avoid swap operations.

BOUNDS AND OPTIMIZATIONS. We optimize GIMLI by first having a deeper look at the inner permutation and by computing the lower bound of the number of cycles used. GIMLI's state representation uses twelve 32-bit words which fit easily into the 27 general-use 32-bit registers. [Ber+17a] shows that only 2 additional registers are required in order to compute the column operations; as a result, in a fully unrolled implementation, the only cycles necessary in the computation are the ones required by the logical operations.

On each application of the round function, GIMLI uses 2 rotations, 6 XORS, 2 ANDS, 1 OR, and 4 shifts. All logical operations have a latency of 1 cycle, except for rotates which have a 3-cycle latency. A column operation requires thus 19 cycles; iterated over 4 columns and 24 rounds, this totals to 1824 cycles.

GIMLI uses 6 constants (each loaded in 2 cycles) derived every 4 rounds (an additional 5 cycles) before being XORed into the state (6 XORS, thus 6 cycles). When the permutation is not directly inlined and used as a function, it requires 12 loads and 12 stores to get the state into registers for an additional 48 cycles. Excluding the cycles needed to preserve some of the callee registers, we have a total of 1885 cycles.

As a baseline, the reference C code runs at 2178 cycles. By using careful scheduling of the instructions, and using a minimum number of registers—saving into the stack only 4 callees, our assembly implementation runs in 2092 cycles. The Optimized C version runs in 2132 cycles. This timing difference is explained by the compiler's use of the 12 callee registers, inducing a 40-cycle penalty.

By unrolling in C—the same approach could have been applied in assembly—over 8 rounds and propagating the swapping by renaming variable to avoid move operations, the compiler manages to achieve further speed-ups by getting down to 1900 cycles. Using this last implementation, we get a 19% speed-up for GIMLI-HASH and GIMLI-CIPHER (Table 5.1).

Table 5.1: Cycle counts for different GIMLI implementations on the SiFive board; GIMLI-HASH over 128 bytes of data, GIMLI-CIPHER over a 128 bytes message with 128 bytes of associated data. Compiled with Clang-10 and `-O3`.

	C-ref	Assembly	Opt. C	8-round Opt. C
GIMLI	2178	2092 *(−4%)*	2132 *(−2%)*	1900 *(−1%)*
GIMLI-HASH	23120	20812 *(−1%)*	21055 *(−9%)*	18678 *(−1%)*
GIMLI-CIPHER	44423	39583 *(−1%)*	40816 *(−8%)*	35853 *(−1%)*

5.3.2 SPARKLE

SPARKLE [Bei+20] is a family of cryptographic permutations based on the block cipher SPARX [Din+16b]. SCHWAEMM (an AEAD cipher scheme) and ESCH (a hash function) follow a not hermetic design approach, and share SPARKLE as the underlying permutation. The SPARKLE permutation is a classic ARX design, which, unlike most ARX constructions, provides security guarantees with regard to differential and linear cryptanalysis based on the long trail strategy (LTS) [Din+16b]. SCHWAEMM and ESCH work on a relatively small state, which is only 256 bits for the most lightweight instance of SCHWAEMM and 384 bits for the lightest variant of ESCH. The biggest possible state size with 512 bits can be applied by both schemes. Both algorithms employ the sponge construction.

Two instances for hashing were proposed in [Bei+20], i. e., ESCH256 and ESCH384, which allow processing messages of arbitrary length and output a digest of 256 bit, and 384 bit, length, respectively. ESCH256, the main instance of ESCH and the one considered in our work, uses the 384-bit SPARKLE permutation and has a claimed security level of 128 bits.

All the four instances for authenticated encryption with associated data proposed in [Bei+20], i. e., SCHWAEMM128-128, SCHWAEMM256-128, SCHWAEMM192-192 and SCHWAEMM256-256 use a variation of the BEETLE mode of operation first presented in [Cha+18], which in turn is based on the duplexed sponge construction. We focus again on the main version SCHWAEMM256-128, which uses the 384 bit SPARKLE (SPARKLE384) permutation, with a rate of $r = 256$ bit and a capacity of $c = 128$ bit, claiming a security level of 120 bits.

SPARKLE384 requires 50 rotations, 68 XORS, 24 ADDS, and 2 shifts for a single round. With the exception of rotation (3 cycles), all operations have a latency of 1 cycle. Thus, iterated over 7 or 11 rounds this totals to an estimated lower bound of 1708 cycles, and 2684 cycles respectively. For further details, we refer to the specification [Bei+20].

LOOP UNROLLING. Although unrolling the main loop within the SPARKLE permutation over 7 or 11 rounds results in a significant speed-up (see Table 5.3) when using instruction cache (like the SiFive core used in this work, see Section 5.2.3), this leads to significantly worse results in the case of AEAD (see Table 5.2).

ROUND CONSTANTS. In this assembly-specific optimization, we speed-up the permutation by increasing the space required. In every round of the permutation, each of the six ARX-boxes uses the same round constant in their computations. The idea is to avoid the loading of the constants for the ARX-boxes in every round by loading and saving these 6 constants in the registers before the transformation. This comes with the cost of dedicating 6 registers to these constants.

This optimization can be applied in the loop as well as in the unrolled variant of the implementation. In the unrolled implementation, we further reduce the loading of round constants, since these 6 constants are also being used as the round constants that are added to the state every round. In the 7-round variant of the permutation, we save the loading of the first 6 round constants and only have to load the 7th constant. In the 11-round variant of the permutation, we only have to load the 8th constant extra. The other three are already loaded because there are only 8 round constants defined and the selection index is calculated modulo 8. In the loop-unrolled implementation we reduce the instruction count by 72, but have to use 6 more registers.

Table 5.2 shows the achieved speed-up for Schwaemm256-128, Table 5.3 presents the achieved results for Esch256.

Table 5.2: Cycle counts for different Schwaemm256-128 implementations on the SiFive board; encryption over a 128 bytes of message with 128 bytes of associated data.

Platform	Compiler	Opt.	Opt. C	Looped + round cst ASM	Loop-unrolled Opt. C
SiFive	Clang-10	`-O3`	72286	43877 *(−40%)*	1059813 *(+94%)*
SiFive	Clang-9	`-O3`	73387	44558 *(−40%)*	1709958 *(+95%)*
SiFive	GCC	`-O3`	71271	42634 *(−40%)*	1790566 *(+95%)*
riscvOVPsim	Clang-10	`-O3`	20842	20840 *(±0%)*	20277 *(−3%)*
riscvOVPsim	Clang-9	`-O3`	20842	20840 *(±0%)*	20277 *(−3%)*
riscvOVPsim	GCC	`-O3`	20762	20161 *(−2%)*	20010 *(−3%)*
VexRiscv	GCC	`-O2`	25464	27018 *(+6%)*	24769 *(−3%)*

5.3.3 Saturnin

Saturnin [Can+20] is the NIST lightweight candidate designed by Canteaut, Duval, Leurent, Naya-Plasencia, Perrin, Pornin, and Schrottenloher. By building on top of a 256-bit block cipher with a 256-bit key, they describe three constructions for hashing (Saturnin-Hash) and

Table 5.3: ESCH256 cycle counts on each platform. The hashing operation hashes 128 bytes of data.

Platform	Compiler	Opt.	Opt. C	Loop-unrolled Opt. C
SiFive	Clang-10	`-O3`	62734	34664 *(−44%)*
SiFive	Clang-9	`-O3`	63893	28952 *(−54%)*
SiFive	GCC	`-O3`	58193	33331 *(−42%)*
riscvOVPsim	Clang-10	`-O3`	17439	16552 *(−5%)*
riscvOVPsim	Clang-9	`-O3`	17439	16552 *(−5%)*
riscvOVPsim	GCC	`-O2`	17849	17231 *(−3%)*
VexRiscv	GCC	`-O2`	18874	17753 *(−6%)*

authenticated encryption of small (SATURNIN-SHORT) and large data segment (SATURNIN-CIPHER). This last AEAD scheme uses the counter mode and a separate MAC.

We ported to our benchmark platform the reference implementation and both the 32-bit optimized "bs32" and "bs32x" C implementations [Can+20, Section 3.4.2]. The "bs32" and "bs32x" implementations both implement SATURNIN in a 32× bitsliced fashion. Their difference is that "bs32" bitslices *inside* of blocks, whereas "bs32x" bitslices *across* blocks. When comparing the two bitsliced implementations, "bs32" showed a consistently better performance than the other, albeit sometimes with a small margin. We decided that "bs32" would be the preferred implementation to use on our platforms.

In all the implementations, we tweaked the code to make sure that any constants would be loaded from SRAM, instead of (the relatively slow) SPI flash. This considerably improved the performance of the bitsliced implementations.

In the end, we see that the Optimized C implementation is considerably faster than the reference implementation in terms of performance, with generally a speed-up by a factor of 2. Another interesting property from the results in Tables 5.4 and 5.5 is the performance stability of the implementations across compilers. Where the "bs32" performance

Table 5.4: SATURNIN-HASH cycle counts on each platform. The hashing operation hashes 128 bytes of data.

Platform	Compiler	Opt.	Ref.	bs32
SiFive	Clang-10	`-O3`	49433	28199 *(−43%)*
SiFive	Clang-9	`-O3`	52868	30483 *(−42%)*
SiFive	GCC	`-O3`	78110	30321 *(−61%)*
riscvOVPsim	Clang-10	`-O3`	46946	27070 *(−42%)*
riscvOVPsim	Clang-9	`-O3`	48785	27972 *(−43%)*
riscvOVPsim	GCC	`-O3`	76211	29030 *(−61%)*
VexRiscv	GCC	`-O2`	103325	32169 *(−69%)*

is very stable—with cycle counts generally varying less than 10%—the performance of the reference implementation varies a lot with different compiler versions. Nonetheless, we see that newer compiler versions seem to produce faster code.

Table 5.5 illustrates the fact that the greedy unrolling and inlining by GCC with `-O3` results in major speed-up on simulators. However once tested on a physical device such as the SiFive development board (Section 5.2.3), this results in a code too large for the 16KB cache, inducing in a slowdown by a factor of 5.

5.3.4 ASCON

ASCON [Dob+16a] is a scheme proposed by Dobraunig, Eichlseder, Mendel and Schläffer. It uses a very small 320-bit state which allows it to fit in registers on most systems. The authors introduce multiple variants of ASCON AEAD as well as a hashing scheme. We focus our efforts on the ASCON-128 AEAD variant. We expect that our results translate fairly well to the other variants and the hashing scheme as they are very similar.

We use the ASCON C [Asc] repository as a baseline, more specifically we use the reference, the 64-bit optimized, and the 32-bit interleaved implementations as starting point for our optimizations.

Table 5.5: SATURNIN-CIPHER cycle counts on each platform. The cipher encrypts 128 AD bytes and 128 message bytes.

platform	Compiler	Opt.	Ref.	bs32	bs32x
SiFive	Clang-10	`-O3`	121651	59368 *(−51%)*	68792 *(−43%)*
SiFive	Clang-9	`-O3`	106665	62743 *(−41%)*	91511 *(−14%)*
SiFive	GCC	`-O3`	151428	60817 *(−60%)*	5210541 *(×34%)*
SiFive	GCC	`-Os`	183464	65469 *(−64%)*	138187 *(−24%)*
riscvOVPsim	Clang-10	`-O3`	93184	55154 *(−41%)*	61077 *(−34%)*
riscvOVPsim	Clang-9	`-O3`	96540	55617 *(−42%)*	63767 *(−33%)*
riscvOVPsim	GCC	`-O3`	145734	57366 *(−61%)*	75646 *(−48%)*
VexRiscv	GCC	`-O2`	202226	65015 *(−68%)*	88278 *(−56%)*

IMPROVED FORMULA. First we optimize the inner permutation by improving the ASCON S-box formula (Figure 5.1). We reduce the number of required instructions from 22 to 17 and the number of temporary registers from 5 to 3 at the cost of less potential for parallelism. Instruction-level parallelism —such as out-of-order execution— is common in high-end CPUs but not so common in lightweight platforms like our RISC-V targets. This optimization gives us a 10% speed-up for both the assembly and Optimized C implementations (Table 5.6).

Table 5.6: Cycle counts for the different ASCON's round functions over 6 rounds; Compiled with Clang-10 and `-O3`.

Platforms	C-ref	Assembly	Optimized C
SiFive	832	750 *(−10%)*	750 *(−10%)*
riscvOVPsim	830	748 *(−10%)*	748 *(−10%)*

Figure 5.1: These formulas compute the Ascon S-box in 17 operations (once the duplicate operations are taken out); o_n indicates output bit n and i_n indicates input bit n.

$$
\begin{aligned}
o_0 &\leftarrow i_3 \oplus i_4 \oplus (i_1 \vee (i_0 \oplus i_2 \oplus i_4)) \\
o_1 &\leftarrow i_0 \oplus i_4 \oplus ((i_1 \oplus i_2) \vee (i_2 \oplus i_3)) \\
o_2 &\leftarrow i_1 \oplus i_2 \oplus (i_3 \vee \neg i_4) \\
o_3 &\leftarrow i_1 \oplus i_2 \oplus (i_0 \vee (i_3 \oplus i_4)) \\
o_4 &\leftarrow i_3 \oplus i_4 \oplus (i_1 \wedge \neg(i_0 \oplus i_4))
\end{aligned}
$$

BIT INTERLEAVING. We also compare the C implementation optimized for 32-bit interleaving. It performs the worst of all others including the baseline implementation. Bit interleaving allows 32-bit rotations to model 64-bit rotations efficiently, unfortunately our targets does not support 32-bit rotations. We expect this implementation will perform better when targeting RISC-V cores that come with the B extension, which adds rotation instructions.

OPTIMIZED 64 BITS. Finally, we compare the C implementation optimized for 64-bit processors. On RISC-V cores without the B extension, the 64-bit operations are compiled to 32-bit operations in a straightforward manner and the compiler has no trouble with it. As RISC-V does not support misaligned memory access, we had to modify the code to handle the authentication tag.

While on the RISC-V OVP simulator the 64-bit optimized version is 7% faster than the baseline, testing it on the SiFive board reveals significant slowdowns due to the code not fitting in the 16KB instruction cache.

Our final implementation makes use of the improved S-box formula in a 6-round unrolled Optimized C permutation. By folding the processing of associated data and message we are able to reuse the code and have to compiled code fit in the instruction cache. Applying these modifications, we achieve our best results: 15% faster than the baseline (Table 5.7).

5.3.5 *Delirium*

ELEPHANT [Bey+19] is a family of lightweight authenticated encryption schemes. The mode of ELEPHANT is a nonce-based encrypt-then-MAC construction, where encryption is performed using counter mode based on permutation masked using LFSRs. One of the instances of ELEPHANT is ELEPHANT-KECCAK-f[200], also called DELIRIUM, which

Table 5.7: Cycle counts for different Ascon implementations in OVP sim for encrypting 128 bytes of message and 128 bytes of associated data; compiled with Clang-10 and `-O3`.

Implementation	OVP sim	SiFive
ref. & default permutation	31990	32038
ref. & asm permutation	28988 *(−9%)*	29036 *(−9%)*
ref. & inlined Optimized C perm.	27489 *(−14%)*	27703 *(−14%)*
bit interleaved inline permutation	32001 *(±0%)*	1559691 *(×49%)*
opt. 64-bit & default unrolled perm.	29646 *(−7%)*	1191702 *(×37%)*
opt. 64-bit & asm permutation	29090 *(−9%)*	29170 *(−9%)*
opt. 64-bit & fully unrolled Opt. C perm.	27589 *(−14%)*	809631 *(×25%)*
opt. 64-bit & 6-round unrolled Opt. C perm.	27184 *(−15%)*	27271 *(−15%)*

uses Keccak as its permutation primitive. Delirium has a state size of 200 bits and claimed a security level of 127 bits. We optimize Delirium by exploiting Elephant 's possibility for parallelization by using bit-interleaving.

BIT INTERLEAVING. In order to make full use of the 32-bit registers, we combine four blocks of byte-sized elements into one block of 4-byte elements. Thus, we can process four blocks at the same time and our state representation changes to an array of 25 32-bit words (5-by-5-by-32) with a total size of 800 bits. In this new representation, one block amounts to four blocks in the standard representation.

There are two possible cases when transforming blocks before encrypting/decrypting to the new representation. The first and the easiest case is when the amount of blocks that need to be transformed is a multiple of four. This means that all groups of four blocks consisting of 8-bit words can be interleaved to make one block of 32-bit words. The second case is when the amount of blocks is not a multiple of four. Since the new representation needs four "old" blocks to transform into one new block, we have to use padding blocks filled with zero values to add to make the amount of blocks to a multiple of four.

After encryption/decryption, when transforming back to a byte representation of the data, we have to de-interleave each interleaved

32-bit block back to four blocks of bytes. Since it is possible that the amount of original blocks was not a multiple of four, we need to make sure none of the data from the added padding blocks gets joined in the output data. This can be done by cutting off any output data which exceeds the message-length variable.

As shown in Table 5.8, we note that shorter inputs perform worse in the optimized implementation. This is because the effort of interleaving data to process four blocks simultaneously is wasted if there are very few blocks to process.

Table 5.8: Cycle counts for different ELEPHANT-KECCAK-f[200] implementations on the SiFive board; encryption over a 32/64/128 bytes message with 32/64/128 bytes of associated data.

Platform	Compiler	Message length	Data length	C-ref	Bit interleaved
SiFive	GCC	16	16	66541	73989 *(+11%)*
SiFive	GCC	32	32	91837	74385 *(−19%)*
SiFive	GCC	64	64	143181	74890 *(−47%)*
SiFive	GCC	128	128	245100	113031 *(−53%)*
SiFive	Clang-9	128	128	294643	160494 *(−46%)*
SiFive	Clang-10	128	128	241975	145936 *(−40%)*
riscvOVPsim	GCC	32	32	64651	66690 *(+3%)*
riscvOVPsim	GCC	64	64	102138	66805 *(−35%)*
riscvOVPsim	GCC	128	128	176086	101966 *(−42%)*
riscvOVPsim	Clang-9	128	128	168313	106904 *(−36%)*
riscvOVPsim	Clang-10	128	128	163973	103631 *(−37%)*

5.3.6 XOODYAK

XOODYAK [Dae+20], based on the XOODOO permutation [Dae+18a; Dae+18b], is a cryptographic scheme that is suitable for several symmetric-key functions, including hashing, encryption, MAC computation and authenticated encryption. XOODOO, according to its authors [Dae+18b], can be seen as a porting of the KECCAK-p [Ber+13b; Ber+13c] design approach to a GIMLI-shaped [Ber+17a] state.

XOODOO iteratively applies 12 rounds to a 384-bit state, which can be treated as 3 horizontal planes, each one consisting of 4 parallel 32-bit lanes. The choice of 12 rounds justifies a security claim in the hermetic philosophy. The claimed security strength for XOODYAK is 128 bits.

An estimated lower bound for cycles taken by XOODOO can be calculated as follows. It requires 24 rotations, 37 XORs, 12 ANDs, and 12 NEGs for a single round. With the exception of rotation (3 cycles), all operations take 1 cycle. Thus, iterated over 12 rounds this totals to 1596 cycles.

LANE COMPLEMENTING. The idea behind lane complementing, first proposed in the KECCAK implementation overview [Ber+13c], is to reduce the number of NEG instructions by complementing certain lanes before the transformation.

In XOODOO the state is ordered in 4 sheets, each containing 3 lanes with a width of 32-bit. The χ layer computes 3 XOR, 3 AND and 3 NEG operations for every sheet in the state. This sums up to 12 NEG operations per round and 144 NEG operations in total. In the default case, the χ transformation for every lane $a[i]$ in a sheet, with $0 \leq i \leq 2$ and index calculation modulo 3, can be calculated as shown in equation (5.1).

To highlight de NEG propagation, we use of $\overline{a}$ instead of $\neg a$.

$$a[i] \leftarrow a[i] \oplus (\overline{a[i+1]} \wedge a[i+2]) \tag{5.1}$$

For example, we now want to complement lane $a[2]$. Thus, the equation of lane $a[0]$ gets rearranged as follows:

$$a[0]' = a[0] \oplus (\overline{a[1]} \wedge \overline{\overline{a[2]}}) = a[0] \oplus \overline{a[1] \vee \overline{a[2]}} = \overline{a[0] \oplus (a[1] \vee \overline{a[2]})},$$
$$\overline{a[0]'} = a[0] \oplus (a[1] \vee \overline{a[2]}).$$

The complementation of $a[2]$ results in the cancellation of the negation of $a[1]$, the switch from an AND to an OR operation and the complement of $a[0]'$. Now we calculate all three lanes of a sheet with the complement of the lane $a[2] \leftarrow \overline{a[2]}$:

$$\begin{aligned} a[0] \leftarrow\ \overline{a[0]'} &= a[0] \oplus (a[1] \vee a[2]), \\ a[1] \leftarrow\ a[1]' &= a[1] \oplus (a[2] \wedge \overline{a[0]}), \\ a[2] \leftarrow\ \overline{a[2]'} &= a[2] \oplus (a[0] \wedge a[1]). \end{aligned}$$

It can be observed that we only need one complementation for this sheet, instead of three. For the computation of $a[1]$, $a[0]$ is complemented to be positive, because $a[0]$ was negated before. This example of lane complementing comes with the cost of applying the input mask $\overline{a[2]}$ and output mask $\overline{a[0]}$, $\overline{a[2]}$.

FINDING THE LOWEST NEG COUNT. The possible transformations of the boolean equations for a sheet are not fixed to one. Thus, there are multiple boolean equations that are still logically congruent, but may differ in the input and output mask. We want to find the boolean equations and input mask with the lowest possible number of NEG-instructions. To simplify this problem, we set the boolean equations to a fixed set and only care about the possible input patterns. Therefore, we employ an algorithm for finding the minimum NEG instruction count for a certain set of boolean equations.

After the application of the algorithm, we obtain an input mask and a sequence of boolean equations. This input mask is 2-round invariant, meaning that the input mask is the always same after every two rounds. Hence, it can be implemented as a loop and therefore have a smaller code size. The obtained input mask P is the following (denoted in x, y coordinates):

$$P = \{(0,0), (1,0), (2,0), (3,0)\}.$$

We reduce the number of NEG operations to exactly one thrid over 12 rounds. The application of our input and output mask, each costs 4 NEG operations. Due to a larger number of lanes in KECCAK, Stoffelen [Sto19] achieved a reduction to 20%.

Lane complementing is not an assembly-specific optimization. As shown in Tables 5.9 to 5.11, we achieve a very similar speed-up in assembly and in C.

Table 5.9: Cycle counts for different implementations of XOODYAK in AEAD mode GCC compiled with `-O2` in riscvOVPsim for encrypting 128 bytes of message and 128 bytes of associated data.

Implementation	riscvOVPsim
reference	105463
loop unrolled + lane complementing assembly	29574 *(−71%)*
loop unrolled + lane complementing Optimized C	28672 *(−72%)*

Table 5.10: Cycle counts for Xoodyak in hash mode on each platform, compiled with `-O3`. The hashing operation hashes 128 bytes of data.

Platform	Compiler	Ref.	Unrolled & lane comp.
SiFive	Clang-10	81349	17963 *(−78%)*
SiFive	Clang-9	88451	18865 *(−79%)*
SiFive	GCC	82741	17063 *(−79%)*
riscvOVPsim	Clang-10	18114	16845 *(−7%)*
riscvOVPsim	Clang-9	18059	16898 *(−6%)*
riscvOVPsim	GCC	23247	16614 *(−29%)*
VexRiscv	GCC `-O2`	261678	38378 *(−85%)*

Table 5.11: Cycle counts for Xoodyak in AEAD mode on each platform, compiled with `-O3`, for encrypting 128 bytes of message and 128 bytes of associated data.

Platform	Compiler	Ref.	unrolled & lane comp.
SiFive	Clang-10	103717	26246 *(−75%)*
SiFive	Clang-9	112414	27392 *(−76%)*
SiFive	GCC	103522	23238 *(−78%)*
riscvOVPsim	Clang-10	25002	23429 *(−6%)*
riscvOVPsim	Clang-9	25002	23429 *(−6%)*
riscvOVPsim	GCC	29775	21668 *(−28%)*
VexRiscv	GCC `-O2`	261678	38378 *(−85%)*

5.3.7 *AES*

In [Sto19], Stoffelen proposes two assembly implementations of AES: the first one is based on lookup tables, and the second one uses a bitsliced approach.

WITH A LOOKUP TABLES. When encrypting a single block of 16 bytes, multiple steps of the round function can be combined in a lookup table, also called T-table by Daemen and Rijmen in [DR02]. Note that this type of implementation is usually vulnerable to cache attacks [Ber05a; BM06; OST06]. Because none of our benchmarking platforms have a data cache, we believe this implementation is likely "safe" to use.

For his table-based implementation, Stoffelen makes use of the baseline instructions described in [BS08]. Most of the proposed optimizations by Bernstein and Schwabe are not applicable due to the small instruction set of the RISC-V architecture. The translation from assembly to C using `uint32_t` to simulate registers is straightforward, and the lookup table is converted to an array as `uint32_t variable[]`. The resulting benchmark shows no difference in timing between the two approaches (see Table 5.12).

Table 5.12: Cycle counts for the Assembly of [Sto19] and its translation to C on the SiFive board, compiled with Clang-10 and `-O3`.

	Assembly	Optimized C
Key schedule	342	342 *(±0%)*
1-block encryption	903	901 *(±0%)*

Note that if the table is declared as `const`, the compiler will place it in the `.rodata` segment. While this change does not have any impact on the verilator and the riscvOVPsim simulators, it induces a major slowdown in the case of the SiFive board as the SPI flash is significantly slower than the SRAM.

In order to prevent the compiler from messing with the pointer arithmetic, data pointers are kept in the `uint8_t*` type. This forces us to cast the pointer to `uint32_t*` before de-referencing to trigger the compiler to use the `lw` instruction.

```
Y0 = RK[0]; T0 = (uint32_t*)(LUT1 + ((*X0 & 0xff) << 4)); Y0 = Y0 ^ *T0;
Y1 = RK[1]; T1 = (uint32_t*)(LUT1 + ((*X1 & 0xff) << 4)); Y1 = Y1 ^ *T1;
Y2 = RK[2]; T2 = (uint32_t*)(LUT1 + ((*X2 & 0xff) << 4)); Y2 = Y2 ^ *T2;
Y3 = RK[3]; T3 = (uint32_t*)(LUT1 + ((*X3 & 0xff) << 4)); Y3 = Y3 ^ *T3;
```

Code 5.1: Code fragment of AES encryption

USING A BITSLICED APPROACH. When using AES in CTR or GCM mode, multiple blocks can be processed in parallel using a bitsliced implementation [RSD06; KS09; Kön08]. This strategy is often more efficient and avoids lookup tables, making the implementation more resistant against timing attacks.

By using the same approach as with lookup tables, we translate the assembly from [Sto19] back into C. While the key schedule it is slightly slower, this translation approach gives us a 4% speed-up in the case of the encryption in CTR mode as illustrated in Table 5.13.

Table 5.13: Cycle counts for the Assembly of [Sto19] and its translation to C on the SiFive board, compiled with Clang-10 `-O3`.

	Assembly	Optimized C
Key schedule	1248	1256 *(±0%)*
Encryption of 128 blocks	260695	249813 *(−4%)*

5.3.8 *Keccak*

We now have a look at the KECCAK-f family permutation—designed by Bertoni, Daemen, Peeters and Van Assche [Ber+13b]—, more precisely its 1600-bit instance found in the SHA-3 standard by NIST [Dwo15]. The permutation is used in multiple cryptographic constructions including future post-quantum candidates such as FrodoKEM [Bos+16], NewHope [Alk+16], SPHINCS+ [Ber+19] and others.

Stoffelen [Sto19] provides us with another optimized implementation for RISC-V inspired by the *Keccak implementation overview* [Ber+13c]. KECCAK-f[1600] works on a state composed of 25 64-bit lanes, in other words a total of 50 32-bits words. This is more than the number of register made available by the ISA, preventing the state from completely fitting in the registers. By using bit interleaving and other techniques, Stoffelen manages to reduce the number of cycles used.

We take his implementation and translate it back to C. We compile with GCC and `-Os` instead of `-O2` or `-O3` to get marginally faster results than the assembly implementation in [Sto19] (see Table 5.14).

Table 5.14: Cycle counts for the Assembly of Keccak [Sto19] and its translation to C on the SiFive board, compiled with GCC `-Os`.

	Assembly	Optimized C
KECCAK-f[1600]	13731	13336 *(−3%)*

5.4 COMPARISON WITH OTHER IMPLEMENTATIONS AND ADDITIONAL BENCHMARKS

Other implementations of lightweight candidates are publicly available; we chose to compare our work against the repository of Weatherley[2] [Wea20] as their implemented are *"focused on good performance in plain C on 32-bit embedded microprocessors"*.

As Clang-10 generally produces faster results than GCC with `-O3`, we used it to compile and benchmark every optimized C implementation provided by Weatherley. We measure the cycle counts for encryption of AEAD schemes for 128-byte messages with 128 bytes of associated data, and provide our results in Section 5.A.

In Table 5.15, we summarize the performance of our software and Weatherley's implementations.

Table 5.15: Cycle counts for AEAD mode on the SiFive and riscvOVPsim platform, compiled with Clang-10 `-O3`, for encrypting 128 bytes of message and 128 bytes of associated data.

Algorithm	Weatherley		Our results	
	OVP	SiFive	OVP	SiFive
GIMLI	37596	38530	35690 *(−5%)*	35853 *(−7%)*
SCHWAEMM256-128	20842	72286	20277 *(−3%)*	43877 *(−40%)*
SATURNIN	55367	152803	55154 *(−1%)*	59368 *(−61%)*
ASCON	41228	42562	27184 *(−34%)*	27271 *(−36%)*
DELIRIUM	110171	765235	103631 *(−6%)*	145936 *(−81%)*
XOODYAK	18852	64869	23451 *(+24%)*	26246 *(−60%)*

While on the OVP simulator most of our implementations produce just slightly better results with an average at -4% cycle counts; when using the SiFive board, the unrolled implementation by Weatherley suffers heavily from the 16KB instruction cache. This makes our RISC-V-optimized code on average 47.5% faster.

2 `https://github.com/rweather/lightweight-crypto`, commit `52c8281`

5.5 THE RISC-V BITMANIP EXTENSION

While still being in development, the bit manipulation extension[3] (B) allows interesting optimizations. We present Table 5.16 the impact and possible gains by enabling such extension.

However, the provided instructions in this extension are still a work in progress; Therefore, some of them and their specification may change before being accepted as a standard by the RISC-V Foundation. The presented performance figures were compiled using the *riscv-bitmanip* branch of the GCC compiler[4] using the flags `-O2 -mcmodel=medany -march=rv64gcb -mabi=lp64d` and calculated with the Spike RISC-V ISA Simulator[5].

primitive	variant	w/o B ext.	with B ext.
GIMLI Hash	C-ref.	27628	22688 *(−18%)*
GIMLI Hash	C-opt.	26771	22080 *(−18%)*
GIMLI Hash	8-round C-opt.	22224	16618 *(−25%)*
ESCH256	C-ref.	20605	13891 *(−33%)*
ESCH256	loop unrolled	17585	11586 *(−34%)*
AES LUT	C-opt.	3647	1578 *(−57%)*
AES CTR-Bitsliced	C-opt.	1509	1431 *(−5%)*
SATURNIN	C-ref.	83516	80866 *(−3%)*
SATURNIN	bs32	33087	30943 *(−6%)*
XOODYAK	C-ref.	28492	22440 *(−21%)*
XOODYAK	unrolled & complementing	19123	14169 *(−26%)*
KECCAK-f[1600]	C-opt.	14633	12402 *(−15%)*
KECCAK-f[200]	C-opt.	9143	6119 *(−33%)*

Table 5.16: Cycle counts comparison for some primitives using the RISC-V bit manipulation extension.

3 `https://github.com/riscv/riscv-bitmanip`, commit a05231d
4 `https://github.com/riscv/riscv-gcc/tree/riscv-bitmanip`, commit 8b86205
5 `https://github.com/riscv/riscv-isa-sim`, commit 958dcdc

5.6 CONCLUSION

We described how multiple lightweight NIST candidates such as GIMLI, SPARKLE, SATURNIN, ASCON, DELIRIUM, and XOODYAK can be efficiently implemented. With strategies such as loop unrolling, we are able to write assembly code close to the lower bound given by the number instructions arithmetic. By translating our assembly implementation back into C, we get the compiler to further optimize our results.

Using the AES and KECCAK assembly implementations from Stoffelen [Sto19], we also show that our approach is applicable to existing code bases, and may provide slightly improved results while increasing the readability and maintainability of the code.

We use the HiFive1 development board to illustrate that algorithms need to be tested on physical devices in order to guarantee useful optimized implementations (Table 5.17). Although strategies such as fully unrolled loops may work nicely in simulated environments such as riscvOVPsim; they will fail at length on physical devices with e. g., a 16KB instruction cache.

As the NIST lightweight competition is currently taking place, we hope our results will be found useful by the candidates' implementors and designers.

APPENDIX OF CHAPTER 5

In the following, we give our benchmark of Weatherley's implementations [Wea20] of the lightweight candidates.

5.A BENCHMARK OF OTHER IMPLEMENTATIONS

RISC-V hardware availability is scarce and most implementations are based on simulators. Having the availability of both—board and simulator, we benchmark the work of Weatherley [Wea20] (commit `52c8281`) and present our measures in Table 5.17. As such, we illustrate the need of testing implementations on boards and constrained environment.

Table 5.17: Cycle counts for different ciphers implementations clang-10 compiled with `-O3` for encrypting 128 bytes of message and 128 bytes of associated data.

Implementation	riscvOVPSim	SiFive
ACE_AEAD_ENCRYPT	206040	1959318 *(+851%)*
ASCON128_AEAD_ENCRYPT	41228	42562 *(+3%)*
ASCON128A_AEAD_ENCRYPT	28284	29457 *(+4%)*
ASCON80PQ_AEAD_ENCRYPT	41264	42639 *(+3%)*
COMET_128_CHAM_AEAD_ENCRYPT	21648	22570 *(+4%)*
COMET_64_SPECK_AEAD_ENCRYPT	24429	25515 *(+4%)*
COMET_64_CHAM_AEAD_ENCRYPT	60197	61552 *(+2%)*
DRYGASCON128_AEAD_ENCRYPT	96365	98194 *(+2%)*
DRYGASCON256_AEAD_ENCRYPT	120862	123710 *(+2%)*
ESTATE_TWEGIFT_AEAD_ENCRYPT	50051	855204 *(+1609%)*
DUMBO_AEAD_ENCRYPT	1365382	2927910 *(+114%)*

JUMBO_AEAD_ENCRYPT	1465147	3265908 *(+123%)*
DELIRIUM_AEAD_ENCRYPT	110171	763144 *(+593%)*
FORKAE_PAEF_64_192_AEAD_ENCRYPT	543575	818122 *(+51%)*
FORKAE_PAEF_128_192_AEAD_ENCRYPT	349468	550627 *(+58%)*
FORKAE_PAEF_128_256_AEAD_ENCRYPT	349485	510584 *(+46%)*
FORKAE_PAEF_128_288_AEAD_ENCRYPT	457833	642481 *(+40%)*
FORKAE_SAEF_128_192_AEAD_ENCRYPT	350409	523108 *(+49%)*
FORKAE_SAEF_128_256_AEAD_ENCRYPT	350767	512754 *(+46%)*
GIFT_COFB_AEAD_ENCRYPT	28277	28642 *(+1%)*
GIMLI24_AEAD_ENCRYPT	37596	38530 *(+2%)*
GRAIN128_AEAD_ENCRYPT	71378	71687 *(−0%)*
HYENA_AEAD_ENCRYPT	37317	136055 *(+265%)*
ISAP_KECCAK_128A_AEAD_ENCRYPT	243243	640729 *(+163%)*
ISAP_ASCON_128A_AEAD_ENCRYPT	204619	222540 *(+9%)*
ISAP_KECCAK_128_AEAD_ENCRYPT	1190410	2467483 *(+107%)*
ISAP_ASCON_128_AEAD_ENCRYPT	585890	605107 *(+3%)*
KNOT_AEAD_128_256_ENCRYPT	51298	224600 *(+338%)*
KNOT_AEAD_128_384_ENCRYPT	30982	102381 *(+230%)*
KNOT_AEAD_192_384_ENCRYPT	69190	228613 *(+230%)*
KNOT_AEAD_256_512_ENCRYPT	97648	266707 *(+173%)*

LOTUS_AEAD_ENCRYPT	84658	1006509 *(+1089%)*
LOCUS_AEAD_ENCRYPT	86693	1008575 *(+1063%)*
ORANGE_ZEST_AEAD_ENCRYPT	84917	159953 *(+88%)*
ORIBATIDA_256_AEAD_ENCRYPT	104129	106399 *(+2%)*
ORIBATIDA_192_AEAD_ENCRYPT	118296	121481 *(+3%)*
PHOTON_BEETLE_128_AEAD_ENCRYPT	157558	298030 *(+89%)*
PHOTON_BEETLE_32_AEAD_ENCRYPT	591344	1124306 *(+90%)*
PYJAMASK_128_AEAD_ENCRYPT	287809	316105 *(+10%)*
PYJAMASK_MASKED_128_AEAD_ENCRYPT	1407899	1602733 *(+14%)*
PYJAMASK_96_AEAD_ENCRYPT	284920	308484 *(+8%)*
PYJAMASK_MASKED_96_AEAD_ENCRYPT	1391787	1500329 *(+8%)*
ROMULUS_N1_AEAD_ENCRYPT	213113	218841 *(+3%)*
ROMULUS_N2_AEAD_ENCRYPT	197988	201165 *(+2%)*
ROMULUS_N3_AEAD_ENCRYPT	166744	309093 *(+85%)*
ROMULUS_M1_AEAD_ENCRYPT	282764	325705 *(+15%)*
ROMULUS_M2_AEAD_ENCRYPT	270063	291863 *(+8%)*
ROMULUS_M3_AEAD_ENCRYPT	231497	238422 *(+3%)*
SKINNY_AEAD_M1_ENCRYPT	248751	251707 *(+1%)*
SKINNY_AEAD_M2_ENCRYPT	248747	251677 *(+1%)*
SKINNY_AEAD_M3_ENCRYPT	248721	251578 *(+1%)*

SKINNY_AEAD_M4_ENCRYPT	248717	251548 *(+1%)*
SKINNY_AEAD_M5_ENCRYPT	211871	215308 *(+2%)*
SKINNY_AEAD_M6_ENCRYPT	211820	215179 *(+2%)*
SCHWAEMM_256_128_AEAD_ENCRYPT	20842	72286 *(+247%)*
SCHWAEMM_128_128_AEAD_ENCRYPT	23918	111207 *(+365%)*
SCHWAEMM_192_192_AEAD_ENCRYPT	28112	98569 *(+251%)*
SCHWAEMM_256_256_AEAD_ENCRYPT	30452	107680 *(+254%)*
SPIX_AEAD_ENCRYPT	93090	348608 *(+274%)*
SUNDAE_GIFT_0_AEAD_ENCRYPT	44214	57268 *(+30%)*
SUNDAE_GIFT_64_AEAD_ENCRYPT	45838	58952 *(+29%)*
SUNDAE_GIFT_96_AEAD_ENCRYPT	45874	58996 *(+29%)*
SUNDAE_GIFT_128_AEAD_ENCRYPT	45906	58996 *(+29%)*
SATURNIN_AEAD_ENCRYPT	55367	152798 *(+176%)*
SATURNIN_SHORT_AEAD_ENCRYPT	42	55 *(+31%)*
SPOC_128_AEAD_ENCRYPT	69789	839487 *(+1103%)*
SPOC_64_AEAD_ENCRYPT	113672	1651442 *(+1353%)*
SPOOK_128_512_SU_AEAD_ENCRYPT	34778	285992 *(+722%)*
SPOOK_128_384_SU_AEAD_ENCRYPT	46390	195868 *(+322%)*
SPOOK_128_512_MU_AEAD_ENCRYPT	34792	285764 *(+721%)*
SPOOK_128_384_MU_AEAD_ENCRYPT	46409	195708 *(+322%)*

SUBTERRANEAN_AEAD_ENCRYPT	127707	132338 *(+4%)*
TINY_JAMBU_128_AEAD_ENCRYPT	34776	37952 *(+9%)*
TINY_JAMBU_192_AEAD_ENCRYPT	37590	40789 *(+9%)*
TINY_JAMBU_256_AEAD_ENCRYPT	40394	43865 *(+9%)*
WAGE_AEAD_ENCRYPT	788038	14336632 *(+1719%)*
XOODYAK_AEAD_ENCRYPT	18852	64869 *(+244%)*

6 CRYPTANALYSIS OF MORUS

In this chapter, we show the existence of linear biases in the output of the authenticated encryption scheme Morus. More precisely, we show that when encrypting a fixed plaintext multiple times, a linear correlation exists between some bits at the output of the cipher. Moreover, the bias depends purely on the plaintext, and not on the secret key of the cipher. In principle, this property could be used to recover unknown bits of a plaintext encrypted a large number of times, provided an initial segment of the plaintext is known.

This chapter is organized as follows. After a short introduction (Section 6.1) we provide a brief description of Morus in Section 6.2. In Section 6.3, we introduce MiniMorus, an abstraction of Morus based on a certain class of rotational symmetries. We analyze this simplified scheme in Section 6.4 and provide a ciphertext-only linear approximation with a weight of 16. We then extend our result to the full scheme in Section 6.5, showing a correlation in the keystream over 5 steps, and discuss the implications of our observation for the security of Morus in Section 6.6. In Section 6.7, we present our results on the security of Morus with round-reduced initialization (in a nonce-misuse setting) or finalization. We conclude in Section 6.8.

6.1 INTRODUCTION

To address the growing need for modern authenticated encryption designs for different application scenarios, the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) was launched in 2013 [CAE13]. The goal of this competition is to select a final portfolio of AEAD designs for three different use-cases: (1) lightweight hardware characteristics, (2) high-speed software performance, and (3) robustness. The competition attracted 57 first-round submissions, 7 of which were recently selected as finalists in the fourth selection round.

Morus is one of the three finalists for use-case (2), together with OCB [KR14] and AEGIS [WP16; WP13]. This family of authenticated ciphers by Wu and Huang [WH16] provides three main variants: Morus-640 with a 128-bit key and Morus-1280 with either a 128-bit or a 256-bit key. The design approach is reminiscent of classical stream cipher designs and continuously updates a relatively large state with a few fast operations. Morus can be efficiently implemented in both software and hardware; in particular, the designers claim that the software performance even surpasses AES-GCM implementations using

Intel's AES-NI instructions, and that MORUS is the fastest authenticated cipher not using AES-NI [WH16].

RELATED WORK. In the MORUS submission document, the designers discuss the security of MORUS against several attacks, including algebraic, differential, and guess-and-determine attacks. The main focus is on differential properties, and not many details are given for other attack vectors. In third-party analysis, Mileva et al. [MDV15] propose a distinguisher in the nonce-reuse setting and practically evaluate the differential behavior of toy variants of MORUS. Shi et al. [Shi+16] analyze the differential properties of the finalization reduced to 2 out of 10 steps, but find no attacks. Dwivedi et al. [Dwi+16] discuss the applicability of SAT solvers for state recovery, but the resulting complexity of 2^{370} for MORUS-640 is well beyond the security claim. Dwivedi et al. [DMW17] also propose key-recovery attacks for MORUS-1280 if initialization is reduced to 3.6 out of 16 steps, and discuss the security of MORUS against internal differentials and rotational cryptanalysis. Salam et al. [Sal+17] apply cube attacks to obtain distinguishers for up to 5 out of 16 steps of the initialization of MORUS-1280 with negligible complexity. Additionally, Kales et al. [KEM17] and Vaudenay and Vizár [VV17] independently propose state-recovery and forgery attacks on MORUS in a nonce-misuse setting with negligible data and time complexities. Finally, a keystream correlation similar in nature to our main attack was uncovered by Minaud [Min14] on the authenticated cipher AEGIS [WP16; WP13], another CAESAR finalist, which shares the same overall structure as MORUS, but uses a very different, AES-based state update function.

OUR CONTRIBUTIONS. Our main contribution is a keystream distinguisher on full MORUS-1280, built from linear approximations of its core `StateUpdate` function. In addition, we provide results for round-reduced MORUS, targeting both the initialization or finalization phases of the cipher.

In more detail, our main result is a linear approximation [Mat93; MY92] linking plaintext and ciphertext bits spanning five consecutive encryption blocks. Moreover, the correlation does not depend on the secret key of the cipher. In principle, this property could be used as a known-plaintext distinguisher, or to recover unknown bits of a plaintext encrypted a large number of times. For MORUS-1280 with 256-bit keys, the linear correlation is 2^{-76} and can be exploited using about 2^{152} encrypted blocks.

To the best of our knowledge, this is the first attack on full MORUS in the nonce-respecting setting. We note that rekeying does not prevent the attack: the biases are independent of the secret encryption key and nonce, and can be exploited for plaintext recovery as long as a given plaintext segment is encrypted sufficiently often, regardless of whether each encryption uses a different key. A notable feature of the linear

trail underpinning our attack is also that it does not depend on the values of rotation constants: a very similar trail would exist for most choices of round constants.

To obtain this result, we propose a simplified abstraction of MORUS, called MINIMORUS. MINIMORUS takes advantage of certain rotational symmetries in MORUS and simplifies the description and analysis of the attack. We then show how the attack can be extended from MINIMORUS to the real MORUS. To confirm the validity of our analysis, we practically verified the correlation of the full linear trail for MINIMORUS, as well as the correlation of trail fragments for the full MORUS. Our analysis is also backed by a symbolic evaluation of the full trail equation and its correlation on all variants of MORUS.

In addition to the previous attack on full MORUS, we provide two secondary results: (1) we analyze the security of MORUS against forgery attacks with round-reduced finalization; and (2) we analyze its security against key recovery in a nonce-misuse setting, with round-reduced initialization. While this extra analysis does not threaten full MORUS, it complements the main result to provide a better overall understanding of the security of MORUS. More precisely, we present a forgery attack for round-reduced MORUS-1280 with success probability 2^{-88} for a 128-bit tag if the finalization is reduced to 3 out of 10 steps. This nonce-respecting attack is based on a differential analysis of the padding rule. The second result targets round-reduced initialization with 10 out of 16 steps, and extends a state-recovery attack (which can be mounted e. g., in a nonce-misuse setting) into a key-recovery attack.

6.2 PRELIMINARIES

MORUS is a family of authenticated ciphers designed by Wu and Huang [WH16]. An instance of MORUS is parameterized by a secret key K. During encryption, it takes as input a plaintext message M, a nonce N, and possibly some associated data A, and outputs a ciphertext C together with an authentication tag T. In this section, we provide a brief description of MORUS and introduce the notation for linear approximations.

6.2.1 *Specification of* MORUS

The MORUS family supports two internal state sizes: 640 and 1280 bits, referred to as MORUS-640 and MORUS-1280, respectively. Three parameter sets are recommended: MORUS-640 supports 128-bit keys and MORUS-1280 supports either 128-bit or 256-bit keys. The tag size is 128 bits or shorter. The designers strongly recommend using a 128-bit tag. With a 128-bit tag, integrity is claimed up to 128 bits and confidentiality is claimed up to the number of key bits (Table 6.1).

Table 6.1: Security goals of MORUS.

	Confidentiality (bits)	Integrity (bits)
MORUS-640-128	128	128
MORUS-1280-128	128	128
MORUS-1280-256	256	128

STATE. The internal state of MORUS is composed of five q-bit *registers* S_i, $i \in \{0,1,2,3,4\}$, where $q = 128$ for MORUS-640 and $q = 256$ for MORUS-1280. The internal state of MORUS may be represented as $S_0 \| S_1 \| S_2 \| S_3 \| S_4$.

Registers are themselves divided into four $q/4$-bit *words*. Throughout the chapter, we denote the word size by $w = q/4$, i. e., $w = 32$ for MORUS-640 and $w = 64$ for MORUS-1280. As a result, an efficient implementation of MORUS will represent the state with 5 vector registers, making use of SSE and AVX2 instructions: `__m128i` and `__mm256i` for respectively the 640-bit and 1280-bit version.

The encryption process of MORUS consists of four parts: initialization, associated data processing, encryption, and finalization. During the initialization phase, the value of the state is initialized using a key and nonce. The associated data and the plaintext are then processed block by block. Then the internal state undergoes the finalization phase, which outputs the authentication tag.

Every part of this process relies on iterating the `StateUpdate` function at the core of MORUS. Each call to the `StateUpdate` function is called a step. The internal state at step t is denoted by $S_0^t \| S_1^t \| S_2^t \| S_3^t \| S_4^t$, where $t = -16$ before the initialization and $t = 0$ after the initialization.

THE `StateUpdate` FUNCTION. `StateUpdate` takes as input the internal state $S^t = S_0^t \| S_1^t \| S_2^t \| S_3^t \| S_4^t$ and an additional q-bit value m^t (recall that q is the size of a register), and outputs an updated internal state.

`StateUpdate` is composed of 5 rounds with similar operations. The additional input m^t is used in rounds 2 to 5, but not in round 1. Each round uses the bit-wise rotation (left circular shift) operation inside word, denoted $\lll_w$ in the following and `Rotl_xxx_yy` in the design document. It divides a q-bit register value into 4 words of $w = q/4$ bits, and performs a rotation on each w-bit word. The bit-wise rotation constants b_i for round i are defined in Table 6.2. Additionally, each round uses rotations on a whole q-bit register by a multiple of the word size, denoted $\lll$ in the following and `<<<` in the design document. The word-wise rotation constants b'_i are also listed in Table 6.2.

Table 6.2: Rotation constants b_i for $\lll_w$ and b'_i for $\lll$ in round i of MORUS.

	Bit-wise rotation $\lll_w$					Word-wise rotation $\lll$				
	b_0	b_1	b_2	b_3	b_4	b'_0	b'_1	b'_2	b'_3	b'_4
MORUS-640	5	31	7	22	13	32	64	96	64	32
MORUS-1280	13	46	38	7	4	64	128	192	128	64

$S^{t+1} \leftarrow \texttt{StateUpdate}(S^t, m^t)$ is defined as follows, where $\cdot$ denotes bit-wise AND, $\oplus$ is bit-wise XOR, and m_i is defined depending on the context:

Here we prefer the use of $\cdot$ instead of $\wedge$ to strengthen the link with the bitwise multiplication.

$$\begin{aligned}
&\text{Round 1:} && S_0^{t+1} \leftarrow (S_0^t \oplus (S_1^t \cdot S_2^t) \oplus S_3^t) \lll_w b_0, && S_3^t \leftarrow S_3^t \lll b'_0.\\
&\text{Round 2:} && S_1^{t+1} \leftarrow (S_1^t \oplus (S_2^t \cdot S_3^t) \oplus S_4^t \oplus m_i) \lll_w b_1, && S_4^t \leftarrow S_4^t \lll b'_1.\\
&\text{Round 3:} && S_2^{t+1} \leftarrow (S_2^t \oplus (S_3^t \cdot S_4^t) \oplus S_0^t \oplus m_i) \lll_w b_2, && S_0^t \leftarrow S_0^t \lll b'_2.\\
&\text{Round 4:} && S_3^{t+1} \leftarrow (S_3^t \oplus (S_4^t \cdot S_0^t) \oplus S_1^t \oplus m_i) \lll_w b_3, && S_1^t \leftarrow S_1^t \lll b'_3.\\
&\text{Round 5:} && S_4^{t+1} \leftarrow (S_4^t \oplus (S_0^t \cdot S_1^t) \oplus S_2^t \oplus m_i) \lll_w b_4, && S_2^t \leftarrow S_2^t \lll b'_4.
\end{aligned}$$

INITIALIZATION. The initialization of MORUS-640 starts by loading the 128-bit key K_{128} and the 128-bit nonce N_{128} into the state together with constants c_0, c_1:

$$\begin{aligned}
S_0^{-16} &= N_{128},\\
S_1^{-16} &= K_{128},\\
S_2^{-16} &= \texttt{`1}^{128}\texttt{'},\\
S_3^{-16} &= c_0,\\
S_4^{-16} &= c_1.
\end{aligned}$$

Then, $\texttt{StateUpdate}(S^t, 0)$ is iterated 16 times for $t = -16, -15, \ldots, -1$. Finally, the key is XORed into the state again with $S_1^0 \leftarrow S_1^0 \oplus K_{128}$.

The initialization of MORUS-1280 differs slightly due to the difference in register size and the two possible key sizes, and uses either $K = K_{128} \| K_{128}$ (for MORUS-1280-128) or $K = K_{256}$ (for MORUS-1280-256) to initialize the state:

$$\begin{aligned}
S_0^{-16} &= N_{128} \parallel \texttt{`0}^{128}\texttt{'},\\
S_1^{-16} &= K,\\
S_2^{-16} &= \texttt{`1}^{256}\texttt{'},\\
S_3^{-16} &= \texttt{`0}^{256}\texttt{'},\\
S_4^{-16} &= c_0 \parallel c_1.
\end{aligned}$$

After iterating `StateUpdate` 16 times, the state is updated with $S_1^0 \leftarrow S_1^0 \oplus K$.

ASSOCIATED DATA PROCESSING. After initialization, the associated data A is processed in blocks of $q \in \{128, 256\}$ bits. For the padding, if the last associated data block is not a full block, it is padded to q bits with zeroes. If the length of A, denoted by $|A|$, is 0, then the associated data processing phase is skipped; else, the state is updated as

$$S^{t+1} \leftarrow \texttt{StateUpdate}(S^t, A^t) \qquad \text{for } t = 0, 1, \ldots, \lceil |A|/q \rceil - 1.$$

ENCRYPTION. Next, the message is processed in blocks M_t of $q \in \{128, 256\}$ bits to update the state and produce the ciphertext blocks C_t. If the last message block is not a full block, a string of 0's is used to pad it to 128 or 256 bits for MORUS-640 and MORUS-1280, respectively, and the padded full block is used to update the state. However, only the partial block is encrypted. Note that if the message length denoted by $|M|$ is 0, encryption is skipped. Let $u = \lceil |A|/q \rceil$ and $v = \lceil |M|/q \rceil$. The following is performed for $t = 0, 1, \ldots, v - 1$:

$$\begin{aligned} C^t &\leftarrow M^t \oplus S_0^{u+t} \oplus (S_1^{u+t} \lll b_2') \oplus (S_2^{u+t} \cdot S_3^{u+t}), \\ S^{u+t+1} &\leftarrow \texttt{StateUpdate}(S^{u+t}, M^t). \end{aligned}$$

FINALIZATION. The finalization phase generates the authentication tag T using 10 more `StateUpdate` steps. We only discuss the case where T is not truncated. The associated data length and the message length are used to update the state:

1. $L \leftarrow |A| \parallel |M|$ for MORUS-640 or
   $L \leftarrow |A| \parallel |M| \parallel 0^{128}$ for MORUS-1280,
   where $|A|, |M|$ are represented as 64-bit integers.
2. $S_4^{u+v} \leftarrow S_4^{u+v} \oplus S_0^{u+v}$.
3. For $t = u + v, u + v + 1, \ldots, u + v + 9$,
   compute $S^{t+1} \leftarrow \texttt{StateUpdate}(S^t, L)$.
4. $T = S_0^{u+v+10} \oplus (S_1^{u+v+10} \lll b_2') \oplus (S_2^{u+v+10} \cdot S_3^{u+v+10})$,
   or the least significant 128 bits of this value in case of MORUS-1280.

6.2.2 *Notation*

In the following, we use linear approximations [Mat93] that hold with probability $\Pr(E) = \frac{1}{2} + \varepsilon$, i. e., they are biased with bias ε. The *correlation* $\hat{=}(E)$ of the approximation and its *weight* $\text{weight}(E)$ [Dae95] are defined as

$$\begin{aligned} \hat{=}(E) &:= 2\Pr(E) - 1 = 2\varepsilon\,, \\ \text{weight}(E) &:= -\log_2 |\hat{=}(E)|\,, \end{aligned}$$

where $\log_2()$ denotes logarithm in base 2. By the Piling-Up Lemma, the correlation (resp. weight) of an XOR of independent variables is

equal to the product (resp. sum) of their individual correlations (resp. weights) [Mat93; Dae95].

We also recall the following notation from the previous section, where an *encryption step* refers to one call to the `StateUpdate` function:

C^t : the ciphertext block output during the t-th encryption step.

C_j^t : the j-th bit of C^t, with C_0^t being the rightmost bit.

S_i^t : the i-th register at the beginning of t-th encryption step.

$S_{i,j}^t$: the j-th bit of S_i^t, with $S_{i,0}^t$ being the rightmost bit.

In the above notation, bit positions are always taken modulo the register size q, i. e., $q = 128$ for MORUS-640 and $q = 256$ for MORUS-1280.

For simplicity, in the remainder, the 0-th encryption step will often denote the encryption step where our linear trail starts. Any encryption step could be chosen for that purpose, as long as at least four more encryption steps follow. In particular the 0-th encryption step from the perspective of the trail does not have to be the first encryption step after initialization.

6.3 ROTATIONAL INVARIANCE AND MINIMORUS

To simplify the description of the attack, we assume all plaintext blocks are zero. This assumption will be removed in Section 6.5.3, where we will show that plaintext bits only contribute linearly to the trail. Recall that the inner state of the cipher consists of five $4w$-bit registers $S_0, \ldots, S_4$, each containing four w-bit words.

6.3.1 *Rotationally Invariant Linear Combinations*

We begin with a few observations about the `StateUpdate` function. Besides XOR and AND operations, the `StateUpdate` function uses two types of bit rotations:

1. *bit-wise* rotations perform a circular shift on each word within a register;
2. *word-wise* rotations perform a circular shift on a whole register.

The second type of rotation always shifts registers by a multiple of the word size w. This amounts to a (circular) permutation of the words within the register: for example, if a register contains the words (A, B, C, D), and a word-wise rotation by w bits to the left is performed, then the register now contains the words (B, C, D, A).

To build our linear trail, we start with a linear combination of bits within a single register.

DEFINITION 6.3.1 (ROTATIONAL INVARIANCE). *Recall that w denotes the word size in bits, and $4w$ is the size of a register. A linear combination of the form:*

$$S^t_{i,j(0)} \oplus S^t_{i,j(1)} \oplus \cdots \oplus S^t_{i,j(k)}$$

is said to be rotationally invariant *iff the set of bits $S^t_{i,j(0)}, \ldots, S^t_{i,j(k)}$ is left invariant by a circular shift by w bits; that is, iff:*

$$\{j(i) : i \leq k\} = \{j(i) + w \bmod 4w : i \leq k\}.$$

Example. The following linear combination is rotationally invariant for MORUS-640, i. e., $w = 32$:

$$S^t_{0,0} \oplus S^t_{0,32} \oplus S^t_{0,64} \oplus S^t_{0,96}. \tag{6.1}$$

This definition naturally extends to a linear combination across multiple registers, and also across ciphertext blocks. The value of such a linear combination is unaffected by word-wise rotations, since those rotations always shift registers by a multiple of the word size. On the other hand, since bit-wise rotations always shift all four words within a register by the same amount, bit-wise rotations preserve the rotational invariance property. Moreover, the XOR of two rotationally invariant linear combinations is also rotationally invariant ; and the same holds for the AND operation (if we extend the symmetric property to non-linear combinations in the natural way).

This naturally leads to the idea of building a linear trail using only rotationally invariant linear combinations, which is what we are going to do. As a result, the effect of word-wise rotations can be ignored. Moreover, since all linear combinations we consider are going to be rotationally invariant, they can be described by truncating the linear combination to the first word of a register. Indeed, an equivalent way of saying a linear combination is rotationally invariant, is that it involves the same bits in each word within a register. For example, in the case of Eq. (6.1) above, the four bits involved are the first bit of each of the four words.

6.3.2 MINIMORUS

In fact, we can go further and consider a reduced version of MORUS where each register contains a single word instead of four. The `StateUpdate` function is unchanged, except for the fact that word-wise rotations are removed: see Figure 6.1. We call these reduced versions MINIMORUS-640 and MINIMORUS-1280, for MORUS-640 and MORUS-1280 respectively. Since registers in MINIMORUS contain a single word, bit-wise and word-wise rotations are the same operation; for simplicity we write $\lll$ for bit-wise rotations (the word-wise one being the identity).

Since the trail we are building is relatively complex, we will first describe it on MINIMORUS. We will then extend it to the full MORUS via the previous rotational invariance property.

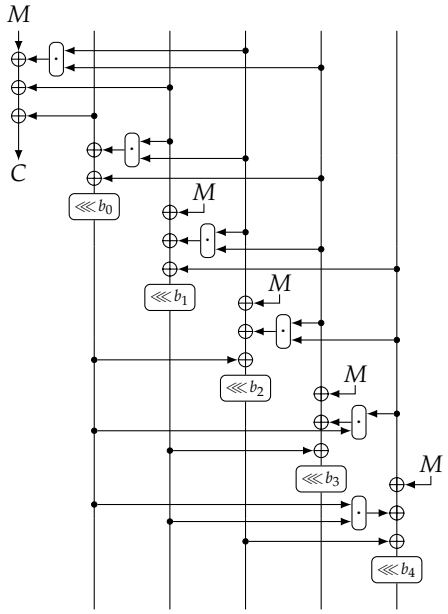


Figure 6.1: MINIMORUS state update function.

6.4 LINEAR TRAIL FOR MINIMORUS

In this section, we describe how we build a trail for MINIMORUS, then compute its correlation and validate the correlation experimentally.

6.4.1 *Overview of the Trail*

To build a linear trail for MINIMORUS, we combine the following five trail fragments α_i^t, β_i^t, γ_i^t, δ_i^t, ε_i^t, where the subscript i denotes a bit position, and the superscript t denotes a step number:

- α_i^t approximates (one bit of) state word S_0 using the ciphertext;
- β_i^t approximates S_1 using S_0 and the ciphertext;
- γ_i^t approximates S_4 using two approximations of S_1 in consecutive steps;
- δ_i^t approximates S_2 using two approximations of S_4 in consecutive steps;
- ε_i^t approximates S_0 using two approximations of S_2 in consecutive steps.

The trail fragments are depicted in Figure 6.2. In all cases except α_i^t, the trail fragment approximates a single AND gate by zero, which holds with probability $3/4$, and hence the trail fragment has weight 1. In the case of α_i^t, two AND gates are involved; however the two gates share an entry in common, and in both cases the other entry also has a linear contribution to the trail, which results in an overall contribution of the form (see [AR16, Sec. 3.3])

$$x \cdot y \oplus x \cdot z \oplus y \oplus z = (x \oplus 1) \cdot (y \oplus z).$$

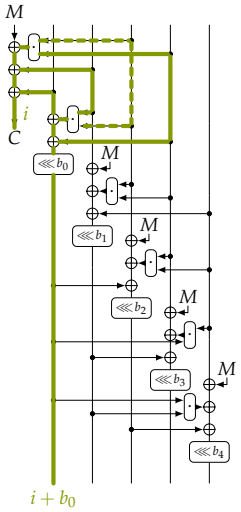
As a result, the trail fragment α_i^t also has a weight of 1. Another way of looking at this phenomenon is that the trail holds for two different approximations of the AND gates: the alternative approximation is depicted by a dashed line on Figure 6.2.

The way we are going to use each trail fragment may be summarized as follows, where in each case, elements to the left of the arrow $\rightarrow$ are used to approximate the element on the right of the arrow:

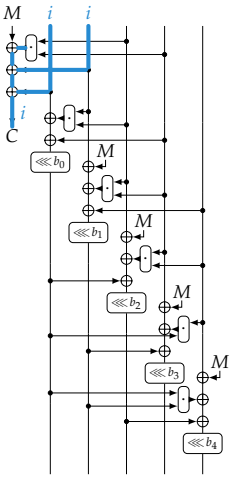
$$\begin{aligned}
\alpha_i^t &: \quad C_i^t \rightarrow S_{0,i+b_0}^{t+1} \\
\beta_i^t &: \quad C_i^t, S_{0,i}^t \rightarrow S_{1,i}^t \\
\gamma_i^t &: \quad S_{1,i}^t, S_{1,i+b_1}^{t+1} \rightarrow S_{4,i}^t \\
\delta_i^t &: \quad S_{4,i}^t, S_{4,i+b_4}^{t+1} \rightarrow S_{2,i}^{t+1} \\
\varepsilon_i^t &: \quad S_{2,i}^t, S_{2,i+b_2}^{t+1} \rightarrow S_{0,i}^{t+1}.
\end{aligned}$$

In more detail, the idea is that by using α_i^t, we are able to approximate a bit of S_0 using only a ciphertext bit. By combining α_i^t with $\beta_{i+b_0}^{t+1}$, we are then able to approximate a bit of S_1 (at step $t+1$) using only ciphertext bits from two consecutive steps. Likewise, γ_i^t allows us to "jump" from S_1 to S_4, i. e., by combining α_i^t with β_i^t and γ_i^t with appropriate choices of parameters t and i for each, we are able to approximate one bit of S_4 using only ciphertext bits. Notice however that γ_i^t requires approximating S_1 in two consecutive steps; and so the previous combination requires using α_i^t and β_i^t *twice* at different steps. In the same way, δ_i^t allows us to jump from S_4 to S_2; and ε_i^t allows jumping from S_2 back to S_0. Eventually, we are able to approximate a bit of S_0 using only ciphertext bits via the combination of all trail fragments α_i^t, β_i^t, γ_i^t, δ_i^t, and ε_i^t.

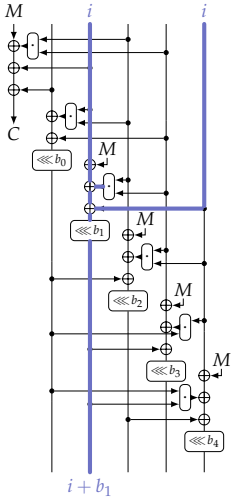
However, the same bit of S_0 can also be approximated directly by using α_i^t at the corresponding step. Thus, that bit can be linearly approximated from two different sides: the first approximation uses a combination of all trail fragments, and involves successive approximations of all state registers (except S_3) spanning several encryption steps, as explained in the previous paragraph. The second approximation only involves using α_i^t at the final step reached by the previous trail. By XORing up these two approximations, we are left with only ciphertext bits, spanning five consecutive encryption steps.



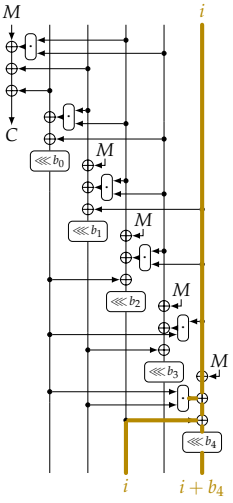
α_i^t: weight 1 (not 2)



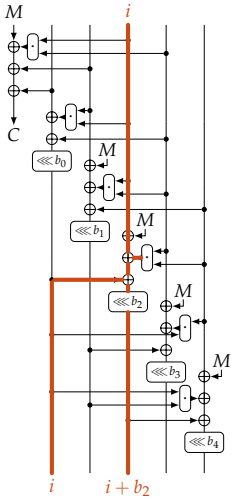
β_i^t: weight 1



γ_i^t: weight 1



δ_i^t: weight 1



ε_i^t: weight 1

$$
\begin{aligned}
\alpha_i^t &: \quad C_i^t \to S_{0,i+5}^{t+1} \\
\beta_i^t &: \quad C_i^t, S_{0,i}^t \to S_{1,i}^t \\
\gamma_i^t &: \quad S_{1,i}^t, S_{1,i+31}^{t+1} \to S_{4,i}^t \\
\delta_i^t &: \quad S_{4,i}^t, S_{4,i+13}^{t+1} \to S_{2,i}^{t+1} \\
\varepsilon_i^t &: \quad S_{2,i}^t, S_{2,i+7}^{t+1} \to S_{0,i}^{t+1}.
\end{aligned}
$$

MINIMORUS-640

$$
\begin{aligned}
\alpha_i^t &: \quad C_i^t \to S_{0,i+13}^{t+1} \\
\beta_i^t &: \quad C_i^t, S_{0,i}^t \to S_{1,i}^t \\
\gamma_i^t &: \quad S_{1,i}^t, S_{1,i+46}^{t+1} \to S_{4,i}^t \\
\delta_i^t &: \quad S_{4,i}^t, S_{4,i+4}^{t+1} \to S_{2,i}^{t+1} \\
\varepsilon_i^t &: \quad S_{2,i}^t, S_{2,i+38}^{t+1} \to S_{0,i}^{t+1}.
\end{aligned}
$$

MINIMORUS-1280

Figure 6.2: MINIMORUS linear trail fragments.

Of course, the overall trail resulting from all the previous combinations is quite complex, especially since γ_i^t, δ_i^t, and ε_i^t each require two copies of the preceding trail fragment in consecutive steps: that is, ε_i^t requires two approximations of S_2, which requires using δ_i^t twice; and δ_i^t in turn requires using γ_i^t twice, which itself requires using α_i^t and β_i^t twice. Then α_i^t is used one final time to close the trail. The full construction with the exact bit indices for MiniMorus-640 and MiniMorus-1280 is illustrated in Figure 6.3, where the left and right half each show half of the full trail. One may naturally wonder if some components of this trail are in conflict. In particular, products of bits from registers S_2 and S_3 are approximated multiple times, by α_i^t, β_i^t and γ_i^t. To address this concern, and ensure that all approximations along the trail are in fact compatible, we now compute the full trail equation explicitly.

6.4.2 *Trail Equation*

The equation corresponding to each of the five trail fragments α_i^t, β_i^t, γ_i^t, δ_i^t, ε_i^t may be written explicitly as A_i^t, B_i^t, C_i^t, D_i^t, E_i^t as follows. For each equation, we write on the left-hand side of the equality the biased linear combination used in the trail; and on the right-hand side, the remainder of the equation, which must have non-zero correlation (in all cases the correlation is 2^{-1}).

$$
\begin{aligned}
\mathrm{A}_i^t &: \quad C_i^t \oplus S_{0,i+b_0}^{t+1} = S_{1,i}^t \oplus S_{3,i}^t \oplus S_{1,i}^t \cdot S_{2,i}^t \oplus S_{2,i}^t \cdot S_{3,i}^t \\
\mathrm{B}_i^t &: \quad C_i^t \oplus S_{0,i}^t \oplus S_{1,i}^t = S_{2,i}^t \cdot S_{3,i}^t \\
\mathrm{C}_i^t &: \quad S_{1,i}^t \oplus S_{1,i+b_1}^{t+1} \oplus S_{4,i}^t = S_{2,i}^t \cdot S_{3,i}^t \\
\mathrm{D}_i^t &: \quad S_{4,i}^t \oplus S_{4,i+b_4}^{t+1} \oplus S_{2,i}^{t+1} = S_{0,i}^{t+1} \cdot S_{1,i}^{t+1} \\
\mathrm{E}_i^t &: \quad S_{2,i}^t \oplus S_{2,i+b_2}^{t+1} \oplus S_{0,i}^{t+1} = S_{3,i}^t \cdot S_{4,i}^t
\end{aligned}
$$

From an algebraic point of view, building the full trail amounts to adding up copies of the previous equations for various choices of t and i, so that eventually all $S_{y,z}^x$ terms on the left-hand side cancel out. Then we are left with only ciphertext terms on the left-hand side, while the right-hand side consists of a sum of biased expressions. By measuring the correlation of the right-hand side expression, we are then able to determine the correlation of the linear combination of ciphertext bits on the left-hand side. We now set out to do so.

In order to build the equation for the full trail, we start with E_0^2:

$$S_{2,0}^2 \oplus S_{2,b_2}^3 \oplus S_{0,0}^3 = S_{3,0}^2 \cdot S_{4,0}^2.$$

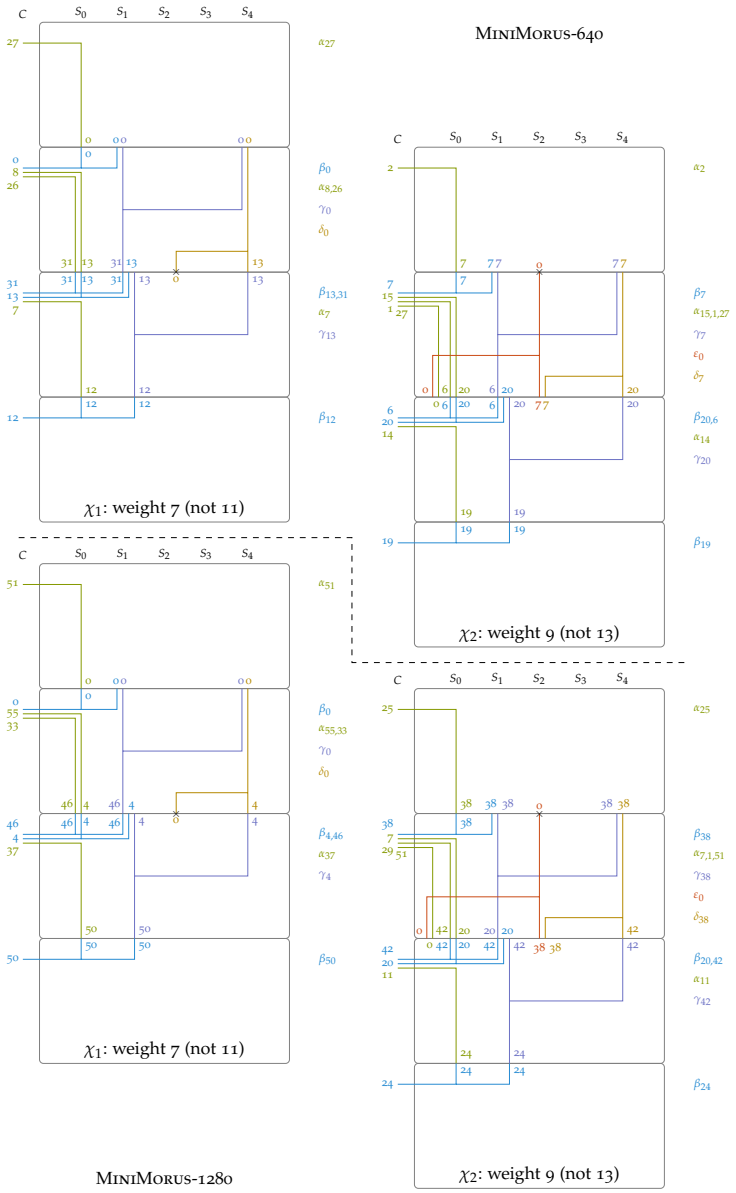


Figure 6.3: MINIMORUS: two approximations for $S^2_{2,0}$. Numbers in each diagram denote bit positions used in the linear approximation, i. e., subscripts of $\alpha, \beta, \gamma, \delta$ and ε. χ_1 and χ_2 are two halves of the full trail which we experimentally verify.

In order to cancel the $S^3_{0,0}$ term on the left-hand side, we add to the equation $\mathrm{A}^2_{-b_0}$ (where the sum of two equations of the form $a = b$ and $c = d$ is defined to be $a + c = b + d$). This yields:

$$\begin{aligned} & S^2_{2,0} \oplus S^3_{2,b_2} \oplus C^2_{-b_0} \\ = {} & S^2_{3,0} \cdot S^2_{4,0} \oplus S^2_{1,-b_0} \oplus S^2_{3,-b_0} \oplus S^2_{1,-b_0} \cdot S^2_{2,-b_0} \oplus S^2_{2,-b_0} \cdot S^2_{3,-b_0}. \end{aligned}$$

We then need to cancel two terms of the form $S^t_{2,i}$. To do this, we add to the equations D^t_i for appropriate choices of t and i. This replaces the two $S^t_{2,i}$ terms by four $S^t_{4,i}$ terms. By using equation B^t_i four times, we can then replace these four $S^t_{4,i}$ terms by eight $S^t_{1,i}$ terms. By applying equation B^t_i eight times, these eight $S^t_{1,i}$ terms can in turn be replaced by eight $S^t_{0,i}$ terms (and some ciphertext terms). Finally, applying A^t_i eight times allows to replace these eight $S^t_{0,i}$ terms by only ciphertext bits. Ultimately, for MiniMorus-1280, this yields the equation:

$$\begin{aligned} & C^0_{51} \oplus C^1_0 \oplus C^1_{25} \oplus C^1_{33} \oplus C^1_{55} \oplus C^2_4 \oplus C^2_7 \oplus C^2_{29} \oplus C^2_{37} \\ \oplus {} & C^2_{38} \oplus C^2_{46} \oplus C^2_{51} \oplus C^3_{11} \oplus C^3_{20} \oplus C^3_{42} \oplus C^3_{50} \oplus C^4_{24} \\ = {} & S^0_{1,51} \cdot S^0_{2,51} \oplus S^0_{2,51} \cdot S^0_{3,51} \oplus S^0_{1,51} \oplus S^0_{3,51} && \text{weight 1} \\ \oplus {} & S^1_{1,25} \cdot S^1_{2,25} \oplus S^1_{2,25} \cdot S^1_{3,25} \oplus S^1_{1,25} \oplus S^1_{3,25} && \text{weight 1} \\ \oplus {} & S^1_{1,33} \cdot S^1_{2,33} \oplus S^1_{2,33} \cdot S^1_{3,33} \oplus S^1_{1,33} \oplus S^1_{3,33} && \text{weight 1} \\ \oplus {} & S^1_{1,55} \cdot S^1_{2,55} \oplus S^1_{2,55} \cdot S^1_{3,55} \oplus S^1_{1,55} \oplus S^1_{3,55} && \text{weight 1} \\ \oplus {} & S^2_{1,7} \cdot S^2_{2,7} \oplus S^2_{2,7} \cdot S^2_{3,7} \oplus S^2_{1,7} \oplus S^2_{3,7} && \text{weight 1} \\ \oplus {} & S^2_{1,29} \cdot S^2_{2,29} \oplus S^2_{2,29} \cdot S^2_{3,29} \oplus S^2_{1,29} \oplus S^2_{3,29} && \text{weight 1} \\ \oplus {} & S^2_{1,37} \cdot S^2_{2,37} \oplus S^2_{2,37} \cdot S^2_{3,37} \oplus S^2_{1,37} \oplus S^2_{3,37} && \text{weight 1} \\ \oplus {} & S^2_{1,51} \cdot S^2_{2,51} \oplus S^2_{2,51} \cdot S^2_{3,51} \oplus S^2_{1,51} \oplus S^2_{3,51} && \text{weight 1} \\ \oplus {} & S^3_{1,11} \cdot S^3_{2,11} \oplus S^3_{2,11} \cdot S^3_{3,11} \oplus S^3_{1,11} \oplus S^3_{3,11} && \text{weight 1} \\ \oplus {} & S^2_{0,0} \cdot S^2_{1,0} && \text{weight 1} \\ \oplus {} & S^2_{2,46} \cdot S^2_{3,46} && \text{weight 1} \\ \oplus {} & S^2_{3,0} \cdot S^2_{4,0} && \text{weight 1} \\ \oplus {} & S^3_{0,38} \cdot S^3_{1,38} && \text{weight 1} \\ \oplus {} & S^3_{2,20} \cdot S^3_{3,20} && \text{weight 1} \\ \oplus {} & S^3_{2,50} \cdot S^3_{3,50} && \text{weight 1} \\ \oplus {} & S^4_{2,24} \cdot S^4_{3,24} && \text{weight 1} \end{aligned}$$

The equation for MiniMorus-640 is very similar, and is given in Section 6.A.

6.4.3 *Correlation of the Trail*

In the equation for MINIMORUS-1280 from the previous section, each line on the right-hand side of the equality involves distinct $S_{i,j}^t$ terms (in the sense that no two lines share a common term), and each line has a weight of 1. By the Piling-Up Lemma, it follows that if we assume distinct $S_{i,j}^t$ terms to be uniform and independent, then the expression on the right-hand side has a weight of 16. Hence the linear combination of ciphertext bits on the left-hand side has a correlation of 2^{-16}. The same holds for MINIMORUS-640.

The correlation is surprising high. The full trail uses trail fragments ε_i^t, δ_i^t, γ_i^t, β_i^t, and α_i^t, once, twice, 4 times, 8 times, and 9 times, respectively. Since each trail fragment has a weight of 1, this would suggest that the total weight should be $1 + 2 + 4 + 8 + 9 = 24$ rather than 16. However, when combining trail fragments β_i and γ_i, notice that the same AND is computed at the same step between registers S_2 and S_3 (equivalently, notice that the right-hand side of equations B_i^t and C_i^t is equal). In both cases it is approximated by zero. When XORing the corresponding equations, these two ANDs cancel each other, which saves two AND gates. Since γ_i^t is used four times in the course of the full trail, this results in saving 8 AND gates overall, which explains why the final correlation is 2^{-16} rather than 2^{-24}.

6.4.4 *Experimental Verification*

To confirm that our analysis is correct, we ran experiments on an implementation of MINIMORUS-1280 and MINIMORUS-640. We consider two halves χ_1 and χ_2 of the full trail (depicted on Figure 6.3), as well as the full trail itself, denoted by χ. In each case, we give the weight predicted by the analysis from the previous section, and the weight measured by our experiments. Results are displayed on Table 6.3. While our analysis predicts a correlation of 2^{-16}, experiments indicate a slightly better empirical correlation of $2^{-15.5}$ for MORUS-640. The discrepancy of $2^{-0.5}$ probably arises from the fact that register bits across different steps are not completely independent.

The programs we used to verify the bias experimentally are available in the associated software that comes with this thesis (Section 1.3).

6.5 TRAIL FOR FULL MORUS

In the previous section, we presented a linear trail for the reduced ciphers MINIMORUS-1280 and MINIMORUS-640. We now turn to the full ciphers MORUS-1280 and MORUS-640.

		Weight		
Approximations for MINIMORUS-640		Exp.	Bool.	Meas.
χ_1	$S_0^{2,2} = C_{27}^0 \oplus C_{0,8,26}^1 \oplus C_{7,13,31}^2 \oplus C_{12}^3$	7	7	7
χ_2	$S_0^{2,2} = C_2^1 \oplus C_{1,7,15,27}^2 \oplus C_{6,14,20}^3 \oplus C_{19}^4$	9	9	9
χ	$0 = C_{27}^0 \oplus C_{0,2,26,8}^1 \oplus C_{1,13,15,27,31}^2 \oplus C_{6,12,14,20}^3 \oplus C_{19}^4$	16	16	15.5
Approximations for MINIMORUS-1280				
χ_1	$S_0^{2,2} = C_{51}^0 \oplus C_{0,33,55}^1 \oplus C_{4,37,46}^2 \oplus C_{50}^3$	7	7	7
χ_2	$S_0^{2,2} = C_{25}^1 \oplus C_{7,29,38,51}^2 \oplus C_{11,20,42}^3 \oplus C_{24}^4$	9	9	9
χ	$0 = C_{51}^0 \oplus C_{0,25,33,55}^1 \oplus C_{4,7,29,37,38,46,51}^2 \oplus C_{11,20,42,50}^3 \oplus C_{24}^4$	16	16	15.9

Table 6.3: Experimental verification of trail correlations.

6.5.1 *Making the Trail Rotationally Invariant*

In order to build a trail for the full MORUS, we proceed exactly as we did for MINIMORUS, following the same path down to step and word rotation values, with one difference: in order to move from the one-word registers of MINIMORUS to the four-word registers of full MORUS, we make every term $S_{i,j}^t$ and C_j^t rotationally invariant, in the sense of Section 6.3. That is, for every $S_{i,j}^t$ (resp. C_j^t) component in every trail fragment and every equation, we expand the term by adding in the terms $S_{i,j+w}^t$, $S_{i,j+2w}^t$, $S_{i,j+3w}^t$ (resp. C_{j+w}^t, C_{j+2w}^t, C_{j+3w}^t), where as usual w denotes the word size. For example, if $w = 64$ (for MORUS-1280), the term $S_{2,0}^3$ is expanded into:

$$S_{2,0}^3 \oplus S_{2,64}^3 \oplus S_{2,128}^3 \oplus S_{2,192}^3.$$

Thus, translating the trail from one of the MINIMORUS ciphers to the corresponding full MORUS cipher amounts to making every linear combination rotationally invariant—indeed, that was the point of introducing MINIMORUS in the first place. Concretely, in order to build the full trail equation for MORUS, we write rotationally invariant versions of equations A_i^t, B_i^t, C_i^t, D_i^t, E_i^t from Section 6.4.2, and then combine them in exactly the same manner as before. This way, the biased linear combination on MINIMORUS-1280 given in Section 6.4.2, namely:

$$C_{51}^0 \oplus C_0^1 \oplus C_{25}^1 \oplus C_{33}^1 \oplus C_{55}^1 \oplus C_4^2 \oplus C_7^2 \oplus C_{29}^2 \oplus C_{37}^2$$

$$\oplus\, C^2_{38} \oplus C^2_{46} \oplus C^2_{51} \oplus C^3_{11} \oplus C^3_{20} \oplus C^3_{42} \oplus C^3_{50} \oplus C^4_{24}$$

ultimately yields the following biased rotationally invariant linear combination on the full MORUS-1280:

$$\begin{aligned}
&C^0_{51} \oplus C^0_{115} \oplus C^0_{179} \oplus C^0_{243} \oplus C^1_{0} \oplus C^1_{25} \oplus C^1_{33} \oplus C^1_{55} \oplus C^1_{64} \oplus C^1_{89} \\
&\oplus C^1_{97} \oplus C^1_{119} \oplus C^1_{128} \oplus C^1_{153} \oplus C^1_{161} \oplus C^1_{183} \oplus C^1_{192} \oplus C^1_{217} \oplus C^1_{225} \oplus C^1_{247} \\
&\oplus C^2_{4} \oplus C^2_{7} \oplus C^2_{29} \oplus C^2_{37} \oplus C^2_{38} \oplus C^2_{46} \oplus C^2_{51} \oplus C^2_{68} \oplus C^2_{71} \oplus C^2_{93} \\
&\oplus C^2_{101} \oplus C^2_{102} \oplus C^2_{110} \oplus C^2_{115} \oplus C^2_{132} \oplus C^2_{135} \oplus C^2_{157} \oplus C^2_{165} \oplus C^2_{166} \oplus C^2_{174} \\
&\oplus C^2_{179} \oplus C^2_{196} \oplus C^2_{199} \oplus C^2_{221} \oplus C^2_{229} \oplus C^2_{230} \oplus C^2_{238} \oplus C^2_{243} \oplus C^3_{11} \oplus C^3_{20} \\
&\oplus C^3_{42} \oplus C^3_{50} \oplus C^3_{75} \oplus C^3_{84} \oplus C^3_{106} \oplus C^3_{114} \oplus C^3_{139} \oplus C^3_{148} \oplus C^3_{170} \oplus C^3_{178} \\
&\oplus C^3_{203} \oplus C^3_{212} \oplus C^3_{234} \oplus C^3_{242} \oplus C^4_{24} \oplus C^4_{88} \oplus C^4_{152} \oplus C^4_{216}
\end{aligned}$$

We refer the reader to Section 6.C for the corresponding linear combination on MORUS-640.

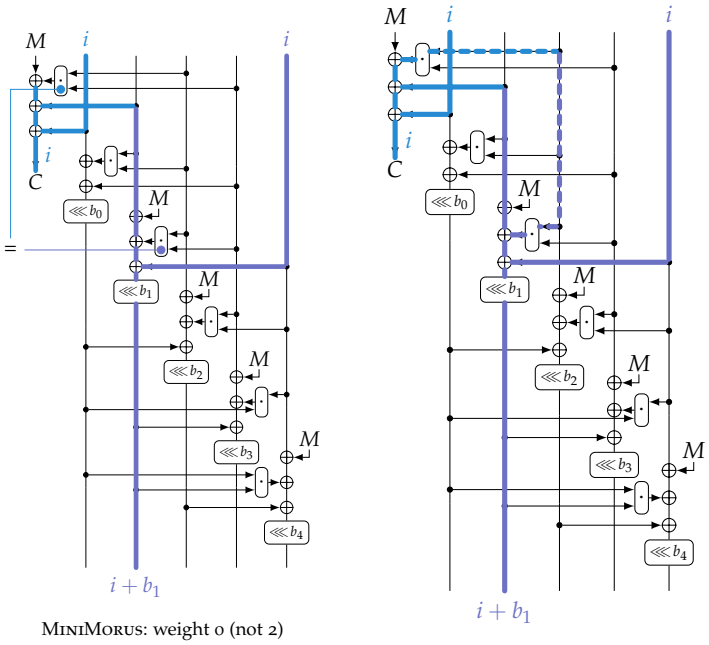
6.5.2 *Correlation of the Full Trail*

The rotationally invariant trail on full MORUS may be intuitively understood as consisting of four copies of the original trail on MINIMORUS. Indeed, the only difference between full MORUS (for either version of MORUS) and four independent copies of MINIMORUS comes from word-wise rotations, which permute words within a register. But as observed in Section 6.3, word-wise rotations preserves the rotational invariance property; and so, insofar as we only ever use rotationally invariant linear combinations on all registers along the trail, word-wise rotations have no effect.

Following the previous intuition, one may expect that the weight of the full trail should simply be four times the weight of the corresponding MINIMORUS trail, namely 64 for both MORUS-1280 and MORUS-640. However, reality is a little more complex, as the full trail does not exactly behave as four copies of the original trail when one considers nonlinear terms.

To understand why that might be the case, assume a nonlinear term $S^0_{2,0} \cdot S^0_{3,0}$ arising from some part of the trail, and another term $S^0_{2,0} \cdot S^0_{3,w}$ arising from a different part of the trail (where w denotes the word size). Then, when we XOR the various trail fragments together, in MINIMORUS these two terms are actually equal and will cancel out, since word-wise rotations by multiples of w bits are ignored. However, in the real MORUS these terms are of course distinct and do not cancel each other.

In the actual trail for (either version of) full MORUS, this exact situation occurs when combining trail fragments β^t_i and γ^t_i. Indeed, β^t_i requires approximating the term $S^t_{2,i} \cdot S^t_{3,i}$, while γ^t_i requires approximating the term $S^t_{2,i} \cdot S^t_{3,i-w}$ (cf. Figure 6.4). While in MINIMORUS,



MINIMORUS: weight 0 (not 2)

MORUS: weight 4×1 (not 4×2)

Figure 6.4: Weight of $\beta_i^t \oplus \gamma_i^t$ for MINIMORUS and MORUS.

these terms cancel out, in the full MORUS, when adding up four copies of the trail to achieve rotational invariance, we end up with the sum:

$$\begin{aligned} &S^t_{2,i} \cdot S^t_{3,i} \oplus S^t_{3,i} \cdot S^t_{2,i+w} \oplus S^t_{2,i+w} \cdot S^t_{3,i+w} \\ \oplus\ &S^t_{3,i+w} \cdot S^t_{2,i+2w} \oplus S^t_{2,i+2w} \cdot S^t_{3,i+2w} \oplus S^t_{3,i+2w} \cdot S^t_{2,i+3w} \\ \oplus\ &S^t_{2,i+3w} \cdot S^t_{3,i+3w} \oplus S^t_{3,i+3w} \cdot S^t_{2,i}. \end{aligned} \tag{6.2}$$

It may be observed that the products occurring in the equation above involve eight terms forming a ring. The weight of this expression can be computed by brute force, and is equal to 3.

For MORUS-1280, since the trail fragment γ_i^t is used four times, this phenomenon adds a contribution of $4 \cdot 3 = 12$ to the overall weight of the full trail. This results in a total weight of $4 \cdot 16 + 12 = 76$ (recall that the weight of the trail on MINIMORUS-1280 is 16). We have confirmed this by explicitly computing the full trail equation in Section 6.D, and evaluating its exact weight like we did for MINIMORUS in Section 6.4.3. That is, since the equation is quadratic, we may view it as a graph, which we split into connected components; we then compute the weight of each connected component separately by brute force, and then add up the weights of all components per the Piling-Up Lemma. Overall, the full trail equation given in Section 6.D yields a weight of 76 for the full trail on MORUS-1280.

In the case of MORUS-640, collisions between rotation constants further complicate the analysis. Specifically, when using trail fragment β_i^t, the term $S_{2,i}^t \cdot S_{3,i}^t$ occurs. As explained previously, a partial collision with the term $S_{2,i}^t \cdot S_{3,i-w}^t$ from trail fragment γ_i^t results in Equation (6.2). However, trail fragment α_{i+d}^t is once used in the course of the full trail with an offset of $d = b_1 + b_4 - b_0 - b_2$ (relative to γ_i^t), which in the case of MORUS-640 is equal to $31 + 13 - 5 - 7 = 0 \bmod 32$. This creates another term $S_{2,i}^t \cdot S_{3,i}^t$, which ultimately destroys one of the four occurrences of Equation (6.2). Therefore, when computing the full trail equation on MORUS-640, we get that the weight of the trail is 73 (cf. Section 6.C).

6.5.3 *Taking Variable Plaintext into Account*

In our analysis so far, for the sake of simplicity, we have assumed that all plaintext blocks are zero. We now examine what happens if we remove that assumption, and integrate plaintext variables into our analysis. What we show is that plaintext variables only contribute linearly to the trail. In other words, the full trail equation with plaintext variables is equal to the full trail equation with all-zero plaintext XORed with a linear combination of plaintext variables.

To see this, recall that plaintext bits contribute to the encryption process in two ways (cf. Section 6.2.1):

1. They are added to some bits derived from the state to form the ciphertext.

2. During each encryption step, the `StateUpdate` function adds a plaintext block to every register except S_0.

The effect of Item 1 is that whenever we use a ciphertext bit in our full trail equation, the corresponding plaintext bit also needs to be XORed in. Because ciphertext bits only contribute linearly to the trail equation, this only adds a linear combination of plaintext bits to the equation.

Regarding Item 2, recall that the full trail equation is a linear combination of (the rotationally invariant version of) equations A_i^t, B_i^t, C_i^t, D_i^t, E_i^t in Section 6.4.2. Also observe that in each equation, state bits that are shifted by a bit-wise rotation only contribute linearly. Because plaintext bits are XORed into each register at the same time bit-wise rotation is performed, this implies that plaintext bits resulting from Item 2 also only contribute linearly. In fact in all cases, it so happens that updating the equation to take plaintext variables into account simply involves XORing in the plaintext bit M_i^t.

It may be observed that message blocks in the `StateUpdate` function only contribute linearly to the state, and in that regard play a role similar to key bits in an SPN cipher; and indeed in SPN ciphers, it is

the case that key bits contribute linearly to linear trails [Mat93]. In this light the previous result may not be surprising.

In the end, with variable plaintext, our trail yields a biased linear combination of ciphertext bits and plaintext bits. With regard to attacks, this means the situation is effectively the same as with a biased stream cipher: in particular if the plaintext is known we obtain a distinguisher; and if a fixed unknown plaintext is encrypted multiple times (possibly also with some known variable part) then our trail yields a plaintext recovery attack.

6.6 DISCUSSION

We now discuss the impact of these attacks on the security of MORUS.

6.6.1 *Keystream Correlation*

We emphasize that the correlation we uncover between plaintext and ciphertext bits is *absolute*, in the sense that it does not depend on the encryption key, or on the nonce. This is the same situation as the keystream correlations in AEGIS [Min14]. As such, they can be leveraged to mount an attack in the broadcast setting, where the same message is encrypted multiple times with different IVs and potentially different keys [MS01]. In particular, the broadcast setting appears in practice in man-in-the-browser attacks against HTTPS connections following the BEAST model [DR11]. In this scenario, an attacker uses Javascript code running in the victim's browser (by tricking the victim to visit a malicious website) to generate a large number of request to a secure website. Because of details of the HTTP protocol, each request includes an authentication token to identify the user, and the attacker can target this token as a repeated plaintext. Concretely, correlations in the RC4 keystream have been exploited in this setting, leading to the recovery of authentication cookies in practice [AlF+13].

6.6.2 *Data Complexity*

The design document of MORUS imposes a limit of 2^{64} encrypted blocks for a given key. However, since our attack is independent of the encryption key, and hence immune to rekeying, this limitation does not apply: all that matters for our attack is that the same plaintext be encrypted enough times.

With the trail presented in this work, the data complexity is clearly out of reach in practice, since exploiting the correlation would require 2^{152} encrypted blocks for MORUS1280, and 2^{146} encrypted blocks for MORUS640. The data complexity could be slightly lowered by leveraging multilinear cryptanalysis; indeed, the trail holds for any bit shift, and if we assume independence, we could run w copies of

the trail in parallel on the same encrypted blocks (recall that w is the word size, and the trail is invariant by rotation by w bits). This would save a factor 2^5 on the data complexity for MORUS640, and 2^6 for MORUS1280; but the resulting complexity is still out of reach.

However, MORUS1280 with a 256-bit key claims a security level of 256 bits for confidentiality, and an attack with complexity 2^{152} violates this claim, even if it is not practical.

6.6.3 *Design Considerations*

The existence of this trail does hint at some weakness in the design of MORUS. Indeed, a notable feature of the trail is that the values of rotation constants are mostly irrelevant: a similar trail would exist for most choices of the constants. That it is possible to build a trail that ignores rotation constants may be surprising. This would have been prevented by adding a bit-wise rotation to one of the state registers at the input of the ciphertext equation.

6.7 ANALYSIS ON INITIALIZATION AND FINALIZATION OF REDUCED MORUS

The bias in the previous sections analyzed the encryption part of the MORUS. In this section, for comprehensive security analysis of MORUS, we provide new attacks on reduced version of the initialization and the finalization. We emphasize that the results in this section do not threaten any security claim by the designers. However, we believe that investigating all parts of the design with different approaches from the existing work on MORUS provides a better understanding and will be useful especially when the design will be tweaked in the future.

6.7.1 *Forgery with Reduced Finalization*

We present forgery attacks on 3 out of 10 steps of MORUS-1280 that claims 128-bit security for integrity. The attack only works for a limited number of steps, while it works in the nonce-respecting setting. As far as we know, this is the first attempt to evaluate integrity of MORUS in the nonce-respecting setting.

OVERVIEW. A general strategy for forgery attacks in the nonce-respecting setting is to inject some difference in a message block and propagate it so that it can be canceled by a difference in another message block. However, this approach does not work well against MORUS due to its large state size which prevents an attacker from easily controlling the differences in different registers.

Here we focus on the property that the padding for an associated data A and a message M is the zero-padding, hence A and $A' = A \| \texttt{`0*'}$

and M and $M' = M \| \text{`0'}$ result in identical states after the associated data processing and the encryption parts, as long as A, A' and M, M' fit in the same number of blocks. During the finalization, since A, A' (resp. M, M') have different lengths, the corresponding 64-bit values $|A|$ (resp. $|M|$) are different, which appears as $\Delta|A|$ (resp. $\Delta|M|$) during the finalization, and is injected through the message input interface. Our strategy is to propagate this difference to the 128-bit tags T and T' such that their difference ΔT appears with higher probability than 2^{-128}. All in all, the forgery succeeds as long as the desired ΔT is obtained or in other words, the attacker does not have to cancel the state difference, which is the main advantage of attacking the finalization part of the scheme.

Note that if the attacker uses different messages M, M', not only the new tag T' but also new ciphertext C' must be guessed correctly. Because the encryption of MORUS is a simple XOR of the key stream, C' can be easily guessed. For this purpose, the attacker should first query a longer message $M' = M \| \text{`0}^{*}\text{'}$ to obtain C'. Then, C can be obtained by truncating C'.

DIFFERENTIAL TRAILS. Recall that the message input during the finalization of MORUS-1280 is $|A| \parallel |M| \parallel \text{`0}^{128}\text{'}$ where $|A|$ and $|M|$ are 64-bit strings. We set $\Delta|A|$ to be of low Hamming weight, e. g., `0x0000000000000001`. This difference propagates through 3 steps as specified in Table 6.4.

Recall that each step consists of 5 rounds and the input message is absorbed to the state in rounds 2 to 5. The trail in Table 6.4 initially does not have any difference and the same continues even after round 1. Differences start to appear from round 2 and they will go through the bitwise-AND operation from round 4. We need to pay 1 bit to control each active AND gate. The probability evaluation for round 15 can be ignored since in this round only S_4 is non-linearly updated, while S_4 is never used for computing the tag. Finally, bitwise-AND in the tag computation is taken into account. Note that the tag is only 128 LSBs, thus the number of active AND gates should be counted only for those bits. As shown in Table 6.4, we can have a particular tag difference ΔT with probability 2^{-88}. Thus after observing A and corresponding T, $A\|0$ and $(T \oplus \Delta T)$ is a valid pair with probability 2^{-88}.

Round	State difference				w	Acc. prob.
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
Ini	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	–
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	

	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
1	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	1
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000400000000000	0000000000000000	0000000000000000	0000000000000000	1	
2	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	1
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000400000000000	0000000000000000	0000000000000000	0000000000000000	1	
3	0000004000000000	0000000000000000	0000000000000000	0000000000000000	1	1
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000400000000000	0000000000000000	1	
4	0000004000000000	0000000000000000	0000000000000000	0000000000000000	1	1
	0020000000000080	0000000000000000	0000000000000000	0000000000000000	2	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0	
	0000000000000000	0000000000000000	0000400000000000	0000000000000000	1	
5	0000000000000000	0000000000000000	0000000000000000	0000004000000000	1	2^{-1}
	0020000000000080	0000000000000000	0000000000000000	0000000000000000	2	
	0000040000000010	0000000000000000	0000000000000000	0000000000000000	2	
	0000000000100004	0000000000000000	0000000000000000	0000000000000000	2	
	0000000000000000	0000000000000000	0000400000000000	0000000000000000	1	
6	0000000000000000	0000000000000000	0000000000000000	0000004000000000	1	2^{-3}
	0000000000000000	0000000000000000	0000000000000000	0020000000000080	2	
	0000040000000010	0000000000000000	0000000000000000	0000000000000000	2	
	0000000000100004	0000000000000000	0000000000000000	0000000000000000	2	
	0004400001000000	0000000000000000	0000000010000000	0000000000000000	4	
7	0000000000000000	0000000000000000	0000000000000000	0000004000000000	1	2^{-6}
	0000000000000000	0000000000000000	0000000000000000	0020000000000080	2	
	0000000000000000	0000000000000000	0000040000000010	0000000000000000	2	
	0000000000000000	0000000000100004	0000000000000000	0000000000000000	2	
	0004400001000000	0000000000000000	0000000010000000	0000000000000000	4	
8	0400014000000000	0000000000000000	0000000000000000	0000000000001000	4	2^{-10}
	0000000000000000	0000000000000000	0000000000000000	0020000000000080	2	
	0000000000000000	0000000000000000	0000040000000010	0000000000000000	2	
	0000000000000000	0000000000100004	0000000000000000	0000000000000000	2	
	0000000010000000	0000000000000000	0004400001000000	0000000000000000	4	
9	0400014000000000	0000000000000000	0000000000000000	0000000000001000	4	2^{-14}
	0220000080000080	0000000000000000	0000000800000000	1000000000004000	7	
	0000000000000000	0000000000000000	0000040000000010	0000000000000000	2	
	0000000000000000	0000000000100004	0000000000000000	0000000000000000	2	
	0000000010000000	0000000000000000	0004400001000000	0000000000000000	4	
10	0000000000000000	0000000000000000	0000000000001000	0400014000000000	4	2^{-20}
	0220000080000080	0000000000000000	0000000800000000	1000000000004000	7	

	4000140000000010	0000000000000000	0000400000000100	0000000000010000	7	
	0000100000100044	0000000200008000	0001000000000000	0000000008000200	9	
	0000000010000000	0000000000000000	0004400001000000	0000000000000000	4	
11	0000000000000000	0000000000000000	0000000000001000	0400014000000000	4	2^{-28}
	0000000000000000	0000000800000000	1000000000004000	0220000080000080	7	
	4000140000000010	0000000000000000	0000400000000100	0000000000010000	7	
	0000100000100044	0000000200008000	0001000000000000	0000000008000200	9	
	0004500005000400	0000000000000000	0040000100000040	4000000000000000	10	
12	0000000000000000	0000000000000000	0000000000001000	0400014000000000	4	2^{-39}
	0000000000000000	0000000800000000	1000000000004000	0220000080000080	7	
	0000400000000100	0000000000010000	4000140000000010	0000000000000000	7	
	0000000008000200	0000100000100044	0000000200008000	0001000000000000	9	
	0004500005000400	0000000000000000	0040000100000040	4000000000000000	10	
13	0400114000040000	0020000000000080	0004000000400000	0000800100005002	14	2^{-53}
	0000000000000000	0000000800000000	1000000000004000	0220000080000080	7	
	0000400000000100	0000000000010000	4000140000000010	0000000000000000	7	
	0000000008000200	0000100000100044	0000000200008000	0001000000000000	9	
	0040000100000040	4000000000000000	0004500005000400	0000000000000000	10	
14	0400114000040000	0020000000000080	0004000000400000	0000800100005002	14	2^{-69}
	0228000280020080	0000040000000000	2000008000202008	1000004000004021	18	
	0000400000000100	0000000000010000	4000140000000010	0000000000000000	7	
	0000000008000200	0000100000100044	0000000200008000	0001000000000000	9	
	0040000100000040	4000000000000000	0004500005000400	0000000000000000	10	
15	0020000000000080	0004000000400000	0000800100005002	0400114000040000	14	–
	0228000280020080	0000040000000000	2000008000202008	1000004000004021	18	
	0000400000000100	0000000000010000	4000140000000010	0000000000000000	7	
ΔT			600080830020f00a	1405414005044421		2^{-88}

Table 6.4: Differential propagation through 3 Steps. Five lines for round i denote the difference of $S_0, \cdots, S_4$ after the round i transformation. In this table Weight is a Hamming weight of the state difference and accumulated probability is a probability to satisfy the trail from the beginning.

REMARKS. The fact that the S_4 is updated in the last round but is not used in the tag generation implies that the MORUS finalization generally includes unnecessary computations with respect to security. It may be interesting to tweak the design such that the tag can also depend on S_4. Indeed, in Table 6.4, we can observe some jump-up of the probability in the tag computation. This is because the non-linearly involved terms are $S_2 \cdot S_3$, and S_3 that was updated 2 rounds before has a high Hamming weight. In this sense, involving S_4 in non-linear terms of the tag computation imposes more difficulties for the attacker.

6.7.2 *Extending State Recovery to Key Recovery*

Kales et al. [KEM17] showed that the internal state of MORUS-640 can be recovered under the nonce-misuse scenario using 2^5 plaintext-ciphertext pairs. As claimed by [KEM17] the attack is naturally extended to MORUS-1280 though Kales et al. [KEM17] did not demonstrate specific attacks. The recovered state allows the attacker to mount a universal forgery attack under the same nonce. However, the key still cannot be recovered because the key is used both at the beginning and end of the initialization, which prevents the attacker from backtracking the state value to the initial state. In this section, we show that meet-in-the-middle attacks allow the attacker to recover the key faster than exhaustive search for a relatively large number of steps, i. e., 10 out of 16 steps in MORUS-1280.

OVERVIEW. We divide the 10 steps of the initialization computation into two subsequent parts F_0 and F_1. (We later set that F_0 is the first 4 steps and F_1 is the last 6 steps.) Let S^{-10} be the initial state value before setting the key, i. e., $S^{-10} = (N \| \texttt{'0}^{128}\texttt{'}, \texttt{'0}^{256}\texttt{'}, \texttt{'1}^{256}\texttt{'}, \texttt{'0}^{256}\texttt{'}, c_0 \| c_1)$. Also let S^0 be 1280-bit state value after the initialization, which is now assumed to be recovered with the nonce-misuse analysis [KEM17]. We then have the following relation.

$$F_1 \circ F_0\big(S^{-10} \oplus (0, K, 0, 0, 0)\big) \oplus (0, K, 0, 0, 0) = S^0.$$

We target the variant MORUS-1280-128, where $K = K_{128} \| K_{128}$.

Here, our strategy is to recover K_{128} by independently processing F_0 and F_1^{-1} to find the following match.

$$F_0(S^{-10} \oplus (0, K_{128} \| K_{128}, 0, 0, 0)) \stackrel{?}{=} F_1^{-1}(S^0 \oplus (0, K_{128} \| K_{128}, 0, 0, 0)).$$

To evaluate the attack complexity, we consider the following parameters.

- G_0: a set of bits of K_{128} that are guessed for computing F_0.
- G_1: a set of bits of K_{128} that are guessed for computing F_1^{-1}.
- G_2: a set of bits in the intersection of G_0 and G_1.
- x bits can match after processing F_0 and F_1^{-1}.

Suppose that the union of G_0 and G_1 covers all the bits of K_{128}. The attack exhaustively guesses G_2 and performs the following procedure for each guess.

1. F_0 is computed $2^{|G_0|-|G_2|}$ times and the results are stored in a table T. (Because $|G_1| - |G_2|$ bits are unknown, only a part of the state is computed.)

2. F_1^{-1} is computed $2^{|G_1|-|G_2|}$ times and for each result we check the match with any entry in T.

3. The number of possible combinations is $2^{|G_0|-|G_2|+|G_1|-|G_2|}$, and the number of valid matches reduces to $2^{|G_0|-|G_2|+|G_1|-|G_2|-x}$ after matching the x bits.

4. Check the correctness of the guess by using one plaintext-ciphertext pair.

In the end, F_0 is computed $2^{|G_2|} \cdot 2^{|G_0|-|G_2|} = 2^{|G_0|}$ times. Similarly, F_1^{-1} is computed $2^{|G_1|}$ times. The number of the total candidates after the x-bit match is $2^{|G_2|} \cdot 2^{|G_0|-|G_2|+|G_1|-|G_2|-x} = 2^{|G_0|+|G_1|-|G_2|-x}$. Hence, the key K_{128} is recovered with complexity

$$\max(2^{|G_0|}, 2^{|G_1|}, 2^{|G_0|+|G_1|-|G_2|-x}).$$

Suppose that we choose $|G_0|$ and $|G_1|$ to be balanced i. e., $|G_0| = |G_1|$. Then, the complexity is

$$\max(2^{|G_0|}, 2^{2|G_0|-|G_2|-x}).$$

Two terms are balanced when $x = |G_0| - |G_2|$. Hence, the number of matched bits in the middle of two functions must be greater than or equal to the number of independently guessed bits to compute F_0 and F_1^{-1}.

In the attack below, we choose $|G_0| = |G_1| = 127$ and $|G_2| = 126$ (equivalently $|G_2| - |G_0| = |G_2| - |G_1| = 1$) in order to aim $x = 1$-bit match in the middle, which maximizes the number of attacked rounds.

Round	State Difference			
$S^{-10} \oplus K_{128}$	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000001	0000000000000000	0000000000000001
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
1	0000000000002000	0000000000000000	0000000000002000	0000000000000000
	0000000000000000	0000400000000000	0000000000000000	0000400000000000
	0008000000000000	0000000000000000	0008000000000000	0000000000000000
	0000000000100000	0020000000000000	0000000000100000	0020000000000000
	0000000000020000	0084000000000000	0000000000020000	0084000000000000
2	0800000000000004	0000000204000001	0800000000000004	0000000204000001
	8000000a00000000	0000002110000004	8000000a00000000	0000002110000004
	0400010221000000	008000400a000081	0400010221000000	008000400a000081
	1000050001000244	4200118a08000280	1000050001000244	4200118a08000280
	880004a0a0200858	4840123350000050	880004a0a0200858	4840123350000050
3	023d63c00050a850	00a1442000489380	023d63c00050a850	00a1442000489380
	02b63380056aaa48	00b5563005dcd6c0	02b63380056aaa48	00b5563005dcd6c0
	d42ab556bf5dfcd6	5a26f633a8556aaa	d42ab556bf5dfcd6	5a26f633a8556aaa
	5fbbf556bd556c65	7aab99aaee6bea2c	5fbbf556bd556c65	7aab99aaee6bea2c
	abff7f3ad7feafad	cfff777ffddffd6d	abff7f3ad7feafad	cfff777ffddffd6d
4	fff77dfffffdcf57	fefad7efffdffbf7	fff77dfffffdcf57	fefad7efffdffbf7
	ffffffffffffbfff	fffbf7fffddfff77	ffffffffffffbfff	fffbf7fffddfff77
	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff
	ffffffffffffffff	fffbffffefffffff	ffffffffffffffff	fffbffffefffffff
	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff

		2-bits match		2-bits match
5	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff
	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff
	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff	ffffffffffffffff
	fffff7ffedffffff	ffffffffedffffff	fffff7ffedffffff	ffffffffedffffff
	ffffffffedffffff	fffff7ffedffffff	ffffffffedffffff	fffff7ffedffffff
6	ffffffffedffffff	fffff7ffedffffff	ffffffffedffffff	fffff7ffedffffff
	fffbf5e7cdfffbf7	fffff7bfcdfff757	fffbf5e7cdfffbf7	fffff7bfcdfff757
	fffbf5e7cdfffbf7	fffff7bfcdfff757	fffbf5e7cdfffbf7	fffff7bfcdfff757
	7ffd75b6cdfff357	fffbf5a6ccfcfb73	7ffd75b6cdfff357	fffbf5a6ccfcfb73
	7ffbf5a6ccfcf373	7ff975b6ccfff353	7ffbf5a6ccfcf373	7ff975b6ccfff353
7	7efbf5a6cc7cf353	7fd975a6cceff353	7efbf5a6cc7cf353	7fd975a6cceff353
	7eb950a4cc78e353	7dd07184cced7153	7eb950a4cc78e353	7dd07184cced7153
	7eb950a4cc78e353	7dd07184ccec7153	7eb950a4cc78e353	7dd07184ccec7153
	7cd051044c6c3153	3e985024cc48a313	7cd051044c6c3153	3e985024cc48a313
	3c905004cc482313	7c9051044c6c2113	3c905004cc482313	7c9051044c6c2113
8	2c905004c4482113	7c9050040c682113	2c905004c4482113	7c9050040c682113
	2810100444082112	5c1010040c402113	2810100444082112	5c1010040c402113
	2810100444082112	1c1010040c402113	2810100444082112	1c1010040c402113
	0c00100404400113	2800000404082112	0c00100404400113	2800000404082112
	0800000404002112	0800100404400113	0800000404002112	0800100404400113
9	0800000404002112	0800100004000112	0800000404002112	0800100004000112
	0000000404000102	0000100004000110	0000000404000102	0000100004000110
	0000000404000102	0000000004000110	0000000404000102	0000000004000110
	0000000004000110	0000000000000102	0000000004000110	0000000000000102
	0000000000000100	0000000004000110	0000000000000100	0000000004000110
10	0000000000000100	0000000004000100	0000000000000100	0000000004000100
	0000000000000000	0000000004000100	0000000000000000	0000000004000100
	0000000000000000	0000000000000100	0000000000000000	0000000000000100
	0000000000000100	0000000000000000	0000000000000100	0000000000000000
	0000000000000000	0000000000000100	0000000000000000	0000000000000100
$S^0 \oplus K_{128}$	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000100	0000000000000000	0000000000000100
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000

Table 6.5: Analysis of the diffusion and matching bits over 10 steps. '0' and '1' denote that the state bit can and cannot be computed from a partial knowledge of K_{128}, respectively. After the partial computations from each direction, 4 bits of S^{-6} can match.

FULL DIFFUSION ROUNDS. We found that `StepUpdate` was designed to have good diffusion in the forward direction. Thus, once the state is recovered, the attacker can perform the partial computation in the backward direction longer than the forward direction. We set G_0 and G_1 as follows.

$$G_0 = \{1, 2, \cdots, 127\} \qquad \text{Bit position 0 is unknown.}$$
$$G_1 = \{0, 1, \cdots, 7, 9, 10, \cdots, 127\} \qquad \text{Bit position 8 is unknown.}$$

Those will lead to 4 matching bits after the 4-step forward computation and the 6-step backward computation. The analysis of the diffusion is given in Table 6.5. In the end, K_{128} can be recovered faster than the exhaustive search by 1 bit, i. e., with complexity 2^{127}.

REMARKS. The matching state does not have to be a border of a step. It can be defined on a border of a round, or even in some more complicated way. We did not find the extension of the number of attacked steps even with this way.

As can be seen in Table 6.5, the updated register in step i is independent of the update function in step $i+1$ in the forward direction, and starts to impact from step $i+2$. By modifying this point, the diffusion speed can increase faster, which makes this attack harder.

6.8 CONCLUSION

This work provides a comprehensive analysis of the components of MORUS. In particular, we show that MORUS-1280's keystream exhibits a correlation of 2^{-76} between certain ciphertext bits. This enables a plaintext recovery attack in the broadcast setting, using about 2^{152} blocks of data. While the amount of data required is impractical, this seems to violate the security claims of MORUS-1280 because the attack works even if the key is refreshed regularly. Moreover, the broadcast setting is practically relevant, as was shown with attacks against RC4 as used in TLS [AlF+13].

Prior its submission, the published paper has been shared with the authors of MORUS and they agreed with the technical details of the keystream bias. However, they consider that it is not a significant weakness in practice because it requires more than 2^{64} ciphertexts bits. In the context of the CAESAR competition, we believe that certificational attacks such as this one should be taken into account, in order to select a portfolio of candidates that reflects the state-of-the-art in terms of cryptographic design.

APPENDIX OF CHAPTER 6

We provide full trail equations for all variants of Morus. In each case, we decompose the right-hand side of the equality (involving state bits) into connected components, and compute the weight of each of these connected components. If we assume that distinct state bits are uniformly random and independent, then each connected component is independent. By the Piling-Up Lemma, it follows that the weight of the full equation is equal to the sum of the weights of the connected components.

6.A TRAIL EQUATION FOR MINIMORUS-640

$$
\begin{aligned}
& C^0_{27} \oplus C^1_0 \oplus C^1_2 \oplus C^1_8 \oplus C^1_{26} \oplus C^2_1 \oplus C^2_{13} \oplus C^2_{15} \\
\oplus\ & C^2_{27} \oplus C^2_{31} \oplus C^3_6 \oplus C^3_{12} \oplus C^3_{14} \oplus C^3_{20} \oplus C^4_{19} \\
=\ & S^0_{1,27} \oplus S^0_{3,27} \oplus S^0_{1,27} \cdot S^0_{2,27} \oplus S^0_{2,27} \cdot S^0_{3,27} && \text{weight 1} \\
\oplus\ & S^1_{1,26} \oplus S^1_{3,26} \oplus S^1_{1,26} \cdot S^1_{2,26} \oplus S^1_{2,26} \cdot S^1_{3,26} && \text{weight 1} \\
\oplus\ & S^1_{1,8} \oplus S^1_{3,8} \oplus S^1_{1,8} \cdot S^1_{2,8} \oplus S^1_{2,8} \cdot S^1_{3,8} && \text{weight 1} \\
\oplus\ & S^2_{3,1} \oplus S^2_{1,1} \oplus S^2_{1,1} \cdot S^2_{2,1} \oplus S^2_{2,1} \cdot S^2_{3,1} && \text{weight 1} \\
\oplus\ & S^2_{1,7} \oplus S^2_{3,7} \oplus S^2_{1,7} \cdot S^2_{2,7} \oplus S^2_{2,7} \cdot S^2_{3,7} && \text{weight 1} \\
\oplus\ & S^2_{1,15} \oplus S^2_{3,15} \oplus S^2_{1,15} \cdot S^2_{2,15} \oplus S^2_{2,15} \cdot S^2_{3,15} && \text{weight 1} \\
\oplus\ & S^2_{1,27} \oplus S^2_{3,27} \oplus S^2_{1,27} \cdot S^2_{2,27} \oplus S^2_{2,27} \cdot S^2_{3,27} && \text{weight 1} \\
\oplus\ & S^1_{1,2} \oplus S^1_{3,2} \oplus S^1_{1,2} \cdot S^1_{2,2} \oplus S^1_{2,2} \cdot S^1_{3,2} && \text{weight 1} \\
\oplus\ & S^3_{3,14} \oplus S^3_{1,14} \oplus S^3_{1,14} \cdot S^3_{2,14} \oplus S^3_{2,14} \cdot S^3_{3,14} && \text{weight 1} \\
\oplus\ & S^2_{0,0} \cdot S^2_{1,0} && \text{weight 1} \\
\oplus\ & S^2_{2,31} \cdot S^2_{3,31} && \text{weight 1} \\
\oplus\ & S^2_{3,0} \cdot S^2_{4,0} && \text{weight 1} \\
\oplus\ & S^3_{0,7} \cdot S^3_{1,7} && \text{weight 1} \\
\oplus\ & S^3_{2,6} \cdot S^3_{3,6} && \text{weight 1} \\
\oplus\ & S^3_{2,12} \cdot S^3_{3,12} && \text{weight 1} \\
\oplus\ & S^4_{2,19} \cdot S^4_{3,19}. && \text{weight 1}
\end{aligned}
$$

The total weight of the trail is 16.

6.B TRAIL EQUATION FOR MINIMORUS-1280

This trail equation was already given in Section 6.4.2.

$$
\begin{aligned}
& C^0_{51} \oplus C^1_0 \oplus C^1_{25} \oplus C^1_{33} \oplus C^1_{55} \oplus C^2_4 \oplus C^2_7 \oplus C^2_{29} \oplus C^2_{37} \\
& \oplus C^2_{38} \oplus C^2_{46} \oplus C^2_{51} \oplus C^3_{11} \oplus C^3_{20} \oplus C^3_{42} \oplus C^3_{50} \oplus C^4_{24} \\
&= S^0_{1,51} \cdot S^0_{2,51} \oplus S^0_{2,51} \cdot S^0_{3,51} \oplus S^0_{1,51} \oplus S^0_{3,51} && \text{weight 1} \\
& \oplus S^1_{1,25} \cdot S^1_{2,25} \oplus S^1_{2,25} \cdot S^1_{3,25} \oplus S^1_{1,25} \oplus S^1_{3,25} && \text{weight 1} \\
& \oplus S^1_{1,33} \cdot S^1_{2,33} \oplus S^1_{2,33} \cdot S^1_{3,33} \oplus S^1_{1,33} \oplus S^1_{3,33} && \text{weight 1} \\
& \oplus S^1_{1,55} \cdot S^1_{2,55} \oplus S^1_{2,55} \cdot S^1_{3,55} \oplus S^1_{1,55} \oplus S^1_{3,55} && \text{weight 1} \\
& \oplus S^2_{1,7} \cdot S^2_{2,7} \oplus S^2_{2,7} \cdot S^2_{3,7} \oplus S^2_{1,7} \oplus S^2_{3,7} && \text{weight 1} \\
& \oplus S^2_{1,29} \cdot S^2_{2,29} \oplus S^2_{2,29} \cdot S^2_{3,29} \oplus S^2_{1,29} \oplus S^2_{3,29} && \text{weight 1} \\
& \oplus S^2_{1,37} \cdot S^2_{2,37} \oplus S^2_{2,37} \cdot S^2_{3,37} \oplus S^2_{1,37} \oplus S^2_{3,37} && \text{weight 1} \\
& \oplus S^2_{1,51} \cdot S^2_{2,51} \oplus S^2_{2,51} \cdot S^2_{3,51} \oplus S^2_{1,51} \oplus S^2_{3,51} && \text{weight 1} \\
& \oplus S^3_{1,11} \cdot S^3_{2,11} \oplus S^3_{2,11} \cdot S^3_{3,11} \oplus S^3_{1,11} \oplus S^3_{3,11} && \text{weight 1} \\
& \oplus S^2_{0,0} \cdot S^2_{1,0} && \text{weight 1} \\
& \oplus S^2_{2,46} \cdot S^2_{3,46} && \text{weight 1} \\
& \oplus S^2_{3,0} \cdot S^2_{4,0} && \text{weight 1} \\
& \oplus S^3_{0,38} \cdot S^3_{1,38} && \text{weight 1} \\
& \oplus S^3_{2,20} \cdot S^3_{3,20} && \text{weight 1} \\
& \oplus S^3_{2,50} \cdot S^3_{3,50} && \text{weight 1} \\
& \oplus S^4_{2,24} \cdot S^4_{3,24} && \text{weight 1}
\end{aligned}
$$

The total weight of the trail is 16.

6.C TRAIL EQUATION FOR FULL MORUS-640

$$
\begin{aligned}
& C^0_{27} \oplus C^0_{59} \oplus C^0_{91} \oplus C^0_{123} \oplus C^1_0 \oplus C^1_2 \oplus C^1_8 \oplus C^1_{26} \\
& \oplus C^1_{32} \oplus C^1_{34} \oplus C^1_{40} \oplus C^1_{58} \oplus C^1_{64} \oplus C^1_{66} \oplus C^1_{72} \oplus C^1_{90} \\
& \oplus C^1_{96} \oplus C^1_{98} \oplus C^1_{104} \oplus C^1_{122} \oplus C^2_1 \oplus C^2_{13} \oplus C^2_{15} \oplus C^2_{27} \\
& \oplus C^2_{31} \oplus C^2_{33} \oplus C^2_{45} \oplus C^2_{47} \oplus C^2_{59} \oplus C^2_{63} \oplus C^2_{65} \oplus C^2_{77} \\
& \oplus C^2_{79} \oplus C^2_{91} \oplus C^2_{95} \oplus C^2_{97} \oplus C^2_{109} \oplus C^2_{111} \oplus C^2_{123} \oplus C^2_{127} \\
& \oplus C^3_6 \oplus C^3_{12} \oplus C^3_{14} \oplus C^3_{20} \oplus C^3_{38} \oplus C^3_{44} \oplus C^3_{46} \oplus C^3_{52} \\
& \oplus C^3_{70} \oplus C^3_{76} \oplus C^3_{78} \oplus C^3_{84} \oplus C^3_{102} \oplus C^3_{108} \oplus C^3_{110} \oplus C^3_{116} \\
& \oplus C^4_{19} \oplus C^4_{51} \oplus C^4_{83} \oplus C^4_{115} \\
&= S^1_{2,0} \cdot S^1_{3,0} \oplus S^1_{2,0} \cdot S^1_{3,96} \oplus S^1_{2,32} \cdot S^1_{3,0} \oplus S^1_{2,96} \cdot S^1_{3,96}
\end{aligned}
$$

$$\oplus\, S^1_{2,96}\cdot S^1_{3,64} \oplus S^1_{2,64}\cdot S^1_{3,64} \oplus S^1_{2,64}\cdot S^1_{3,32} \oplus S^1_{2,32}\cdot S^1_{3,32} \quad \text{weight 3}$$

$$\oplus\, S^2_{2,13}\cdot S^2_{3,13} \oplus S^2_{2,13}\cdot S^2_{3,109} \oplus S^2_{2,45}\cdot S^2_{3,13} \oplus S^2_{2,109}\cdot S^2_{3,109}$$

$$\oplus\, S^2_{2,45}\cdot S^2_{3,45} \oplus S^2_{2,109}\cdot S^2_{3,77} \oplus S^2_{2,77}\cdot S^2_{3,45} \oplus S^2_{2,77}\cdot S^2_{3,77} \quad \text{weight 3}$$

$$\oplus\, S^3_{2,20}\cdot S^3_{3,20} \oplus S^3_{2,20}\cdot S^3_{3,116} \oplus S^3_{2,52}\cdot S^3_{3,20} \oplus S^3_{2,116}\cdot S^3_{3,116}$$

$$\oplus\, S^3_{2,52}\cdot S^3_{3,52} \oplus S^3_{2,116}\cdot S^3_{3,84} \oplus S^3_{2,84}\cdot S^3_{3,52} \oplus S^3_{2,84}\cdot S^3_{3,84} \quad \text{weight 3}$$

$$\oplus\, S^0_{1,27} \oplus S^0_{1,27}\cdot S^0_{2,27} \oplus S^0_{2,27}\cdot S^0_{3,27} \oplus S^0_{3,27} \quad \text{weight 1}$$

$$\oplus\, S^0_{1,59} \oplus S^0_{1,59}\cdot S^0_{2,59} \oplus S^0_{2,59}\cdot S^0_{3,59} \oplus S^0_{3,59} \quad \text{weight 1}$$

$$\oplus\, S^0_{1,91} \oplus S^0_{1,91}\cdot S^0_{2,91} \oplus S^0_{2,91}\cdot S^0_{3,91} \oplus S^0_{3,91} \quad \text{weight 1}$$

$$\oplus\, S^0_{1,123}\cdot S^0_{2,123} \oplus S^0_{2,123}\cdot S^0_{3,123} \oplus S^0_{1,123} \oplus S^0_{3,123} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,2} \oplus S^1_{1,2}\cdot S^1_{2,2} \oplus S^1_{2,2}\cdot S^1_{3,2} \oplus S^1_{3,2} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,8} \oplus S^1_{1,8}\cdot S^1_{2,8} \oplus S^1_{2,8}\cdot S^1_{3,8} \oplus S^1_{3,8} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,26} \oplus S^1_{1,26}\cdot S^1_{2,26} \oplus S^1_{2,26}\cdot S^1_{3,26} \oplus S^1_{3,26} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,34} \oplus S^1_{1,34}\cdot S^1_{2,34} \oplus S^1_{2,34}\cdot S^1_{3,34} \oplus S^1_{3,34} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,40} \oplus S^1_{1,40}\cdot S^1_{2,40} \oplus S^1_{2,40}\cdot S^1_{3,40} \oplus S^1_{3,40} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,58} \oplus S^1_{1,58}\cdot S^1_{2,58} \oplus S^1_{2,58}\cdot S^1_{3,58} \oplus S^1_{3,58} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,66} \oplus S^1_{1,66}\cdot S^1_{2,66} \oplus S^1_{2,66}\cdot S^1_{3,66} \oplus S^1_{3,66} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,72} \oplus S^1_{1,72}\cdot S^1_{2,72} \oplus S^1_{2,72}\cdot S^1_{3,72} \oplus S^1_{3,72} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,90} \oplus S^1_{1,90}\cdot S^1_{2,90} \oplus S^1_{2,90}\cdot S^1_{3,90} \oplus S^1_{3,90} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,98} \oplus S^1_{1,98}\cdot S^1_{2,98} \oplus S^1_{2,98}\cdot S^1_{3,98} \oplus S^1_{3,98} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,104} \oplus S^1_{1,104}\cdot S^1_{2,104} \oplus S^1_{2,104}\cdot S^1_{3,104} \oplus S^1_{3,104} \quad \text{weight 1}$$

$$\oplus\, S^1_{1,122} \oplus S^1_{3,122} \oplus S^1_{1,122}\cdot S^1_{2,122} \oplus S^1_{2,122}\cdot S^1_{3,122} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,1} \oplus S^2_{1,1}\cdot S^2_{2,1} \oplus S^2_{2,1}\cdot S^2_{3,1} \oplus S^2_{3,1} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,7} \oplus S^2_{1,7}\cdot S^2_{2,7} \oplus S^2_{2,7}\cdot S^2_{3,103} \oplus S^2_{3,103} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,15} \oplus S^2_{1,15}\cdot S^2_{2,15} \oplus S^2_{2,15}\cdot S^2_{3,15} \oplus S^2_{3,15} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,27} \oplus S^2_{1,27}\cdot S^2_{2,27} \oplus S^2_{2,27}\cdot S^2_{3,27} \oplus S^2_{3,27} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,33} \oplus S^2_{1,33}\cdot S^2_{2,33} \oplus S^2_{2,33}\cdot S^2_{3,33} \oplus S^2_{3,33} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,39} \oplus S^2_{1,39}\cdot S^2_{2,39} \oplus S^2_{2,39}\cdot S^2_{3,7} \oplus S^2_{3,7} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,47} \oplus S^2_{1,47}\cdot S^2_{2,47} \oplus S^2_{2,47}\cdot S^2_{3,47} \oplus S^2_{3,47} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,59} \oplus S^2_{1,59}\cdot S^2_{2,59} \oplus S^2_{2,59}\cdot S^2_{3,59} \oplus S^2_{3,59} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,65} \oplus S^2_{1,65}\cdot S^2_{2,65} \oplus S^2_{2,65}\cdot S^2_{3,65} \oplus S^2_{3,65} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,71} \oplus S^2_{1,71}\cdot S^2_{2,71} \oplus S^2_{2,71}\cdot S^2_{3,39} \oplus S^2_{3,39} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,79} \oplus S^2_{1,79}\cdot S^2_{2,79} \oplus S^2_{2,79}\cdot S^2_{3,79} \oplus S^2_{3,79} \quad \text{weight 1}$$

$$\oplus\, S^2_{1,91} \oplus S^2_{1,91}\cdot S^2_{2,91} \oplus S^2_{2,91}\cdot S^2_{3,91} \oplus S^2_{3,91} \quad \text{weight 1}$$

$$\oplus\ S^2_{1,97} \oplus S^2_{1,97} \cdot S^2_{2,97} \oplus S^2_{2,97} \cdot S^2_{3,97} \oplus S^2_{3,97} \quad \text{weight 1}$$
$$\oplus\ S^2_{1,103} \oplus S^2_{1,103} \cdot S^2_{2,103} \oplus S^2_{2,103} \cdot S^2_{3,71} \oplus S^2_{3,71} \quad \text{weight 1}$$
$$\oplus\ S^2_{1,111} \oplus S^2_{1,111} \cdot S^2_{2,111} \oplus S^2_{2,111} \cdot S^2_{3,111} \oplus S^2_{3,111} \quad \text{weight 1}$$
$$\oplus\ S^2_{1,123} \oplus S^2_{3,123} \oplus S^2_{2,123} \cdot S^2_{3,123} \oplus S^2_{1,123} \cdot S^2_{2,123} \quad \text{weight 1}$$
$$\oplus\ S^3_{1,14} \oplus S^3_{1,14} \cdot S^3_{2,14} \oplus S^3_{2,14} \cdot S^3_{3,14} \oplus S^3_{3,14} \quad \text{weight 1}$$
$$\oplus\ S^3_{1,46} \oplus S^3_{1,46} \cdot S^3_{2,46} \oplus S^3_{2,46} \cdot S^3_{3,46} \oplus S^3_{3,46} \quad \text{weight 1}$$
$$\oplus\ S^3_{1,78} \oplus S^3_{1,78} \cdot S^3_{2,78} \oplus S^3_{2,78} \cdot S^3_{3,78} \oplus S^3_{3,78} \quad \text{weight 1}$$
$$\oplus\ S^3_{1,110} \oplus S^3_{3,110} \oplus S^3_{1,110} \cdot S^3_{2,110} \oplus S^3_{2,110} \cdot S^3_{3,110} \quad \text{weight 1}$$
$$\oplus\ S^2_{0,0} \cdot S^2_{1,0} \quad \text{weight 1}$$
$$\oplus\ S^2_{0,32} \cdot S^2_{1,32} \quad \text{weight 1}$$
$$\oplus\ S^2_{0,64} \cdot S^2_{1,64} \quad \text{weight 1}$$
$$\oplus\ S^2_{0,96} \cdot S^2_{1,96} \quad \text{weight 1}$$
$$\oplus\ S^2_{2,31} \cdot S^2_{3,31} \quad \text{weight 1}$$
$$\oplus\ S^2_{2,63} \cdot S^2_{3,63} \quad \text{weight 1}$$
$$\oplus\ S^2_{2,95} \cdot S^2_{3,95} \quad \text{weight 1}$$
$$\oplus\ S^2_{2,127} \cdot S^2_{3,127} \quad \text{weight 1}$$
$$\oplus\ S^2_{3,32} \cdot S^2_{4,0} \quad \text{weight 1}$$
$$\oplus\ S^2_{3,64} \cdot S^2_{4,32} \quad \text{weight 1}$$
$$\oplus\ S^2_{3,96} \cdot S^2_{4,64} \quad \text{weight 1}$$
$$\oplus\ S^2_{3,0} \cdot S^2_{4,96} \quad \text{weight 1}$$
$$\oplus\ S^3_{0,7} \cdot S^3_{1,7} \quad \text{weight 1}$$
$$\oplus\ S^3_{0,39} \cdot S^3_{1,39} \quad \text{weight 1}$$
$$\oplus\ S^3_{0,71} \cdot S^3_{1,71} \quad \text{weight 1}$$
$$\oplus\ S^3_{0,103} \cdot S^3_{1,103} \quad \text{weight 1}$$
$$\oplus\ S^3_{2,6} \cdot S^3_{3,6} \quad \text{weight 1}$$
$$\oplus\ S^3_{2,12} \cdot S^3_{3,12} \quad \text{weight 1}$$
$$\oplus\ S^3_{2,38} \cdot S^3_{3,38} \quad \text{weight 1}$$
$$\oplus\ S^3_{2,44} \cdot S^3_{3,44} \quad \text{weight 1}$$
$$\oplus\ S^3_{2,70} \cdot S^3_{3,70} \quad \text{weight 1}$$
$$\oplus\ S^3_{2,76} \cdot S^3_{3,76} \quad \text{weight 1}$$
$$\oplus\ S^3_{2,102} \cdot S^3_{3,102} \quad \text{weight 1}$$
$$\oplus\ S^3_{2,108} \cdot S^3_{3,108} \quad \text{weight 1}$$
$$\oplus\ S^4_{2,19} \cdot S^4_{3,19} \quad \text{weight 1}$$

$\oplus\ S^4_{2,51} \cdot S^4_{3,51}$ weight 1

$\oplus\ S^4_{2,83} \cdot S^4_{3,83}$ weight 1

$\oplus\ S^4_{2,115} \cdot S^4_{3,115}$ weight 1

The total weight of the trail is 73.

6.D TRAIL EQUATION FOR FULL MORUS-1280

$$
\begin{aligned}
& C^0_{51} \oplus C^0_{115} \oplus C^0_{179} \oplus C^0_{243} \oplus C^1_0 \oplus C^1_{25} \oplus C^1_{33} \oplus C^1_{55} \\
\oplus\ & C^1_{64} \oplus C^1_{89} \oplus C^1_{97} \oplus C^1_{119} \oplus C^1_{128} \oplus C^1_{153} \oplus C^1_{161} \oplus C^1_{183} \\
\oplus\ & C^1_{192} \oplus C^1_{217} \oplus C^1_{225} \oplus C^1_{247} \oplus C^2_4 \oplus C^2_7 \oplus C^2_{29} \oplus C^2_{37} \\
\oplus\ & C^2_{38} \oplus C^2_{46} \oplus C^2_{51} \oplus C^2_{68} \oplus C^2_{71} \oplus C^2_{93} \oplus C^2_{101} \oplus C^2_{102} \\
\oplus\ & C^2_{110} \oplus C^2_{115} \oplus C^2_{132} \oplus C^2_{135} \oplus C^2_{157} \oplus C^2_{165} \oplus C^2_{166} \oplus C^2_{174} \\
\oplus\ & C^2_{179} \oplus C^2_{196} \oplus C^2_{199} \oplus C^2_{221} \oplus C^2_{229} \oplus C^2_{230} \oplus C^2_{238} \oplus C^2_{243} \\
\oplus\ & C^3_{11} \oplus C^3_{20} \oplus C^3_{42} \oplus C^3_{50} \oplus C^3_{75} \oplus C^3_{84} \oplus C^3_{106} \oplus C^3_{114} \\
\oplus\ & C^3_{139} \oplus C^3_{148} \oplus C^3_{170} \oplus C^3_{178} \oplus C^3_{203} \oplus C^3_{212} \oplus C^3_{234} \oplus C^3_{242} \\
\oplus\ & C^4_{24} \oplus C^4_{88} \oplus C^4_{152} \oplus C^4_{216} \\
=\ & S^1_{2,0} \cdot S^1_{3,192} \oplus S^1_{2,0} \cdot S^1_{3,0} \oplus S^1_{2,64} \cdot S^1_{3,0} \\
& \quad \oplus S^1_{2,64} \cdot S^1_{3,64} \oplus S^1_{2,128} \cdot S^1_{3,64} \oplus S^1_{2,128} \cdot S^1_{3,128} \\
& \quad \oplus S^1_{2,192} \cdot S^1_{3,128} \oplus S^1_{2,192} \cdot S^1_{3,192} && \text{weight 3} \\
\oplus\ & S^2_{2,4} \cdot S^2_{3,4} \oplus S^2_{2,68} \cdot S^2_{3,4} \oplus S^2_{2,68} \cdot S^2_{3,68} \\
& \quad \oplus S^2_{2,132} \cdot S^2_{3,68} \oplus S^2_{2,132} \cdot S^2_{3,132} \oplus S^2_{2,196} \cdot S^2_{3,132} \\
& \quad \oplus S^2_{2,196} \cdot S^2_{3,196} \oplus S^2_{2,4} \cdot S^2_{3,196} && \text{weight 3} \\
\oplus\ & S^2_{2,102} \cdot S^2_{3,38} \oplus S^2_{2,102} \cdot S^2_{3,102} \oplus S^2_{2,166} \cdot S^2_{3,102} \\
& \quad \oplus S^2_{2,166} \cdot S^2_{3,166} \oplus S^2_{2,230} \cdot S^2_{3,166} \oplus S^2_{2,230} \cdot S^2_{3,230} \\
& \quad \oplus S^2_{2,38} \cdot S^2_{3,230} \oplus S^2_{2,38} \cdot S^2_{3,38} && \text{weight 3} \\
\oplus\ & S^3_{2,42} \cdot S^3_{3,42} \oplus S^3_{2,106} \cdot S^3_{3,42} \oplus S^3_{2,106} \cdot S^3_{3,106} \\
& \quad \oplus S^3_{2,170} \cdot S^3_{3,106} \oplus S^3_{2,170} \cdot S^3_{3,170} \oplus S^3_{2,234} \cdot S^3_{3,170} \\
& \quad \oplus S^3_{2,234} \cdot S^3_{3,234} \oplus S^3_{2,42} \cdot S^3_{3,234} && \text{weight 3} \\
\oplus\ & S^0_{1,51} \cdot S^0_{2,51} \oplus S^0_{1,51} \oplus S^0_{2,51} \cdot S^0_{3,51} \oplus S^0_{3,51} && \text{weight 1} \\
\oplus\ & S^0_{1,115} \cdot S^0_{2,115} \oplus S^0_{1,115} \oplus S^0_{2,115} \cdot S^0_{3,115} \oplus S^0_{3,115} && \text{weight 1} \\
\oplus\ & S^0_{1,179} \cdot S^0_{2,179} \oplus S^0_{1,179} \oplus S^0_{2,179} \cdot S^0_{3,179} \oplus S^0_{3,179} && \text{weight 1} \\
\oplus\ & S^0_{1,243} \cdot S^0_{2,243} \oplus S^0_{1,243} \oplus S^0_{2,243} \cdot S^0_{3,243} \oplus S^0_{3,243} && \text{weight 1} \\
\oplus\ & S^1_{1,25} \cdot S^1_{2,25} \oplus S^1_{1,25} \oplus S^1_{2,25} \cdot S^1_{3,25} \oplus S^1_{3,25} && \text{weight 1} \\
\oplus\ & S^1_{1,33} \cdot S^1_{2,33} \oplus S^1_{1,33} \oplus S^1_{2,33} \cdot S^1_{3,33} \oplus S^1_{3,33} && \text{weight 1}
\end{aligned}
$$

$\oplus\ S^1_{1,55} \cdot S^1_{2,55} \oplus S^1_{1,55} \oplus S^1_{2,55} \cdot S^1_{3,55} \oplus S^1_{3,55}$ weight 1

$\oplus\ S^1_{1,89} \cdot S^1_{2,89} \oplus S^1_{1,89} \oplus S^1_{2,89} \cdot S^1_{3,89} \oplus S^1_{3,89}$ weight 1

$\oplus\ S^1_{1,97} \cdot S^1_{2,97} \oplus S^1_{1,97} \oplus S^1_{2,97} \cdot S^1_{3,97} \oplus S^1_{3,97}$ weight 1

$\oplus\ S^1_{1,119} \cdot S^1_{2,119} \oplus S^1_{1,119} \oplus S^1_{2,119} \cdot S^1_{3,119} \oplus S^1_{3,119}$ weight 1

$\oplus\ S^1_{1,153} \cdot S^1_{2,153} \oplus S^1_{1,153} \oplus S^1_{2,153} \cdot S^1_{3,153} \oplus S^1_{3,153}$ weight 1

$\oplus\ S^1_{1,161} \cdot S^1_{2,161} \oplus S^1_{1,161} \oplus S^1_{2,161} \cdot S^1_{3,161} \oplus S^1_{3,161}$ weight 1

$\oplus\ S^1_{1,183} \cdot S^1_{2,183} \oplus S^1_{1,183} \oplus S^1_{2,183} \cdot S^1_{3,183} \oplus S^1_{3,183}$ weight 1

$\oplus\ S^1_{1,217} \cdot S^1_{2,217} \oplus S^1_{1,217} \oplus S^1_{2,217} \cdot S^1_{3,217} \oplus S^1_{3,217}$ weight 1

$\oplus\ S^1_{1,225} \cdot S^1_{2,225} \oplus S^1_{1,225} \oplus S^1_{2,225} \cdot S^1_{3,225} \oplus S^1_{3,225}$ weight 1

$\oplus\ S^1_{1,247} \cdot S^1_{2,247} \oplus S^1_{1,247} \oplus S^1_{2,247} \cdot S^1_{3,247} \oplus S^1_{3,247}$ weight 1

$\oplus\ S^2_{1,7} \cdot S^2_{2,7} \oplus S^2_{1,7} \oplus S^2_{2,7} \cdot S^2_{3,7} \oplus S^2_{3,7}$ weight 1

$\oplus\ S^2_{1,29} \cdot S^2_{2,29} \oplus S^2_{1,29} \oplus S^2_{2,29} \cdot S^2_{3,29} \oplus S^2_{3,29}$ weight 1

$\oplus\ S^2_{1,37} \cdot S^2_{2,37} \oplus S^2_{1,37} \oplus S^2_{2,37} \cdot S^2_{3,37} \oplus S^2_{3,37}$ weight 1

$\oplus\ S^2_{1,51} \cdot S^2_{2,51} \oplus S^2_{1,51} \oplus S^2_{2,51} \cdot S^2_{3,51} \oplus S^2_{3,51}$ weight 1

$\oplus\ S^2_{1,71} \cdot S^2_{2,71} \oplus S^2_{1,71} \oplus S^2_{2,71} \cdot S^2_{3,71} \oplus S^2_{3,71}$ weight 1

$\oplus\ S^2_{1,93} \cdot S^2_{2,93} \oplus S^2_{1,93} \oplus S^2_{2,93} \cdot S^2_{3,93} \oplus S^2_{3,93}$ weight 1

$\oplus\ S^2_{1,101} \cdot S^2_{2,101} \oplus S^2_{1,101} \oplus S^2_{2,101} \cdot S^2_{3,101} \oplus S^2_{3,101}$ weight 1

$\oplus\ S^2_{1,115} \cdot S^2_{2,115} \oplus S^2_{1,115} \oplus S^2_{2,115} \cdot S^2_{3,115} \oplus S^2_{3,115}$ weight 1

$\oplus\ S^2_{1,135} \cdot S^2_{2,135} \oplus S^2_{1,135} \oplus S^2_{2,135} \cdot S^2_{3,135} \oplus S^2_{3,135}$ weight 1

$\oplus\ S^2_{1,157} \cdot S^2_{2,157} \oplus S^2_{1,157} \oplus S^2_{2,157} \cdot S^2_{3,157} \oplus S^2_{3,157}$ weight 1

$\oplus\ S^2_{1,165} \cdot S^2_{2,165} \oplus S^2_{1,165} \oplus S^2_{2,165} \cdot S^2_{3,165} \oplus S^2_{3,165}$ weight 1

$\oplus\ S^2_{1,179} \cdot S^2_{2,179} \oplus S^2_{1,179} \oplus S^2_{2,179} \cdot S^2_{3,179} \oplus S^2_{3,179}$ weight 1

$\oplus\ S^2_{1,199} \cdot S^2_{2,199} \oplus S^2_{1,199} \oplus S^2_{2,199} \cdot S^2_{3,199} \oplus S^2_{3,199}$ weight 1

$\oplus\ S^2_{1,221} \cdot S^2_{2,221} \oplus S^2_{1,221} \oplus S^2_{2,221} \cdot S^2_{3,221} \oplus S^2_{3,221}$ weight 1

$\oplus\ S^2_{1,229} \cdot S^2_{2,229} \oplus S^2_{1,229} \oplus S^2_{2,229} \cdot S^2_{3,229} \oplus S^2_{3,229}$ weight 1

$\oplus\ S^2_{1,243} \cdot S^2_{2,243} \oplus S^2_{1,243} \oplus S^2_{2,243} \cdot S^2_{3,243} \oplus S^2_{3,243}$ weight 1

$\oplus\ S^3_{1,11} \cdot S^3_{2,11} \oplus S^3_{1,11} \oplus S^3_{2,11} \cdot S^3_{3,11} \oplus S^3_{3,11}$ weight 1

$\oplus\ S^3_{1,75} \cdot S^3_{2,75} \oplus S^3_{1,75} \oplus S^3_{2,75} \cdot S^3_{3,75} \oplus S^3_{3,75}$ weight 1

$\oplus\ S^3_{1,139} \cdot S^3_{2,139} \oplus S^3_{1,139} \oplus S^3_{2,139} \cdot S^3_{3,139} \oplus S^3_{3,139}$ weight 1

$\oplus\ S^3_{1,203} \cdot S^3_{2,203} \oplus S^3_{1,203} \oplus S^3_{2,203} \cdot S^3_{3,203} \oplus S^3_{3,203}$ weight 1

$\oplus\ S^2_{0,0} \cdot S^2_{1,0}$ weight 1

$\oplus\ S^2_{0,64} \cdot S^2_{1,64}$ weight 1

$\oplus\ S^2_{0,128} \cdot S^2_{1,128}$ weight 1

$\oplus\ S^2_{0,192} \cdot S^2_{1,192}$ weight 1
$\oplus\ S^3_{0,230} \cdot S^3_{1,230}$ weight 1
$\oplus\ S^2_{2,46} \cdot S^2_{3,46}$ weight 1
$\oplus\ S^2_{2,110} \cdot S^2_{3,110}$ weight 1
$\oplus\ S^2_{2,174} \cdot S^2_{3,174}$ weight 1
$\oplus\ S^2_{2,238} \cdot S^2_{3,238}$ weight 1
$\oplus\ S^2_{3,64} \cdot S^2_{4,0}$ weight 1
$\oplus\ S^2_{3,128} \cdot S^2_{4,64}$ weight 1
$\oplus\ S^2_{3,192} \cdot S^2_{4,128}$ weight 1
$\oplus\ S^2_{3,0} \cdot S^2_{4,192}$ weight 1
$\oplus\ S^3_{0,38} \cdot S^3_{1,38}$ weight 1
$\oplus\ S^3_{0,102} \cdot S^3_{1,102}$ weight 1
$\oplus\ S^3_{0,166} \cdot S^3_{1,166}$ weight 1
$\oplus\ S^3_{2,20} \cdot S^3_{3,20}$ weight 1
$\oplus\ S^3_{2,50} \cdot S^3_{3,50}$ weight 1
$\oplus\ S^3_{2,84} \cdot S^3_{3,84}$ weight 1
$\oplus\ S^3_{2,114} \cdot S^3_{3,114}$ weight 1
$\oplus\ S^3_{2,148} \cdot S^3_{3,148}$ weight 1
$\oplus\ S^3_{2,178} \cdot S^3_{3,178}$ weight 1
$\oplus\ S^3_{2,212} \cdot S^3_{3,212}$ weight 1
$\oplus\ S^3_{2,242} \cdot S^3_{3,242}$ weight 1
$\oplus\ S^4_{2,24} \cdot S^4_{3,24}$ weight 1
$\oplus\ S^4_{2,88} \cdot S^4_{3,88}$ weight 1
$\oplus\ S^4_{2,152} \cdot S^4_{3,152}$ weight 1
$\oplus\ S^4_{2,216} \cdot S^4_{3,216}$ weight 1

The total weight of the trail is 76.

Part III

VERIFYING

7 A COQ PROOF OF THE CORRECTNESS OF X25519 IN TWEETNACL

In this chapter we provide a computer-verified proof that the X25519 implementation in TweetNaCl correctly matches RFC 7748 and that RFC 7748 correctly computes a scalar multiplication on the elliptic curve Curve25519.

7.1 INTRODUCTION

The Networking and Cryptography library (NaCl) [BLS12] is an easy-to-use, high-security, high-speed cryptography library. It uses specialized code for different platforms, which makes it rather complex and hard to audit. TweetNaCl [Ber+15b] is a compact re-implementation in C of the core functionalities of NaCl and is claimed to be *"the first cryptographic library that allows correct functionality to be verified by auditors with reasonable effort"* [Ber+15b]. The original paper presenting TweetNaCl describes some effort to support this claim, for example, formal verification of memory safety, but does not actually prove correctness of any of the primitives implemented by the library.

One core component of TweetNaCl (and NaCl) is the key-exchange protocol X25519 presented by Bernstein in [Bero6b]. This protocol is standardized in RFC 7748 and used by a wide variety of applications [Thi] such as Secure Shell (SSH), Signal Protocol, Tor, Zcash, and Transport Layer Security (TLS) to establish a shared secret over an insecure channel. The X25519 key-exchange protocol is an x-coordinate-only elliptic-curve Merkle-Diffie-Hellman key exchange using the Montgomery curve $E : y^2 = x^3 + 486662x^2 + x$ over the field $\mathbb{F}_{2^{255}-19}$. Note that originally, the name "Curve25519" referred to the key-exchange protocol, but Bernstein suggested to rename the protocol to X25519 and to use the name Curve25519 for the underlying elliptic curve [Bero8a]. We use this updated terminology in this chapter.

CONTRIBUTIONS. In short, in this chapter we provide a computer-verified proof that the X25519 implementation in TweetNaCl matches the mathematical definition of the function given in [Bero6b, Sec. 2]. This proof is done in three steps:

1. We first formalize RFC 7748 [LHT] in Coq [Coq].
2. We prove equivalence of the C implementation of X25519 to our RFC formalization. This part of the proof uses the Verifiable Software Toolchain (VST) [App12] to establish a link between C and Coq. VST uses separation logic [Hoa69; Rey02] to show that

the semantics of the program satisfies a functional specification in Coq. To the best of our knowledge, this is the first time that VST is used in the formal proof of correctness of an implementation of an asymmetric cryptographic primitive.

3. We prove that the Coq formalization of the RFC matches the mathematical definition of X25519 as given in [Ber06b, Sec. 2]. We do this by extending the Coq library for elliptic curves [BS14] by Bartzia and Strub to support Montgomery curves, and in particular Curve25519.

To our knowledge, this verification effort is the first to not just connect a low-level implementation to a higher-level implementation (or "specification"), but to prove correctness all the way up to the mathematical definition in terms of scalar multiplication on an elliptic curve. As a consequence, the result of this chapter can readily be used in mechanized proofs arguing about the security of cryptographic constructions on the more abstract level of operations in groups and related problems, like the elliptic-curve discrete-logarithm (ECDLP) or elliptic-curve Diffie-Hellman (ECDH) problem. Also, connecting our formalization of the RFC to the mathematical definition significantly increases trust into the correctness of the formalization and reduces the effort of manually auditing the formalization.

THE BIGGER PICTURE OF HIGH-ASSURANCE CRYPTO. This work fits into the bigger area of *high-assurance* cryptography, i. e., a line of work that applies techniques and tools from formal methods to obtain computer-verified guarantees for cryptographic software. Traditionally, high-assurance cryptography is concerned with three main properties of cryptographic software:

1. verifying CORRECTNESS of cryptographic software, typically against a high-level specification;

2. verifying IMPLEMENTATION SECURITY and in particular security against timing attacks; and

3. verifying CRYPTOGRAPHIC SECURITY notions of primitives and protocols through computer-checked reductions from some assumed-to-be-hard mathematical problem.

A recent addition to this triplet (or rather an extension of implementation security) is security also against attacks exploiting speculative execution; see, e. g., [Cau+20]. This chapter targets only the first point and attempts to make results immediately usable for verification efforts of cryptographic security.

Verification of implementation security is probably equally important as verification of correctness, but working on the C language level as we do in this chapter is not helpful. To obtain guarantees of security

against timing-attack we recommend verifying *compiled* code on LLVM level with, e. g., ct-verif [Alm+16], or even better on binary level with, e. g., BINSEC/REL [LAD20].

RELATED WORK. The field of computer-aided cryptography, i. e., using computer-verified proofs to strengthen our trust into cryptographic constructions and cryptographic software, has seen massive progress in the recent past. This progress, the state of the art, and future challenges have recently been compiled in a SoK paper by Barbosa, Barthe, Bhargavan, Blanchet, Cremers, Liao, and Parno [Bar+19]. This SoK paper, in Section III.C, also gives an overview of verification efforts of X25519 software. What all the previous approaches have in common is that they prove correctness by verifying that some low-level implementation matches a higher-level specification. This specification is kept in terms of a sequence of finite-field operations, typically close to the pseudocode in RFC 7748.

There are two general approaches to establish this link between low-level code and higher-level specification: Synthesize low-level code from the specification or write the low-level code by hand and prove that it matches the specification.

The X25519 implementation from the Evercrypt project [Pro+19] uses a low-level language called Vale that translates directly to assembly and proves equivalence to a high-level specification in F*. In [ZBB16], Zinzindohoué, Bartzia, and Bhargavan describe a verified extensible library of elliptic curves in F* [Pro+17]. This served as ground work for the cryptographic library HACL* [Zin+17] used in the Network Security Services (NSS) suite from Mozilla. The approach they use is a combination of proving and synthesising: A fairly low-level implementation written in Low* is proven to be equivalent to a high-level specification in F*. The Low* code is then compiled to C using an unverified and thus trusted compiler called Kremlin. The difference between this approach and ours is highlighted in Figure 7.1.

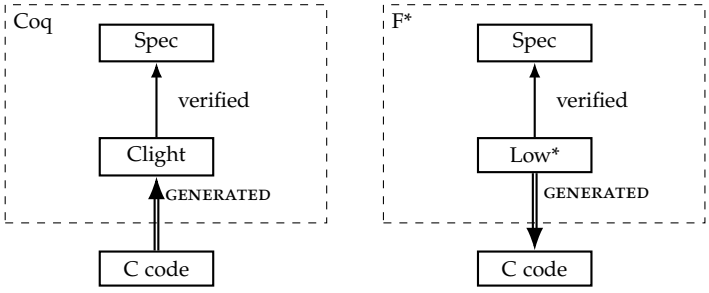


Figure 7.1: The VST approach (ours, on the left) compared to the F*/Low* approach (on the right, [Pro+19; ZBB16; Pro+17; Zin+17]).

Coq not only allows verification but also synthesis [Chl10]. Erbsen, Philipoom, Gross, and Chlipala make use of it to have correct-by-construction finite-field arithmetic, which is used to synthesize certified elliptic-curve crypto software [Phi18; Erb17; Erb+16]. This software suite is now being used in BoringSSL [Erb+19].

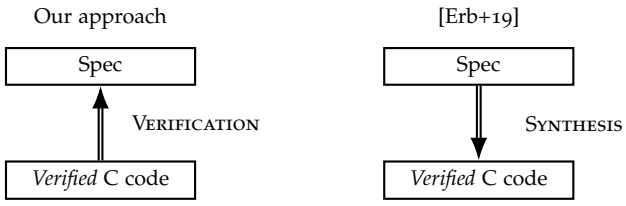


Figure 7.2: Our approach compared to [Erb+19].

All of these X25519 verification efforts use a clean-slate approach to obtain code and proofs (e. g., see Figure 7.2). Our effort targets existing software; we are aware of only one earlier work verifying existing X25519 software: In [Che+14], Chen, Hsu, Lin, Schwabe, Tsai, Wang, Yang, and Yang present a mechanized proof of two assembly-level implementations of the core function of X25519. Their proof takes a different approach from ours. It uses heavy annotation of the assembly-level code in order to "guide" a SAT solver; also, it does not cover the full X25519 functionality and does not make the link to the mathematical definition from [Ber06b]. As a consequence, this work would not find bugs in any of the routines processing the scalar (like "clamping", see Section 7.2.2), bugs in the serialization routines or, maybe most importantly, bugs in the high-level specification that the code is verified against.

Finally, in terms of languages and tooling the work closest to what we present here is the proof of correctness of OpenSSL's implementations of HMAC [Ber+15a], and mbedTLS' implementations of HMAC-DRBG [Ye+17] and SHA-256 [App15]. As those are all symmetric primitives without the rich mathematical structure of finite fields and elliptic curves the actual proofs are quite different.

REPRODUCING THE PROOFS. To maximize reusability of our results we place the code of our formal proof presented in this chapter into the public domain. It is available in the associated materials of this thesis (Section 1.3) and at `https://doi.org/10.5281/zenodo.4439686` with instructions of how to compile and verify our proof. A description of the content of the code archive is provided in Section 7.C.

ORGANIZATION OF THIS CHAPTER. Section 7.2 gives the necessary background on Curve25519 and X25519 implementations and a brief explanation of how formal verification works. Section 7.3 provides

our Coq formalization of X25519 as specified in RFC 7748 [LHT]. Section 7.4 provides the specification of X25519 in TweetNaCl and some of the proof techniques used to show the correctness with respect to RFC 7748 [LHT]. Section 7.5 describes our extension of the formal library by Bartzia and Strub and the proof of correctness of X25519 implementation with respect to Bernstein's specification [Ber08a]. Finally, in Section 7.6 we discuss the trusted code base of our proofs and conclude with some lessons learned about TweetNaCl and with sketching the effort required to extend our work to other elliptic-curve software.

Figure 7.3 shows a graph of dependencies of the proofs. C source files are represented by .C while .V corresponds to Coq files. In a nutshell, we formalize X25519 into a Coq function `RFC`, and we write a specification in separation logic with VST. The first step of CompCert compilation (clightgen) is used to translate the C source code into a DSL in Coq (Clight). By using VST, we step through the translated instructions and verify that the program satisfies the specification. Additionally, we formally define the scalar multiplication over elliptic curves and show that, under the same preconditions as used in the specification, `RFC` computes the desired results.

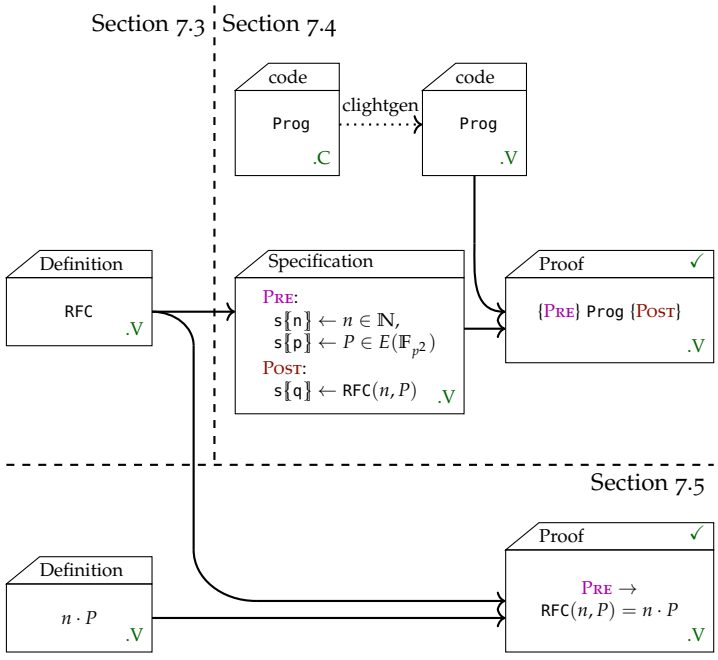


Figure 7.3: Structure of the proof.

7.2 PRELIMINARIES

In this section, we first give a brief summary of the mathematical background on elliptic curves. We then describe X25519 and its implementation in TweetNaCl. Finally, we provide a brief description of the formal tools we use in our proofs.

7.2.1 *Arithmetic on Montgomery curves*

DEFINITION 7.2.1. *Given a field* $\mathbb{K}$*, and* $a, b \in \mathbb{K}$ *such that* $a^2 \neq 4$ *and* $b \neq 0$*,* $M_{a,b}$ *is the Montgomery curve defined over* $\mathbb{K}$ *with equation*

$$M_{a,b} : by^2 = x^3 + ax^2 + x.$$

DEFINITION 7.2.2. *For any algebraic extension* $Ł \supseteq \mathbb{K}$*, we call* $M_{a,b}(Ł)$ *the set of* Ł*-rational points, defined as*

$$M_{a,b}(Ł) = \{\mathcal{O}\} \cup \{(x, y) \in Ł \times Ł \mid by^2 = x^3 + ax^2 + x\}.$$

Here, the additional element $\mathcal{O}$ *denotes the point at infinity.*

For $M_{a,b}$ over a finite field $\mathbb{F}_p$, the parameter b is known as the "twisting factor". For $b' \in \mathbb{F}_p \backslash \{0\}$ and $b' \neq b$, the curves $M_{a,b}$ and $M_{a,b'}$ are isomorphic via $(x, y) \mapsto (x, \sqrt{b/b'} \cdot y)$.

DEFINITION 7.2.3. *When* b'/b *is not a square in* $\mathbb{F}_p$*,* $M_{a,b'}$ *is a* quadratic twist *of* $M_{a,b}$*, i.e., a curve that is isomorphic over* $\mathbb{F}_{p^2}$ *[CS18].*

Points in $M_{a,b}(\mathbb{K})$ can be equipped with a structure of an abelian group with the addition operation $+$ and with neutral element the point at infinity $\mathcal{O}$. For a point $P \in M_{a,b}(\mathbb{K})$ and a positive integer n we obtain the scalar product

$$n \cdot P = \underbrace{P + \cdots + P}_{n \text{ times}}.$$

In order to efficiently compute the scalar multiplication we use an algorithm similar to square-and-multiply: the Montgomery ladder where the basic operations are differential addition and doubling [Mon87].

We consider x-coordinate-only operations. Throughout the computation, these x-coordinates are kept in projective representation $(X : Z)$, with $x = X/Z$; the point at infinity is represented as $(1 : 0)$. See Section 7.5.1 for more details. We define the operation:

$$\texttt{xDBL\&ADD} : (x_{Q-P}, (X_P : Z_P), (X_Q : Z_Q)) \mapsto \\ ((X_{2\cdot P} : Z_{2\cdot P}), (X_{P+Q} : Z_{P+Q}))$$

A pseudocode description of the Montgomery ladder using this `xDBL&ADD` routine is given in Algorithm 7. The main loop iterates

over the bits of the scalar n. The k^{th} iteration conditionally swaps the arguments P and Q of `xDBL&ADD` depending on the value of the k^{th} bit of n. We use a conditional swap `CSWAP` to change the arguments of the above function while keeping the same body of the loop. Given a pair (P_0, P_1) and a bit b, `CSWAP` returns the pair (P_b, P_{1-b}).

ALGORITHM 7. Montgomery ladder for scalar mult.

```
Input:  x-coordinate x_P of a point P, scalar n with n < 2^m
Output: x-coordinate x_Q of Q = n · P
  Q = (X_Q : Z_Q) ← (1 : 0)
  R = (X_R : Z_R) ← (x_P : 1)
  for k := m down to 1 do
      (Q, R) ← CSWAP((Q, R), k^th bit of n)
      (Q, R) ← xDBL&ADD(x_P, Q, R)
      (Q, R) ← CSWAP((Q, R), k^th bit of n)
  end forreturn X_Q/Z_Q
```

7.2.2 *The X25519 key exchange*

From now on let $\mathbb{F}_p$ be the field with $p = 2^{255} - 19$ elements. We consider the elliptic curve E over $\mathbb{F}_p$ defined by the equation $y^2 = x^3 + 486662x^2 + x$. For every $x \in \mathbb{F}_p$ there exists a point P in $E(\mathbb{F}_{p^2})$ such that x is the x-coordinate of P.

The core of the X25519 key-exchange protocol is a scalar-multiplication function, which we will also refer to as X25519. This function receives as input two arrays of 32 bytes each. One of them is interpreted as the little-endian encoding of a non-negative 256-bit integer n (see Section 7.3). The other is interpreted as the little-endian encoding of the x-coordinate $x_P \in \mathbb{F}_p$ of a point in $E(\mathbb{F}_{p^2})$, using the standard mapping of integers modulo p to elements in $\mathbb{F}_p$.

The X25519 function first computes a scalar n' from n by setting bits at position 0, 1, 2 and 255 to '0'; and at position 254 to '1'. This operation is often called "clamping" of the scalar n. Note that $n' \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\}$. X25519 then computes the x-coordinate of $n' \cdot P$.

RFC 7748 [LHT] standardizes the X25519 Diffie–Hellman key-exchange algorithm. Given the base point B where $X_B = 9$, each party generates a secret random number s_a (respectively s_b), and computes X_{P_a} (respectively X_{P_b}), the x-coordinate of $P_A = s_a \cdot B$ (respectively $P_B = s_b \cdot B$). The parties exchange X_{P_a} and X_{P_b} and compute their shared secret $s_a \cdot s_b \cdot B$ with X25519 on s_a and X_{P_b} (respectively s_b and X_{P_a}).

7.2.3 *TweetNaCl specifics*

As its name suggests, TweetNaCl aims for code compactness (*"a crypto library in 100 tweets"*). As a result it uses a few defines and typedefs to gain precious bytes while still remaining human-readable.

```
#define FOR(i,n) for (i = 0;i < n;++i)
#define sv static void
typedef unsigned char u8;
typedef long long i64;
```

TweetNaCl functions take pointers as arguments. By convention the first one points to the output array `o`. It is then followed by the input arguments.

For a seamless use of VST, indexes used in `for` loops have to be of type `int` instead of `i64`. We changed the code to allow our proofs to carry through. We believe this does not affect the correctness of the original code. A complete diff of our modifications to TweetNaCl can be found in Section 7.A.

7.2.4 *X25519 in TweetNaCl*

We now describe the implementation of X25519 in TweetNaCl.

ARITHMETIC IN $\mathbb{F}_{2^{255}-19}$. In X25519, all computations are performed in $\mathbb{F}_p$. Throughout the computation, elements of that field are represented in radix 2^{16}, i. e., an element A is represented as $(a_0, \ldots, a_{15})$, with $A = \sum_{i=0}^{15} a_i 2^{16i}$. The individual "limbs" a_i are represented as 64-bit `long long` variables:

```
typedef i64 gf[16];
```

The conversion from the input byte array to this representation in radix 2^{16} is done with the `unpack25519` function.

The radix-2^{16} representation in limbs of 64 bits is highly redundant; for any element $A \in \mathbb{F}_{2^{255}-19}$ there are multiple ways to represent A as $(a_0, \ldots, a_{15})$. This is used to avoid or delay carry handling in basic operations such as Addition (`A`), subtraction (`Z`), multiplication (`M`) and squaring (`S`). After a multiplication, limbs of the result `o` are too large to be used again as input. Two calls to `car25519` at the end of `M` takes care of the carry propagation.

Inverses in $\mathbb{F}_{2^{255}-19}$ are computed with `inv25519`. This function uses exponentiation by $p - 2 = 2^{255} - 21$, computed with the square-and-multiply algorithm.

`sel25519` implements a constant-time conditional swap (`CSWAP`) by applying a mask between two fields elements.

Finally, we need the `pack25519` function, which converts from the internal redundant radix-2^{16} representation to a unique byte array representing an integer in $\{0, \ldots, p-1\}$ in little-endian format. This function is considerably more complex as it needs to convert to a

unique representation, i. e., also fully reduce modulo p and remove the redundancy of the radix-2^{16} representation.

The C definitions of those functions are available in Section 7.A.

THE MONTGOMERY LADDER. With these low-level arithmetic and helper functions defined, we can now turn our attention to the core of the X25519 computation: the `crypto_scalarmult` API function of TweetNaCl, which is implemented through the Montgomery ladder.

```
1  int crypto_scalarmult(u8 *q,
2                        const u8 *n,
3                        const u8 *p)
4  {
5    u8 z[32];
6    i64 r;
7    int i;
8    gf x,a,b,c,d,e,f;
9    FOR(i,31) z[i]=n[i];
10   z[31]=(n[31]&127)|64;
11   z[0]&=248;
12   unpack25519(x,p);
13   FOR(i,16) {
14     b[i]=x[i];
15     d[i]=a[i]=c[i]=0;
16   }
17   a[0]=d[0]=1;
18   for(i=254;i>=0;--i) {
19     r=(z[i>>3]>>(i&7))&1;
20     sel25519(a,b,r);
21     sel25519(c,d,r);
22     A(e,a,c);
23     Z(a,a,c);
24     A(c,b,d);
25     Z(b,b,d);
26     S(d,e);
27     S(f,a);
28     M(a,c,a);
29     M(c,b,e);
30     A(e,a,c);
31     Z(a,a,c);
32     S(b,a);
33     Z(c,d,f);
34     M(a,c,_121665);
35     A(a,a,d);
36     M(c,c,a);
37     M(a,d,f);
38     M(d,b,x);
39     S(b,e);
40     sel25519(a,b,r);
41     sel25519(c,d,r);
42   }
43   inv25519(c,c);
44   M(a,a,c);
45   pack25519(q,a);
46   return 0;
47 }
```

Note that lines 10 & 11 represent the "clamping" operation. the sequence of arithmetic operations in lines 22 through 39 implement the `xDBL&ADD` routine. Additionally, on line 34 the constant `_121665` of type `i64` represents the value $121665 = \frac{a-2}{4}$ from curve25519.

7.2.5 *Coq, separation logic, and VST*

Chapter 3 provides a comprehensive introduction to the Coq, the separation logic, and the use of VST. In the following, we provide the reader with a brief summary.

Coq [Coq] is an interactive theorem prover based on type theory. It provides an expressive formal language to write mathematical definitions, algorithms, and theorems together with their proofs. It has been used in the proof of the four-color theorem [Gon08], and it is also the system underlying the CompCert formally verified C compiler [Ler09a]. Unlike systems like F* [Pro+17], Coq does not rely on an SMT solver in its trusted code base. It uses its type system to verify the applications of hypotheses, lemmas, and theorems [How95].

Hoare-Floyd logic is a formal system which allows reasoning about programs. It uses triples such as

$$\{\textbf{Pre}\}\ \texttt{Prog}\ \{\textbf{Post}\}$$

where **Pre** and **Post** are assertions and `Prog` is a fragment of code. It is read as "when the precondition **Pre** is met, executing `Prog` will yield postcondition **Post**". We use compositional rules to prove the truth value of a Hoare triple. For example, here is the rule for sequential composition:

$$\text{Hoare-Seq}\ \frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}}$$

Separation logic is an extension of Hoare logic which allows reasoning about pointers and memory manipulation. Reasoning in separation logic assumes that certain memory regions are non-overlapping. We discuss this limitation further in Section 7.4.1.

The Verifiable Software Toolchain (VST) [Cao+18] is a framework which uses separation logic (proven correct with respect to CompCert semantics) to prove the functional correctness of C programs. The first step consists of translating the source code into Clight, an intermediate representation used by CompCert. For such purpose we use the parser of CompCert called `clightgen`. In a second step one defines the Hoare triple representing the specification of the piece of software one wants to prove. With the help of VST we then use the strongest-postcondition approach to prove the correctness of the triple. This can be seen as a forward symbolic execution of the program.

7.3 FORMALIZING X25519 FROM RFC 7748

In this section we present our formalization of RFC 7748 [LHT].

The specification of X25519 in RFC 7748 is formalized by the function `RFC` *in Coq.*

More specifically, we formalized X25519 with the following definition:

```
Definition RFC (n: list Z) (p: list Z) : list Z :=
  let k := decodeScalar25519 n in
  let u := decodeUCoordinate p in
  let t := montgomery_rec_swap
    255  (* iterate 255 times  *)
    k    (* clamped n          *)
    1    (* x2                 *)
    u    (* x3                 *)
    0    (* z2                 *)
    1    (* z3                 *)
    0    (* dummy              *)
    0    (* dummy              *)
    u    (* x1                 *)
    0    (* previous bit = 0   *) in
  let a := get_a t in
  let c := get_c t in
  let o := ZPack25519 (Z.mul a (ZInv25519 c))
  in encodeUCoordinate o.
```

In this definition `montgomery_rec_swap` is a generic ladder (defined below) over a type `T`. In the case of `RFC` above, we instantiate it over integers.

```
Fixpoint montgomery_rec_swap (m : nat) (z : T')
(a: T) (b: T) (c: T) (d: T) (e: T) (f: T) (x: T) (swap:Z) :
(* a:    x2                 *)
(* b:    x3                 *)
(* c:    z2                 *)
(* d:    z3                 *)
(* e:    temporary  var     *)
(* f:    temporary  var     *)
(* x:    x1                 *)
(* swap: previous bit value *)
(T * T * T * T * T * T) :=
match m with
| S n ⇒
  let r := Getbit (Z.of_nat n) z in
    (* k_t = (k >> t) & 1                              *)
  let swap := Z.lxor swap r in
    (* swap ^= k_t                                     *)
  let (a, b) := (Sel25519 swap a b, Sel25519 swap b a) in
    (* (x2, x3) = cswap(swap, x2, x3)                  *)
  let (c, d) := (Sel25519 swap c d, Sel25519 swap d c) in
    (* (z2, z3) = cswap(swap, z2, z3)                  *)
  let e := a + c in        (* A  = x2 + z2             *)
  let a := a - c in        (* B  = x2 - z2             *)
  let c := b + d in        (* C  = x3 + z3             *)
  let b := b - d in        (* D  = x3 - z3             *)
  let d := e^2    in       (* AA = A^2                 *)
  let f := a^2    in       (* BB = B^2                 *)
  let a := c * a in        (* CB = C * B               *)
  let c := b * e in        (* DA = D * A               *)
  let e := a + c in        (* x3= (DA + CB)^2          *)
  let a := a - c in        (* z3= x1* (DA - CB)^2      *)
  let b := a^2    in       (* z3= x1* (DA - CB)^2      *)
  let c := d - f in        (* E  = AA - BB             *)
  let a := c * C_121665 in (* z2 = E * (AA + a24 * E)  *)
  let a := a + d in        (* z2 = E * (AA + a24 * E)  *)
  let c := c * a in        (* z2 = E * (AA + a24 * E)  *)
  let a := d * f in        (* x2 = AA * BB             *)
```

```
  let d := b * x in          (* z3 = x1* (DA - CB)^2          *)
  let b := e^2     in        (* x3 = (DA + CB)^2              *)
  montgomery_rec_swap n z a b c d e f x r
    (* swap = k_t                                             *)

| 0%nat ⇒
  let (a, b) := (Sel25519 swap a b, Sel25519 swap b a) in
    (* (x2, x3) = cswap(swap, x2, x3)                         *)
  let (c, d) := (Sel25519 swap c d, Sel25519 swap d c) in
    (* (z2, z3) = cswap(swap, z2, z3)                         *)
  (a,b,c,d,e,f)
end.
```

The comments in the ladder represent the text from the RFC which our formalization matches perfectly. In order to optimize the number of calls to `CSWAP` (defined in Section 7.2.1) the RFC uses an additional variable to decide whether a conditional swap is required or not.

Later in our proof we use a simpler description of the ladder (`montgomery_rec`) which follows strictly the Algorithm 7 and prove those ladders equivalent.

RFC 7748 describes the calculations done in X25519 as follows: *"To implement the X25519(k, u) [...] functions (where k is the scalar and u is the u-coordinate), first decode k and u and then perform the following procedure, which is taken from [curve25519] and based on formulas from [montgomery]. All calculations are performed in GF(p), i.e., they are performed modulo p."* [LHT]

Operations used in the Montgomery ladder of or Coq formalization `RFC` are performed on integers (See Section 7.B.2). The reduction modulo $2^{255} - 19$ is deferred to the very end as part of the `ZPack25519` operation.

We now turn our attention to the decoding and encoding of the byte arrays. We define the little-endian projection to integers as follows.

DEFINITION 7.3.1. *Let* `ZofList` : $\mathbb{Z} \to$ `list` $\mathbb{Z} \to \mathbb{Z}$, *a function given n and a list l returns its little-endian decoding with radix* 2^n.

```
Fixpoint ZofList {n:Z} (a:list Z) : Z :=
  match a with
  | [] ⇒ 0
  | h :: q ⇒ h + 2^n * ZofList q
  end.
```

Similarly, we define the projection from integers to little-endian lists.

DEFINITION 7.3.2. *Let* `ListofZ32` : $\mathbb{Z} \to \mathbb{Z} \to$ `list` $\mathbb{Z}$, *given n and a returns a's little-endian encoding as a list with radix* 2^n.

```
Fixpoint ListofZn_fp {n:Z} (a:Z) (f:nat) : list Z :=
match f with
  | 0%nat ⇒ []
  | S fuel ⇒ (a mod 2^n) :: ListofZn_fp (a/2^n) fuel
end.
```

```
Definition ListofZ32 {n:Z} (a:Z) : list Z :=
  ListofZn_fp n a 32.
```

In order to increase the trust in our formalization, we prove that `ListofZ32` and `ZofList` are inverse to each other.

```
Lemma ListofZ32_ZofList_Zlength: forall (l:list Z),
  Forall (fun x ⇒ 0 ≤ x < 2^n) l →
  Zlength l = 32 →
  ListofZ32 n (ZofList n l) = l.
```

Note that the fuel is used to guarantee an output list of 32 elements. This allows us to prove that for all list of 32 bytes, `ListofZn_fp (ZofList L) = L`.

With those tools at hand, we formally define the decoding and encoding as specified in the RFC.

```
Definition decodeScalar25519 (l: list Z) : Z :=
  ZofList 8 (clamp l).

Definition decodeUCoordinate (l: list Z) : Z :=
  ZofList 8 (upd_nth 31 l (Z.land (nth 31 l 0) 127)).

Definition encodeUCoordinate (x: Z) : list Z :=
  ListofZ32 8 x.
```

In the definition of `decodeScalar25519`, `clamp` is taking care of setting and clearing the selected bits as stated in the RFC and described in Section 7.2.2.

7.4 PROVING EQUIVALENCE OF X25519 IN C AND COQ

In this section we outline the structure of our proofs of the following theorem:

The implementation of X25519 in TweetNaCl (`crypto_scalarmult`) matches the specifications of RFC 7748 [LHT] (`RFC`).

More formally:

```
Theorem body_crypto_scalarmult:
  (* VST boiler plate. *)
  semax_body
    (* Global variables used in the code. *)
    Vprog
    (* Hoare triples for function calls. *)
    Gprog
    (* Clight AST of the function we verify. *)
    f_crypto_scalarmult_curve25519_tweet
    (* Our Hoare triple, see below. *)
    crypto_scalarmult_spec.
```

Using our formalization of RFC 7748 (in Section 7.3) we specify the Hoare triple before proving its correctness with VST (in Section 7.4.1). We provide an example of equivalence of operations over different number representations (in Section 7.4.2). Then, we describe efficient techniques used in some of our more complex proofs (in Section 7.4.3).

7.4.1 *Applying the Verifiable Software Toolchain*

We now turn our focus to the formal specification of `crypto_scalarmult`. We use our definition of X25519 from the RFC in the Hoare triple and prove its correctness.

SPECIFICATIONS. We show the soundness of TweetNaCl by proving a correspondence between the C version of TweetNaCl and the same code as a pure Coq function. This defines the equivalence between the Clight representation and our Coq definition of the ladder (`RFC`).

```
Definition crypto_scalarmult_spec :=
DECLARE _crypto_scalarmult_curve25519_tweet
WITH
  v_q: val, v_n: val, v_p: val, c121665:val,
  sh : share,
  q : list val, n : list Z, p : list Z
(*-----------------------------------------*)
PRE [ _q OF (tptr tuchar),
     _n OF (tptr tuchar),
     _p OF (tptr tuchar) ]
PROP (writable_share sh;
      Forall (λ x ↦ 0 ≤ x < 2^8) p;
      Forall (λ x ↦ 0 ≤ x < 2^8) n;
      Zlength q = 32; Zlength n = 32; Zlength p = 32)
LOCAL(temp _q v_q; temp _n v_n; temp _p v_p;
      gvar __121665 c121665)
SEP  (sh⟦ v_q ⟧←(uch32)– q;
      sh⟦ v_n ⟧←(uch32)– mVI n;
      sh⟦ v_p ⟧←(uch32)– mVI p;
      Ews⟦ c121665 ⟧←(lg16)– mVI64 c_121665)
(*-----------------------------------------*)
POST [ tint ]
PROP (Forall (λ x ↦ 0 ≤ x < 2^8) (RFC n p);
      Zlength (RFC n p) = 32)
LOCAL(temp ret_temp (Vint Int.zero))
SEP  (sh⟦ v_q ⟧←(uch32)– mVI (RFC n p);
      sh⟦ v_n ⟧←(uch32)– mVI n;
      sh⟦ v_p ⟧←(uch32)– mVI p;
      Ews⟦ c121665 ⟧←(lg16)– mVI64 c_121665)
```

In this specification we state preconditions like:

`PRE`: `_p OF (tptr tuchar)`
The function `crypto_scalarmult` takes as input three pointers to arrays of unsigned bytes (`tptr tuchar`) `_p`, `_q` and `_n`.

`LOCAL`: `temp _p v_p`
Each pointer represents an address `v_p`, `v_q` and `v_n`.

`SEP`: `sh⟦ v_p ⟧←(uch32)– mVI p`
In the memory share `sh`, the address `v_p` points to a list of integer values `mVI p`.

`Ews⟦ c121665 ⟧←(lg16)– mVI64 c_121665`
In the global memory share `Ews`, the address `c121665` points to a list of 16 64-bit integer values corresponding to $\frac{a-2}{4} = 121665$.

`PROP`: `Forall (fun x ↦ 0 ≤ x < 2^8) p`
In order to consider all the possible inputs, we assume each element of the list `p` to be bounded by 0 included and 2^8 excluded.

`PROP`: `Zlength p = 32`
We also assume that the length of the list `p` is 32. This defines the complete representation of `u8[32]`.

As postcondition we have conditions like:

`POST`: `tint`
The function `crypto_scalarmult` returns an integer.
`LOCAL`: `temp ret_temp (Vint Int.zero)`
The returned integer has value 0.
`SEP`: `sh ⟦ v_q ⟧⟷(uch32)⊢ mVI (RFC n p)`
In the memory share `sh`, the address `v_q` points to a list of integer values `mVI (RFC n p)` where `RFC n p` is the result of the `crypto_scalarmult` of `n` and `p`.
`PROP`: `Forall (fun x ↦ 0 ≤ x < 2^8) (RFC n p)`
`PROP`: `Zlength (RFC n p) = 32`
We show that the computation for `RFC` fits in `u8[32]`.

`crypto_scalarmult` computes the same result as `RFC` in Coq provided that inputs are within their respective bounds: arrays of 32 bytes.

The correctness of this specification is formally proven in Coq as `Theorem body_crypto_scalarmult`.

MEMORY ALIASING. The semicolon in the `SEP` parts of the Hoare triples represents the *separating conjunction* (often written as a star), which means that the memory shares of `q`, `n` and `p` do not overlap. In other words, we only prove correctness of `crypto_scalarmult` when it is called without aliasing. But for other TweetNaCl functions, like the multiplication function `M(o,a,b)`, we cannot ignore aliasing, as it is called in the ladder in an aliased manner.

In VST, a simple specification of this function will assume that the pointer arguments point to non-overlapping space in memory. When called with three memory shares (`o, a, b`), the three of them will be *consumed*. However assuming this naive specification when `M(o,a,a)` is called (squaring), the first two memory shares (`o, a`) are *consumed* and VST will expect a third memory shares (`a`) which does not *exist* anymore. Examples of such cases are illustrated in Figure 7.4.

As a result, a function must either have multiple specifications or specify which aliasing case is being used. The first option would require us to do very similar proofs multiple times for a same function. We chose the second approach: for functions with 3 arguments, named hereafter `o, a, b`, we define an additional parameter k with values in $\{0,1,2,3\}$:

- if $k = 0$ then `o` and `a` are aliased,
- if $k = 1$ then `o` and `b` are aliased,
- if $k = 2$ then `a` and `b` are aliased,
- else there is no aliasing.

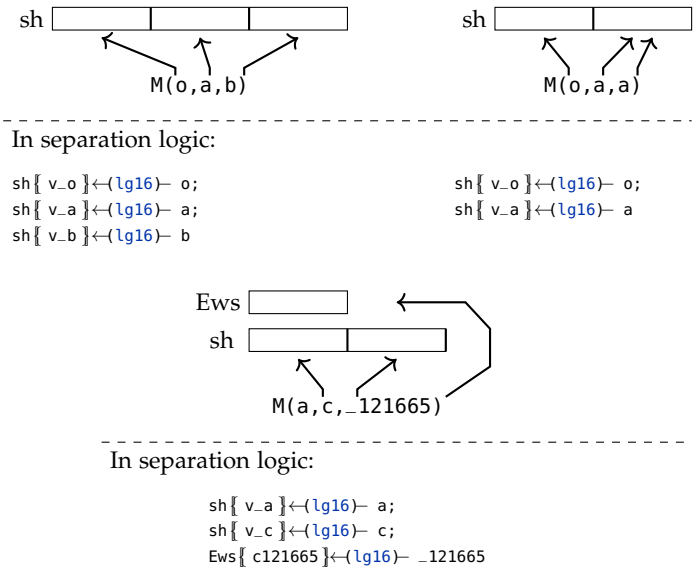


Figure 7.4: Aliasing and separation logic.

In the proof of our specification, we do a case analysis over k when needed. This solution does not cover all the possible cases of aliasing over 3 pointers (e. g., `o = a = b`) but it is enough to satisfy our needs.

PARTIAL OR TOTAL CORRECTNESS. An astute reader would argue that our proof is not "totally correct", not in the sense that our proof is wrong, but referring to the notion of *total correctness*. The difference between *partial* and *total correctness* comes down to the proof of termination, in other words that there is no infinite loop in the code.

In their paper, the authors of VST note that:

> *"Our tools implement a Hoare logic of partial correctness, meaning that C may loop infinitely."* [Cao+18].

Consequently, our proof of `Theorem body_crypto_scalarmult` does not cover the total correctness. This could be solved by (1) extending the WHILE RULE to enforce a decreasing argument or (2) by incorporating it in the pre/post conditions, and loop invariant.

On a simpler level, a weaker argument for the total correctness of our proof is that there are no `while` loops used, and the `for` loops in `crypto_scalarmult` and all subsequent functions are with decreasing arguments, finishing in a finite number of steps.

7.4.2 *Number representation and C implementation*

As described in Section 7.2.3, numbers in `gf` (array of 16 `long long`) are represented in base 2^{16} and we use a direct mapping to represent that array as a list integers in Coq. However, in order to show the correctness of the basic operations, we need to convert this number to an integer. We reuse the mapping `ZofList` : $\mathbb{Z} \to$ `list` $\mathbb{Z} \to \mathbb{Z}$ from Section 7.3 and define a notation where n is 16, to fix a radix of 2^{16}.

```
Notation "ℤ16.lst A" := (ZofList 16 A).
```

To facilitate working in $\mathbb{Z}_{2^{255}-19}$, we define the $:GF$ notation.

```
Notation "A :GF" := (A mod (2^255 - 19)).
```

Later in Section 7.5.2, we formally define $\mathbb{F}_{2^{255}-19}$ as a field. Equivalence between operations in $\mathbb{Z}_{2^{255}-19}$ (i. e., under $:GF$) and in $\mathbb{F}_{2^{255}-19}$ are easily proven.

We define `Low.M` to replicate the computations and steps done the multiplication over `gf` in C; similarly we also define `Low.A`, `Low.Sq`, `Low.Zub` etc. respectively to cover the other operations. Then using the two definition above-mentioned, we prove intermediate lemmas such as Lemma 7.4.1 for the correctness of the multiplication.

LEMMA 7.4.1. *`Low.M` correctly implements the multiplication over* $\mathbb{Z}_{2^{255}-19}$.

Lemma 7.4.1 is proven in Coq as follows:

```
Lemma mult_GF_Zlength :
  forall (a:list Z) (b:list Z),
  Zlength a = 16 →
  Zlength b = 16 →
   (ℤ16.lst (Low.M a b)) :GF =
   (ℤ16.lst a * ℤ16.lst b) :GF.
```

However, for our purpose simple functional correctness is not enough. We also need to define the bounds under which the operations are correct, allowing us to chain them, and guaranteeing us the absence of overflow.

LEMMA 7.4.2. *if all the values of the input arrays are constrained between* -2^{26} *and* 2^{26}*, then the output array of `Low.M` will have its values constrained between* -38 *and* $2^{16} + 38$.

Lemma 7.4.2 is proven in Coq as follows:

```
Lemma M_bound_Zlength :
  forall (a:list Z) (b:list Z),
  Zlength a = 16 →
  Zlength b = 16 →
  Forall (fun x ⇒  -2^26 < x < 2^26) a →
  Forall (fun x ⇒  -2^26 < x < 2^26) b →
    Forall (fun x ⇒  -38 ≤  x < 2^16 + 38) (Low.M a b).
```

By using each function `Low.M`; `Low.A`; `Low.Sq`; `Low.Zub`; `Unpack25519`; `clamp`; `Pack25519`; `Inv25519`; `car25519`; `montgomery_rec`, we defined `Crypto_Scalarmult` in Coq and with VST proved that it matches the exact behavior of X25519 in TweetNaCl.

By proving that each function `Low.M`; `Low.A`; `Low.Sq`; `Low.Zub`; `Unpack25519`; `clamp`; `Pack25519`; `Inv25519`; `car25519` behave over `list Z` as their equivalent over `Z` with `:GF` (in $\mathbb{Z}_{2^{255}-19}$), we prove that given the same inputs `Crypto_Scalarmult` performs the same computation as `RFC`.

```
Lemma Crypto_Scalarmult_RFC_eq :
  forall (n: list Z) (p: list Z),
  Zlength n = 32 →
  Zlength p = 32 →
  Forall (fun x ⇒ 0 ≤ x ∧ x < 2 ^ 8) n →
  Forall (fun x ⇒ 0 ≤ x ∧ x < 2 ^ 8) p →
  Crypto_Scalarmult n p = RFC n p.
```

Using this equality, we can directly replace `Crypto_Scalarmult` in our specification by `RFC`, proving that TweetNaCl's X25519 implementation respects RFC 7748.

7.4.3 *Towards faster proofs*

The idea of speed in verification is quite ambiguous as it may refer to: (1) the performances of proof automation on large terms (generating), (2) similarly the performances of verifying the proofs on large terms (checking), (3) strategies to formalize and prove the correctness of some functions. The idea that a proof is executed *only once* is an easy misconception that one can have about formal verification: *"We have proven the correctness of* foo*, so we do not need to do it again."* This is easily disproved by considering that any modification of the dependencies requires the checking of all the subsequent proofs. As a result, this recompilation may take from minutes to an hour and hinder the efforts of a researcher (see Figure 7.5).

Figure 7.5: Compiling. — `https://xkcd.com/303/`

Appel summarized the answer to this questions on the Coq-club mailing list: *"Question: how to improve Coq's performance on proof automation for large terms. Short answer: reflection. Long answer: reflection, reflection, reflection. And perhaps uconstr."* [App21]

This *reflection* technique mentioned by Appel is most notably known for being the base of the Ssreflect library [GMT16] and is discussed at length in chapter 15 of [Chl10].

In the rest of this section we provide two examples where we applied reflection to illustrate briefly how this technique works and some of its use cases.

INVERSE IN $\mathbb{Z}_{2^{255}-19}$. In TweetNaCl, `inv25519` computes an inverse in $\mathbb{Z}_{2^{255}-19}$. It uses Fermat's little theorem by raising to the power of $p-2 = 2^{255}-21$ with a square-and-multiply algorithm. The binary representation of $2^{255}-21$ implies that every step does a multiplication except for bits 2 and 4 (see Code 7.1).

```
1  sv inv25519(gf o,const gf i)
2  {
3    gf c;
4    int a;
5    set25519(c,i);
6    for(a=253;a>=0;a--) {
7      S(c,c);
8      if(a!=2&&a!=4) M(c,c,i);
9    }
10   FOR(a,16) o[a]=c[a];
11 }
```

Code 7.1: Inverse modulo $2^{255}-19$ in TweetNaCl

To prove the correctness of the result we could use multiple strategies such as:

- Proving it is a special case of the square-and-multiply algorithm applied to $2^{255}-21$.
- Unrolling the for loop step-by-step and applying the equalities $x^a \times x^b = x^{(a+b)}$ and $(x^a)^2 = x^{(2\times a)}$, and subsequently proving that the resulting exponent is $2^{255}-21$.

We use the second method because it is simpler. However, it requires us to apply the unrolling and exponentiation formulas 255 times. This could be automated in Coq with tacticals such as `repeat`, but it generates a large proof object which will take a long time to verify, i. e., type-check the proof term.

That is where in order to speed up the process we use the *reflection* technique. It provides us with flexibility, e. g., we don't need to know the number of times nor the order in which the lemmas need to be applied.

The idea is to *reflect* the goal into a decidable environment. We show that for a proposition P, we can define a decidable Boolean property p such that $[\![p]\!] \equiv_{env} P$, in other words that p is denotes P in the

environment *env*. Additionally, by applying a decision procedure on p we infer that P holds.

$$sound\_P : \forall\ env,\ \texttt{decide}\ p = true \rightarrow [\![p]\!]_{env}$$

By applying *sound_P* on P our goal becomes `decide` $p = true$. After simple computation, if everything goes well we are left with the tautology $true = true$.

The reflection used in the proof of `inv25519` is described in further details in Section 7.D.

PACKING IN $\mathbb{Z}_{2^{255}-19}$. This reflection technique is also used where proofs require some computing over a small and finite domain of variables to test e. g., for all i such that $0 \leq i < 16$. Using reflection we prove that we can split the `for` loop in `pack25519` (Code 7.2) into two separate loops.

```
1  for(i=1;i<15;i++) {
2    m[i]=t[i]-0xffff-((m[i-1]>>16)&1);
3    m[i-1]&=0xffff;
4  }
```

Code 7.2: Subtract-and-carry.

The first loop computes the subtraction, and the second applies the carries. Effectively making the code equivalent to Code 7.3.

```
1  for(i=1;i<15;i++) {
2    m[i]=t[i]-0xffff
3  }
4
5  for(i=1;i<15;i++) {
6    m[i]=m[i]-((m[i-1]>>16)&1);
7    m[i-1]&=0xffff;
8  }
```

Code 7.3: Subtract then carry.

This is done by creating a domain-specific language which captures the possible operations applied in Code 7.2 and 7.3:

```
Inductive red_expr :=
  | R_red : term → red_expr
  | Sub_red : red_expr → red_expr
  | SubC_red : red_expr → red_expr → red_expr
  | Mod_red : red_expr → red_expr
  | Nth_red : nat → list red_expr → red_expr → red_expr.
```

Therefore, the result of the `for` loop is seen as a list of `red_expr` where:

- `R_red` denotes base expressions,
- `Sub_red` denotes `x-0xffff`,
- `SubC_red` denotes `x-((y>>16)&1)`,
- `Mod_red` denotes `x&0xffff`,
- `Nth_red` denotes `m[i-1]`.

Consequently, `for` loops in Code 7.2 and Code 7.3 are turned into Coq functions which apply the respective transformations on an input list of `red_expr`. The equality between the resulting list translate

directly into the equality of the loop as the chain of operations is the same.

This `for`-loop separation allows for simpler proofs: the first loop is seen as the subtraction of $2^{255} - 19$ and the resulting number represented in $\mathbb{Z}_{2^{255}-19}$ is invariant with the iteration of the second loop. Down the line, this helps us prove that `pack25519` reduces modulo $2^{255} - 19$ and returns a number in base 2^8.

```
Lemma Pack25519_mod_25519 :
  forall (l:list Z),
  Zlength l = 16 →
  Forall (λ x ⇒  -2^62 < x < 2^62) l →
  ZofList 8 (Pack25519 l) = (ℤ16.lst l) mod (2^255 − 19).
```

7.5 PROVING THAT X25519 MATCHES THE MATHEMATICAL MODEL

In this section we prove the following informal theorem:

The implementation of X25519 in TweetNaCl computes the $\mathbb{F}_p$-restricted x-coordinate scalar multiplication on $E(\mathbb{F}_{p^2})$ where p is $2^{255} - 19$ and E is the elliptic curve $y^2 = x^3 + 486662x^2 + x$.

More precisely, we prove that our formalization of the RFC matches the definitions of Curve25519 by Bernstein:

```
Theorem RFC_Correct: forall (n p : list Z)
  (P:mc curve25519_Fp2_mcuType),
  Zlength n = 32 →
  Zlength p = 32 →
  Forall (fun x ⇒  0 ≤  x ∧  x < 2 ^ 8) n →
  Forall (fun x ⇒  0 ≤  x ∧  x < 2 ^ 8) p →
  Fp2_x (decodeUCoordinate p) = P#x0 →
  RFC n p =
    encodeUCoordinate
      ((P *+ (Z.to_nat (decodeScalar25519 n))) _x0).
```

In Section 7.5.1, we first review the work of Bartzia and Strub [BS14], then we extend it to support Montgomery curves with homogeneous coordinates, and we prove the correctness of the ladder. In Section 7.5.2, we discuss the twist of Curve25519, and we explain how we deal with it in the proofs.

7.5.1 *Formalization of elliptic Curves*

Figure 7.6 presents the structure of the proof of the ladder's correctness. The white tiles are definitions, the orange one is a hypothesis and the green tiles represent major lemmas and theorems.

We consider the field $\mathbb{K}$ and formalize the Montgomery curves $(M_{a,b}(\mathbb{K}))$. Then, by using the equivalent Weierstraß form $(E_{a',b'}(\mathbb{K}))$ from the library of Bartzia and Strub, we prove that $M_{a,b}(\mathbb{K})$ forms an abelian group. Under the hypotheses that $a^2 - 4$ is not a square in $\mathbb{K}$, we prove the correctness of the ladder (Theorem 7.5.9).

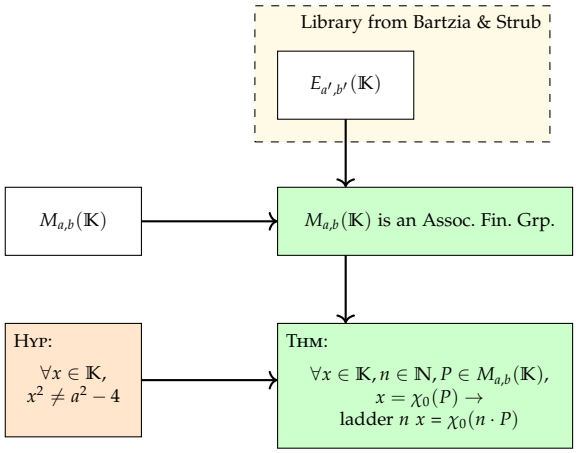


Figure 7.6: Overview of the proof of Montgomery ladder's correctness.

We now turn our attention to the details of the proof of the ladder's correctness.

DEFINITION 7.5.1. *Given a field $\mathbb{K}$, using an appropriate choice of coordinates, an elliptic curve E is a plane cubic algebraic curve defined by an equation $E(x, y)$ of the form:*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where the a_i's are in $\mathbb{K}$ and the curve has no singular point (i. e., no cusps or self-intersections). The set of points defined over $\mathbb{K}$, written $E(\mathbb{K})$, is formed by the solutions (x, y) of E together with a distinguished point $\mathcal{O}$ called point at infinity:

$$E(\mathbb{K}) = \{(x, y) \in \mathbb{K} \times \mathbb{K} \mid E(x, y)\} \cup \{\mathcal{O}\}$$

SHORT WEIERSTRASS CURVES.

For the remainder of this chapter, we assume that the characteristic of $\mathbb{K}$ is neither 2 nor 3. Then, this equation $E(x, y)$ can be reduced into its short Weierstraß form.

DEFINITION 7.5.2. *Let $a \in \mathbb{K}$ and $b \in \mathbb{K}$ such that*

$$\Delta(a, b) = -16(4a^3 + 27b^2) \neq 0.$$

The elliptic curve $E_{a,b}$ is defined by the equation:

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

$E_{a,b}(\mathbb{K})$ is the set of all points $(x, y) \in \mathbb{K}^2$ satisfying the $E_{a,b}$ along with an additional formal point $\mathcal{O}$, "at infinity". Such a curve does not have any singularity.

In this setting, Bartzia and Strub defined the parametric type `ec` which represent the points on a specific curve. It is parameterized by a `K : ecuFieldType` — the type of fields which characteristic is not 2 or 3 — and `E : ecuType` — a record that packs the curve parameters a and b along with the proof that $\Delta(a,b) \neq 0$.

```
Inductive point := EC_Inf | EC_In of K * K.
Notation "(| x, y |)" := (EC_In x y).
Notation "∞" := (EC_Inf).

Record ecuType :=
  { A : K; B : K; _ : 4 * A^3 + 27 * B^2 ≠ 0}.
Definition oncurve (p : point) :=
  if p is (| x, y |)
    then y^2 == x^3 + A * x + B
    else true.
Inductive ec : Type := EC p of oncurve p.
```

Points on an elliptic curve are equipped with the structure of an abelian group.

- The negation of a point $P = (x, y)$ is defined by reflection over the x axis $-P = (x, -y)$.
- The addition of two points P and Q is defined as the negation of the third intersection point of the line passing through P and Q, or tangent to P if $P = Q$.
- $\mathcal{O}$ is the neutral element under this law: if 3 points are collinear, their sum is equal to $\mathcal{O}$.

These operations are defined in Coq as follows (where we omit the code for the tangent case):

```
Definition neg (p : point K) :=
  if p is (| x, y |) then (| x, -y |) else EC_Inf.

Definition add (p1 p2 : point) :=
  match p1, p2 with
    | ∞ , _ ⇒ p2
    | _ , ∞ ⇒ p1
    | (| x1, y1 |), (| x2, y2 |) ⇒
      if x1 == x2 then ... else
        let s := (y2 - y1) / (x2 - x1) in
        let xs := s^2 - x1 - x2 in
          (| xs, - s * (xs - x1 ) - y1 |)
  end.
```

The value of `add` is proven to be on the curve (with coercion):

```
Lemma add0 (p q : ec): oncurve (add p q).

Definition addec (p1 p2 : ec) : ec :=
  EC (add p1 p2) (add0 p1 p2)
```

MONTGOMERY CURVES.

Speedups can be obtained by switching to homogeneous coordinates and other forms than the Weierstraß form. We consider the Montgomery form [Mon87].

DEFINITION 7.5.3. *Let $a \in \mathbb{K}\backslash\{-2,2\}$, and $b \in \mathbb{K}\backslash\{0\}$. The elliptic curve $M_{a,b}$ is defined by the equation:*

$$by^2 = x^3 + ax^2 + x,$$

$M_{a,b}(\mathbb{K})$ is the set of all points $(x,y) \in \mathbb{K}^2$ satisfying the $M_{a,b}$ along with an additional formal point $\mathcal{O}$, "at infinity".

Similar to the definition of `ec`, we defined the parametric type `mc` which represents the points on a specific Montgomery curve. It is parameterized by a `K : ecuFieldType` — the type of fields whose characteristic is neither 2 nor 3— and `M : mcuType` — a record that packs the curve parameters a and b along with the proofs that $b \neq 0$ and $a^2 \neq 4$.

```
Record mcuType :=
  { cA : K; cB : K; _ : cB ≠ 0; _ : cA^2 ≠ 4}.
Definition oncurve (p : point) :=
if p is (| x, y |)
  then cB * y^2 == x^3 + cA * x^2 + x
  else true.
Inductive mc : Type := MC p of oncurve p.

Lemma oncurve_mc: forall p : mc, oncurve p.
```

We define the addition on Montgomery curves in a similar way as for the Weierstraß form.

```
Definition add (p1 p2 : point) :=
  match p1, p2 with
    | ∞, _ ⇒ p2
    | _, ∞ ⇒ p1
    | (|x1, y1|), (|x2, y2|) ⇒
      if   x1 == x2
      then if  (y1 == y2) && (y1 ≠ 0)
           then ... else ∞
      else
        let s  := (y2 - y1) / (x2 - x1) in
        let xs := s^2 * cB - cA - x1 - x2 in
          (| xs, - s * (xs - x1) - y1 |)
    end.
```

And again we prove the result is on the curve:

```
Lemma add0 (p q : mc) : oncurve (add p q).

Definition addmc (p1 p2 : mc) : mc :=
  MC (add p1 p2) (add0 p1 p2)
```

We define a bijection between a Montgomery curve and its short Weierstraß form (Lemma 7.5.4) and prove that it respects the addition

as defined on the respective curves. In this way we get all the group laws for Montgomery curves from the Weierstraß ones.

After having verified the group properties, it follows that the bijection is a group isomorphism.

LEMMA 7.5.4. *Let $M_{a,b}$ be a Montgomery curve, define*

$$a' = \frac{3 - a^2}{3b^2} \quad \text{and} \quad b' = \frac{2a^3 - 9a}{27b^3},$$

then $E_{a',b'}$ is a Weierstraß curve, and the mapping $\varphi : M_{a,b} \mapsto E_{a',b'}$ defined as:

$$\varphi(\mathcal{O}_M) = \mathcal{O}_E$$
$$\varphi((x,y)) = \left(\frac{x}{b} + \frac{a}{3b}, \frac{y}{b}\right)$$

is a group isomorphism between elliptic curves.

```
Definition ec_of_mc_point p :=
  match p with
  | ∞ ⇒ ∞
  | (|x, y|) ⇒ (|x/b + a/(3 * b), y/b|)
  end.
Lemma ec_of_mc_point_ok p :
  oncurve M p →
  ec.oncurve E (ec_of_mc_point p).

Definition ec_of_mc p :=
  EC (ec_of_mc_point_ok (oncurve_mc p)).

Lemma ec_of_mc_bij : bijective ec_of_mc.
```

PROJECTIVE COORDINATES.

In a projective plane, points are represented with triples $(X : Y : Z)$, with the exception of $(0 : 0 : 0)$. Scalar multiples are representing the same point, i. e., for all $\lambda \neq 0$, the triples $(X : Y : Z)$ and $(\lambda X : \lambda Y : \lambda Z)$ represent the same point. For $Z \neq 0$, the projective point $(X : Y : Z)$ corresponds to the point $(X/Z, Y/Z)$ on the affine plane. Likewise, the point (X, Y) on the affine plane corresponds to $(X : Y : 1)$ on the projective plane.

Using fractions as coordinates, the equation for a Montgomery curve $M_{a,b}$ becomes:

$$b\left(\frac{Y}{Z}\right)^2 = \left(\frac{X}{Z}\right)^3 + a\left(\frac{X}{Z}\right)^2 + \left(\frac{X}{Z}\right)$$

Multiplying both sides by Z^3 yields:

$$bY^2Z = X^3 + aX^2Z + XZ^2$$

Setting $Z = 0$ in this equation, we derive $X = 0$. Hence, $(0 : 1 : 0)$ is the unique point on the curve at infinity.

By restricting the parameter a of $M_{a,b}(\mathbb{K})$ such that $a^2 - 4$ is not a square in $\mathbb{K}$ (Hypothesis 7.5.5), we ensure that $(0,0)$ is the only point with a y-coordinate of 0.

HYPOTHESIS 7.5.5. *$a^2 - 4$ is not a square in $\mathbb{K}$.*

```
Hypothesis mcu_no_square : forall x : K, x^2 ≠ a^2 - 4.
```

We define χ and χ_0 to return the x-coordinate of points on a curve.

DEFINITION 7.5.6. *Let $\chi : M_{a,b}(\mathbb{K}) \mapsto \mathbb{K} \cup \{\infty\}$ and $\chi_0 : M_{a,b}(\mathbb{K}) \mapsto \mathbb{K}$ such that*

$$\chi((x,y)) = x, \qquad \chi(\mathcal{O}) = \infty, \qquad \text{and}$$
$$\chi_0((x,y)) = x, \qquad \chi_0(\mathcal{O}) = 0.$$

Using projective coordinates we prove the formula for differential addition.

LEMMA 7.5.7. *Let $M_{a,b}$ be a Montgomery curve such that $a^2 - 4$ is not a square in $\mathbb{K}$, and let $X_1, Z_1, X_2, Z_2, X_4, Z_4 \in \mathbb{K}$, such that $(X_1, Z_1) \neq (0,0)$, $(X_2, Z_2) \neq (0,0)$, $X_4 \neq 0$ and $Z_4 \neq 0$. Define*

$$X_3 = Z_4((X_1 - Z_1)(X_2 + Z_2) + (X_1 + Z_1)(X_2 - Z_2))^2$$
$$Z_3 = X_4((X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2))^2,$$

then for any point P_1 and P_2 in $M_{a,b}(\mathbb{K})$ such that $X_1/Z_1 = \chi(P_1)$, $X_2/Z_2 = \chi(P_2)$, and $X_4/Z_4 = \chi(P_1 - P_2)$, we have $X_3/Z_3 = \chi(P_1 + P_2)$.
REMARK: *These definitions should be understood in $\mathbb{K} \cup \{\infty\}$. If $x \neq 0$ then we define $x/0 = \infty$.*

Similarly we also prove the formula for point doubling.

LEMMA 7.5.8. *Let $M_{a,b}$ be a Montgomery curve such that $a^2 - 4$ is not a square in $\mathbb{K}$, and let $X_1, Z_1 \in \mathbb{K}$, such that $(X_1, Z_1) \neq (0,0)$. Define*

$$c = (X_1 + Z_1)^2 - (X_1 - Z_1)^2$$
$$X_3 = (X_1 + Z_1)^2 (X_1 - Z_1)^2$$
$$Z_3 = c\Big((X_1 + Z_1)^2 + \frac{a-2}{4} \times c\Big),$$

then for any point P_1 in $M_{a,b}(\mathbb{K})$ such that $X_1/Z_1 = \chi(P_1)$, we have $X_3/Z_3 = \chi(2P_1)$.

With Lemma 7.5.7 and Lemma 7.5.8, we are able to compute efficiently differential addition and point doubling using projective coordinates.

SCALAR MULTIPLICATION ALGORITHMS.

By taking Algorithm 7 and replacing `xDBL&ADD` by a combination of the formulas from Lemma 7.5.7 and Lemma 7.5.8,, we define a ladder `opt_montgomery` (in which $\mathbb{K}$ has not been fixed yet). This gives us the theorem of the correctness of the Montgomery ladder.

THEOREM 7.5.9. *For all $n, m \in \mathbb{N}$, $x \in \mathbb{K}$, $P \in M_{a,b}(\mathbb{K})$, if $\chi_0(P) = x$ then `opt_montgomery` returns $\chi_0(n \cdot P)$*

```
Theorem opt_montgomery_ok (n m: nat) (x : K) :
  n < 2^m →
  forall (p : mc M), p#x0 = x →
  opt_montgomery n m x = (p *+ n)#x0.
```

The definition of `opt_montgomery` is similar to `montgomery_rec_swap` that was used in `RFC`. We proved their equivalence, and used it in our final proof of `Theorem RFC_Correct`.

7.5.2 *Curves, twists and extension fields*

Figure 7.7 gives a high-level view of the proofs presented here. The white tiles are definitions while green tiles are important lemmas and theorems.

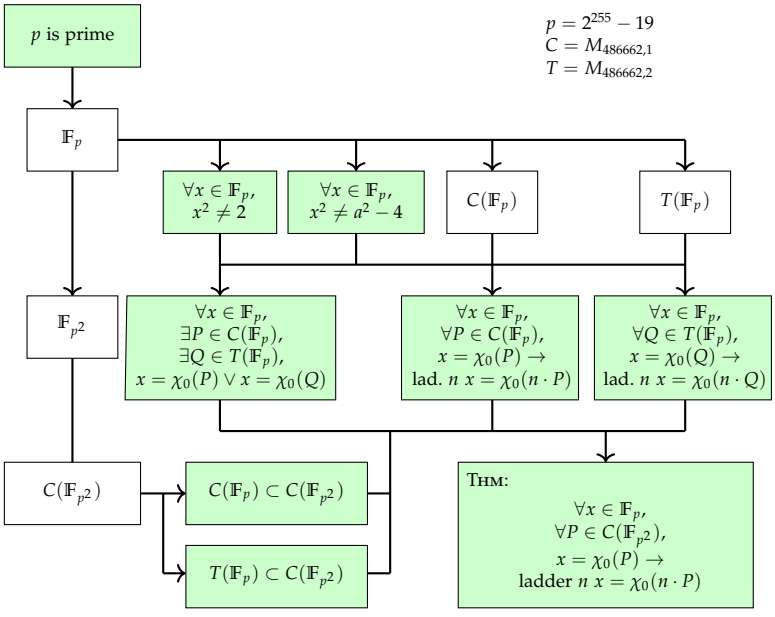


Figure 7.7: Proof dependencies for the correctness of X25519.

A brief overview of the complete proof is described below. We first set $a = 486662$, $b = 1$, $b' = 2$, $p = 2^{255} - 19$, with the equations $C = M_{a,b}$, and $T = M_{a,b'}$. We prove the primality of p and define the field $\mathbb{F}_p$. Subsequently, we show that neither 2 nor $a^2 - 2$ is a square in $\mathbb{F}_p$. We consider $\mathbb{F}_{p^2}$ and define $C(\mathbb{F}_p)$, $T(\mathbb{F}_p)$, and $C(\mathbb{F}_{p^2})$. We prove that for all $x \in \mathbb{F}_p$ there exists a point with x-coordinate x either on $C(\mathbb{F}_p)$ or on the quadratic twist $T(\mathbb{F}_p)$. We prove that all points in $M(\mathbb{F}_p)$ and $T(\mathbb{F}_p)$ can be projected in $M(\mathbb{F}_{p^2})$ and derive that computations done in $M(\mathbb{F}_p)$ and $T(\mathbb{F}_p)$ yield the same results if projected to $M(\mathbb{F}_{p^2})$. Using Theorem 7.5.9 we prove that the ladder is correct for $M(\mathbb{F}_p)$ and $T(\mathbb{F}_p)$; with the previous results, this results in the correctness of the ladder for $M(\mathbb{F}_{p^2})$, in other words the correctness of X25519.

Now that we have an aperçu of the proof, we turn our attention to the details. Indeed, Theorem 7.5.9 cannot be applied directly to prove that X25519 is doing the computations over $M(\mathbb{F}_{p^2})$. This would infer that $\mathbb{K} = \mathbb{F}_{p^2}$, and we would need to satisfy Hypothesis 7.5.5:

$$\forall x \in \mathbb{K},\ x^2 \neq a^2 - 4,$$

which is not possible as there always exists $x \in \mathbb{F}_{p^2}$ such that $x^2 = a^2 - 4$. Consequently, we first study Curve25519 and one of its quadratic twists Twist25519, both defined over $\mathbb{F}_p$.

CURVES AND TWISTS.

We define $\mathbb{F}_p$ as the numbers between 0 and $p = 2^{255} - 19$. We create a `Zmodp` module to encapsulate those definitions.

```
Module Zmodp.
Definition betweenb x y z := (x ≤? z) && (z <? y).
Definition p := locked (2^255 - 19).
Fact Hp_gt0 : p > 0.
Inductive type := Zmodp x of betweenb 0 p x.

Lemma Z_mod_betweenb (x y : Z) :
  y > 0 → betweenb 0 y (x mod y).
Definition pi (x : Z) : type :=
  Zmodp (Z_mod_betweenb x Hp_gt0).
Coercion repr (x : type) : Z :=
  let: @Zmodp x _ := x in x.
```

We define the basic operations $(+, -, \times)$ with their respective neutral elements $(0, 1)$ and prove Lemma 7.5.10.

LEMMA 7.5.10. *$\mathbb{F}_p$ is a field.*

For $a = 486662$, by using the Legendre symbol we prove that $a^2 - 4$ and 2 are not squares in $\mathbb{F}_p$.

```
Fact a_not_square : forall x: Zmodp.type,
  x^2 ≠ (Zmodp.pi 486662)^2 - 4.
```

```
Fact two_not_square : forall x: Zmodp.type,
  x^2 ≠ 2.
```

This allows us to study $M_{486662,1}(\mathbb{F}_p)$ and $M_{486662,2}(\mathbb{F}_p)$, one of its quadratic twists.

DEFINITION 7.5.11. *We instantiate* `opt_montgomery` *in two specific ways:*

- $Curve25519\_Fp(n, x)$ *for* $M_{486662,1}(\mathbb{F}_p)$.
- $Twist25519\_Fp(n, x)$ *for* $M_{486662,2}(\mathbb{F}_p)$.

With Theorem 7.5.9 we derive the following two lemmas:

LEMMA 7.5.12. *Let* $x \in \mathbb{F}_p$, $n \in \mathbb{N}$, $P \in \mathbb{F}_p \times \mathbb{F}_p$ *such that* $P \in M_{486662,1}(\mathbb{F}_p)$ *and* $\chi_0(P) = x$, *then*

$$Curve25519\_Fp(n, x) = \chi_0(n \cdot P).$$

LEMMA 7.5.13. *Let* $x \in \mathbb{F}_p$, $n \in \mathbb{N}$, $P \in \mathbb{F}_p \times \mathbb{F}_p$ *such that* $P \in M_{486662,2}(\mathbb{F}_p)$ *and* $\chi_0(P) = x$, *then*

$$Twist25519\_Fp(n, x) = \chi_0(n \cdot P).$$

As the Montgomery ladder does not depend on b, it is trivial to see that the computations done for points in $M_{486662,1}(\mathbb{F}_p)$ and in $M_{486662,2}(\mathbb{F}_p)$ are the same.

```
Theorem curve_twist_eq: forall n x,
  curve25519_Fp_ladder n x = twist25519_Fp_ladder n x.
```

Because 2 is not a square in $\mathbb{F}_p$, it allows us split $\mathbb{F}_p$ into two sets.

LEMMA 7.5.14. *For all* x *in* $\mathbb{F}_p$, *there exists* y *in* $\mathbb{F}_p$ *such that*

$$y^2 = x \quad \vee \quad 2y^2 = x$$

For all $x \in \mathbb{F}_p$, we can compute $x^3 + ax^2 + x$. Using Lemma 7.5.14 we can find a y such that (x, y) is either on the curve or on the quadratic twist:

LEMMA 7.5.15. *For all* $x \in \mathbb{F}_p$, *there exists a point* P *in* $M_{486662,1}(\mathbb{F}_p)$ *or in* $M_{486662,2}(\mathbb{F}_p)$ *such that the x-coordinate of* P *is* x.

```
Theorem x_is_on_curve_or_twist:
  forall x : Zmodp.type,
  (exists (p : mc curve25519_mcuType), p#x0 = x) ∨
  (exists (p' : mc twist25519_mcuType), p'#x0 = x).
```

CURVE25519 OVER $\mathbb{F}_{p^2}$.

The quadratic extension $\mathbb{F}_{p^2}$ is defined as $\mathbb{F}_p[\sqrt{2}]$ by [Bero6b]. The theory of finite fields already has been formalized in the Mathematical Components library [MT21], but this formalization is rather abstract, and we need concrete representations of field elements here. For this reason we decided to formalize a definition of $\mathbb{F}_{p^2}$ ourselves.

We can represent $\mathbb{F}_{p^2}$ as the set $\mathbb{F}_p \times \mathbb{F}_p$, in other words, representing polynomials with coefficients in $\mathbb{F}_p$ modulo $X^2 - 2$. In a similar way as for $\mathbb{F}_p$ we use a module in Coq.

```
Module Zmodp2.
Inductive type := Zmodp2 (x: Zmodp.type) (y:Zmodp.type).

Definition pi (x: Zmodp.type * Zmodp.type) : type :=
  Zmodp2 x.1 x.2.
Coercion repr (x: type) : Zmodp.type*Zmodp.type :=
  let: Zmodp2 u v := x in (u, v).

Definition zero : type :=
  pi ( 0, 0 ).
Definition one : type :=
  pi ( 1, 0 ).
```

We define the basic operations $(+, -, \times)$ with their respective neutral elements 0 and 1. Additionally, we verify that for each element of in $\mathbb{F}_{p^2} \setminus \{0\}$, there exists a multiplicative inverse.

LEMMA 7.5.16. *For all $x \in \mathbb{F}_{p^2} \setminus \{0\}$ and $a, b \in \mathbb{F}_p$ such that $x = (a, b)$,*

$$x^{-1} = \left(\frac{a}{a^2 - 2b^2} , \frac{-b}{a^2 - 2b^2} \right)$$

As in $\mathbb{F}_p$, we define $0^{-1} = 0$ and prove Lemma 7.5.17.

LEMMA 7.5.17. *$\mathbb{F}_{p^2}$ is a commutative field.*

We then specialize the basic operations in order to speed up the verification of formulas by using rewrite rules:

$$(a, 0) + (b, 0) = (a + b, 0) \qquad (a, 0) \cdot (b, 0) = (a \cdot b, 0)$$
$$(a, 0)^{-1} = (a^{-1}, 0) \qquad (0, a)^{-1} = (0, (2 \cdot a)^{-1})$$

The injection $a \mapsto (a, 0)$ from $\mathbb{F}_p$ to $\mathbb{F}_{p^2}$ preserves $0, 1, +, -, \times$. Thus $(a, 0)$ can be abbreviated as a without confusions.

We define $M_{486662,1}(\mathbb{F}_{p^2})$ and the mappings φ_c, φ_t, and ψ as in Definition 7.5.18. With the rewrite rule above, it is straightforward to prove that any point on the curve $M_{486662,1}(\mathbb{F}_p)$ is also on the curve $M_{486662,1}(\mathbb{F}_{p^2})$. Similarly, any point on the quadratic twist $M_{486662,2}(\mathbb{F}_p)$ also corresponds to a point on the curve $M_{486662,1}(\mathbb{F}_{p^2})$.

DEFINITION 7.5.18. *Define the functions φ_c, φ_t and ψ*
- $\varphi_c : M_{486662,1}(\mathbb{F}_p) \mapsto M_{486662,1}(\mathbb{F}_{p^2})$ *such that* $\varphi((x,y)) = ((x,0),(y,0))$.
- $\varphi_t : M_{486662,2}(\mathbb{F}_p) \mapsto M_{486662,1}(\mathbb{F}_{p^2})$ *such that* $\varphi((x,y)) = ((x,0),(0,y))$.
- $\psi : \mathbb{F}_{p^2} \mapsto \mathbb{F}_p$ *such that* $\psi(x,y) = x$.

As direct consequence, using Lemma 7.5.15, we prove that for all $x \in \mathbb{F}_p$, there exists a point $P \in \mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ on $M_{486662,1}(\mathbb{F}_{p^2})$ such that $\chi_0(P) = (x,0) = x$.

```
Lemma x_is_on_curve_or_twist_implies_x_in_Fp2:
  forall (x:Zmodp.type),
    exists (p: mc curve25519_Fp2_mcuType),
      p#x0 = Zmodp2.Zmodp2 x 0.
```

We now study the case of the scalar multiplication and show similar proofs.

LEMMA 7.5.19. *For all $n \in \mathbb{N}$, for all point $P \in \mathbb{F}_p \times \mathbb{F}_p$ on the curve $M_{486662,1}(\mathbb{F}_p)$ (respectively on the quadratic twist $M_{486662,2}(\mathbb{F}_p)$), we have:*

$$P \in M_{486662,1}(\mathbb{F}_p) \rightarrow \varphi_c(n \cdot P) = n \cdot \varphi_c(P)$$
$$P \in M_{486662,2}(\mathbb{F}_p) \rightarrow \varphi_t(n \cdot P) = n \cdot \varphi_t(P)$$

Notice that:

$$\forall P \in M_{486662,1}(\mathbb{F}_p),\ \ \psi(\chi_0(\varphi_c(P))) = \chi_0(P)$$
$$\forall P \in M_{486662,2}(\mathbb{F}_p),\ \ \psi(\chi_0(\varphi_t(P))) = \chi_0(P)$$

In summary for all $n \in \mathbb{N}$, $n < 2^{255}$, for any given point $P \in \mathbb{F}_p \times \mathbb{F}_p$ in $M_{486662,1}(\mathbb{F}_p)$ or $M_{486662,2}(\mathbb{F}_p)$, $Curve25519\_Fp$ computes the $\chi_0(n \cdot P)$. We proved that for all $P \in \mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ such that $\chi_0(P) \in \mathbb{F}_p$ there exists a corresponding point on the curve or the twist over $\mathbb{F}_p$. Moreover, we have proved that for any point on the curve or the twist, we can compute the scalar multiplication by n and obtain the same result as if we did the computation in $\mathbb{F}_{p^2}$.

THEOREM 7.5.20. *For all $n \in \mathbb{N}$, such that $n < 2^{255}$, for all $x \in \mathbb{F}_p$ and $P \in M_{486662,1}(\mathbb{F}_{p^2})$ such that $\chi_0(P) = x$, $Curve25519\_Fp(n,x)$ computes $\chi_0(n \cdot P)$.*

which is formalized in Coq as:

```
Theorem curve25519_Fp2_ladder_ok:
  forall (n : nat) (x:Zmodp.type),
  (n < 2^255)%nat →
  forall (p  : mc curve25519_Fp2_mcuType),
  p #x0 = Zmodp2.Zmodp2 x 0 →
  curve25519_Fp_ladder n x = (p *+ n)#x0 /p.
```

We then prove the equivalence between of operations in $\mathbb{F}_{2^{255}-19}$ and $\mathbb{Z}_{2^{255}-19}$, in other words between `Zmodp` and `:GF`. This allows us to show that given a clamped value n and normalized x-coordinate of P, `RFC` gives the same results as $Curve25519\_Fp$.

All put together, this finishes the proof of the mathematical correctness of X25519: the fact that the code in X25519, both in the RFC 7748 and in the TweetNaCl implementation, correctly computes scalar multiplication in the elliptic curve.

7.6 CONCLUSION

Any formal system relies on a trusted base. In this section we describe our chain of trust.

TRUSTED CODE BASE OF THE PROOF. Our proof relies on a trusted base, i. e., a foundation of definitions that must be correct. One should not be able to prove a false statement in that system, i. e., it should be consistent.

In our case we rely on:

- CALCULUS OF INDUCTIVE CONSTRUCTIONS. The intuitionistic logic used by Coq must be consistent in order to trust the proofs. As an axiom, we assume that the functional extensionality is also consistent with that logic.

$$\forall x, f(x) = g(x) \to f = g$$

```
Lemma f_ext: forall (A B:Type),
  forall (f g: A → B),
  (forall x, f(x) = g(x)) → f = g.
```

- VERIFIABLE SOFTWARE TOOLCHAIN. This framework developed at Princeton allows a user to prove that a Clight code matches pure Coq specification.
- COMPCERT. When compiling with CompCert we only need to trust CompCert's assembly semantics, as the compilation chain has been formally proven correct. However, when compiling with other C compilers like Clang or GCC, we need to trust that the CompCert's Clight semantics matches the C17 standard.
- `clightgen`. The tool translating from C to Clight, the first step of the CompCert compilation. This compilation step is not covered by the proofs of CompCert and VST requires Clight input. VST does not support the direct verification of `o[i] = a[i] + b[i]`. This needs to be rewritten into:

  ```
  aux1 = a[i]; aux2 = b[i];
  o[i] = aux1 + aux2;
  ```

  The `-normalize` flag is taking care of this rewriting and factors out assignments from inside subexpressions.

- Finally, we must trust the Coq kernel and its associated libraries; the Ocaml compiler on which we compiled Coq; the Ocaml Runtime and the CPU. Those are common to all proofs done with this architecture [App15; Coq].

CORRECTIONS IN TWEETNACL. As a result of this verification, we removed superfluous code. Indeed, indexes 17 to 79 of the `i64 x[80]` intermediate variable of `crypto_scalarmult` were adding unnecessary complexity to the code, we removed them.

Peter Wu and Jason A. Donenfeld brought to our attention that the original `car25519` function carried a risk of undefined behavior if `c` is a negative number.

```
c=o[i]>>16;
o[i]-=c<<16; // c < 0 = UB !
```

We replaced this statement with a logical `and`, proved correctness, and thus solved this problem.

```
o[i]&=0xffff;
```

Aside from these modifications, all subsequent alterations to the TweetNaCl code—such as the type change of loop indexes (`int` instead of `i64`)—were required for VST to step through the code properly. We believe that those adjustments do not impact the trust of our proof.

We contacted the authors of TweetNaCl and expect that the changes described above will soon be integrated in a new version of the library.

LESSONS LEARNED. Most efforts in the area of high-assurance crypto are carried out by teams who at the same time work on tools and proofs and often even co-develop the implementations with the proofs. In this effort we set out to verify pre-existing software, written in a not particularly verification-friendly language using a set of tools (VST and Coq) whose development we are not actively involved in.

TweetNaCl comes with a claim of verifiability, but it became clear rather quickly that this claim is only based on the overall simplicity of the library and not supported by code written carefully such that it can efficiently be verified with existing tools. In Section 7.A we provide the verified version of TweetNaCl and the difference with the original TweetNaCl, and thus gives an idea of some minimal changes we had to apply to work with VST; many more small changes would have made the proof much easier and more elegant. As one example, in `pack25519` the subtraction of p and the carry propagation are done in a single `for` loop; had they been split into two loops, the final result would have been the same but with a much smaller verification effort.

There were many positive lessons to be learned from this verification effort; most importantly that it is possible to prove "legacy" cryptographic software written in C correct without having to co-develop proofs and tools. However, we also learned that it is still necessary to understand to some extent how these tools (in particular VST) work

under the hood. VST is a collection of lemmas and proof tactics; the idea is to expose the user only to the tactics and hide the details of the underlying lemmas. At least in the VST versions we worked with, this approach did not quite work and at various stages in the proofs we had to look into the underlying lemmas. This was due to the provided tactics not terminating, for example in the last few steps of `pack25519`'s VST proof. Some struggle with VST also taught us another pleasant lesson, namely that the VST development team is very responsive and helpful. Various of our issues were sorted out with their help, and we hope that some of the experience we brought in also helped improve VST.

EXTENDING OUR WORK. The high-level definition (Section 7.5) can easily be ported to any other Montgomery curve and with it the proof of the ladder's correctness assuming the same formulas are used. In addition to the curve equation, the field $\mathbb{F}_p$ would need to be redefined as $p = 2^{255} - 19$ is hard-coded in order to speed up some proofs.

With respect to the C code verification (Section 7.4), the extension of the verification effort to Ed25519 would make directly use of the low-level arithmetic. As the ladder-steps formula is different, this would require a high level verification similar to Theorem 7.5.9; also, a full correctness verification of Ed25519 signatures would require verifying correctness of SHA-512.

The verification of e. g., X448 [Ham15; LHT] in C would require the adaptation of most of the low-level arithmetic (mainly the multiplication, carry propagation and reduction). Once the correctness and bounds of the basic operations are established, reproving the full ladder would make use of our generic definition.

A COMPLETE PROOF. We provide a mechanized formal proof of the correctness of the X25519 implementation in TweetNaCl from C up the mathematical definitions with a single tool. We first formalized X25519 from RFC 7748 [LHT] in Coq. We then proved that TweetNaCl's implementation of X25519 matches our formalization. In a second step we extended the Coq library for elliptic curves [BS14] by Bartzia and Strub to support Montgomery curves. Using this extension we proved that the X25519 specification from the RFC matches the mathematical definitions as given in [Ber06b, Sec. 2]. Therefore, in addition to proving the mathematical correctness of TweetNaCl, we also increase the trust of other works such as [Zin+17; Erb+16] which rely on RFC 7748.

APPENDIX OF CHAPTER 7

This appendix provides supplementary materials of Chapter 7:
- in Section 7.A, the verified section of TweetNaCl,
- in Section 7.B, the Coq definition and formalization of RFC 7748,
- in Section 7.C, the organization of the proof folders and files,
- and finally in Section 7.D, the proof of correctness of `inv25519`.

7.A THE COMPLETE X25519 CODE FROM TWEETNACL

VERIFIED C CODE. We provide below the code we verified.

```
#define FOR(i,n) for (i = 0;i < n;++i)
#define sv static void

typedef unsigned char u8;
typedef long long i64 __attribute__((aligned(8)));
typedef i64 gf[16];

sv set25519(gf r, const gf a)
{
  int i;
  FOR(i,16) r[i]=a[i];
}

sv car25519(gf o)
{
  int i;
  FOR(i,16) {
    o[(i+1)%16]+=(i<15?1:38)*(o[i]>>16);
    o[i]&=0xffff;
  }
}

sv sel25519(gf p,gf q,int b)
{
  int i;
  i64 t,c=~(b-1);
  FOR(i,16) {
    t= c&(p[i]^q[i]);
    p[i]^=t;
    q[i]^=t;
  }
}

sv pack25519(u8 *o,const gf n)
{
  int i,j,b;
  gf t,m={0};
  set25519(t,n);
  car25519(t);
  car25519(t);
  car25519(t);
  FOR(j,2) {
    m[0]=t[0]-0xffed;
    for(i=1;i<15;i++) {
      m[i]=t[i]-0xffff-((m[i-1]>>16)&1);
```

```
46        m[i-1]&=0xffff;
47      }
48      m[15]=t[15]-0x7fff-((m[14]>>16)&1);
49      b=(m[15]>>16)&1;
50      m[14]&=0xffff;
51      b=1-b;
52      sel25519(t,m,b);
53    }
54    FOR(i,16) {
55      o[2*i]=t[i]&0xff;
56      o[2*i+1]=t[i]>>8;
57    }
58  }
59
60  sv unpack25519(gf o, const u8 *n)
61  {
62    int i;
63    FOR(i,16) o[i]=n[2*i]+((i64)n[2*i+1]<<8);
64    o[15]&=0x7fff;
65  }
66
67  sv A(gf o,const gf a,const gf b)
68  {
69    int i;
70    FOR(i,16) o[i]=a[i]+b[i];
71  }
72
73  sv Z(gf o,const gf a,const gf b)
74  {
75    int i;
76    FOR(i,16) o[i]=a[i]-b[i];
77  }
78
79  sv M(gf o,const gf a,const gf b)
80  {
81    int i,j;
82    i64 t[31];
83    FOR(i,31) t[i]=0;
84    FOR(i,16) FOR(j,16) t[i+j]+=a[i]*b[j];
85    FOR(i,15) t[i]+=38*t[i+16];
86    FOR(i,16) o[i]=t[i];
87    car25519(o);
88    car25519(o);
89  }
90
91  sv S(gf o,const gf a)
92  {
93    M(o,a,a);
94  }
95
96  sv inv25519(gf o,const gf i)
97  {
98    gf c;
99    int a;
100   set25519(c,i);
101   for(a=253;a>=0;a--) {
102     S(c,c);
103     if(a!=2&&a!=4) M(c,c,i);
104   }
105   FOR(a,16) o[a]=c[a];
106 }
107
108 int crypto_scalarmult(u8 *q,const u8 *n,const u8 *p)
109 {
```

```
110     u8 z[32];
111     int r,i;
112     gf x,a,b,c,d,e,f;
113     FOR(i,31) z[i]=n[i];
114     z[31]=(n[31]&127)|64;
115     z[0]&=248;
116     unpack25519(x,p);
117     FOR(i,16) {
118       b[i]=x[i];
119       d[i]=a[i]=c[i]=0;
120     }
121     a[0]=d[0]=1;
122     for(i=254;i>=0;--i) {
123       r=(z[i>>3]>>(i&7))&1;
124       sel25519(a,b,r);
125       sel25519(c,d,r);
126       A(e,a,c);
127       Z(a,a,c);
128       A(c,b,d);
129       Z(b,b,d);
130       S(d,e);
131       S(f,a);
132       M(a,c,a);
133       M(c,b,e);
134       A(e,a,c);
135       Z(a,a,c);
136       S(b,a);
137       Z(c,d,f);
138       M(a,c,_121665);
139       A(a,a,d);
140       M(c,c,a);
141       M(a,d,f);
142       M(d,b,x);
143       S(b,e);
144       sel25519(a,b,r);
145       sel25519(c,d,r);
146     }
147     inv25519(c,c);
148     M(a,a,c);
149     pack25519(q,a);
150     return 0;
151   }
```

DIFF FROM TWEETNACL. We provide below the diff between the original code of TweetNaCl and the code we verified.

```
1   --- tweetnacl.c
2   +++ tweetnaclVerifiableC.c
3   @@ -5,7 +5,7 @@
4    typedef unsigned char u8;
5    typedef unsigned long u32;
6    typedef unsigned long long u64;
7   -typedef long long i64;
8   +typedef long long i64 __attribute__((aligned(8)));
9    typedef i64 gf[16];
10   extern void randombytes(u8 *,u64);
11
12  @@ -273,18 +273,16 @@
13   sv car25519(gf o)
14   {
15     int i;
16  -  i64 c;
17     FOR(i,16) {
```

```
-    o[i]+=(1LL<<16);
-    c=o[i]>>16;
-    o[(i+1)*(i<15)]+=c-1+37*(c-1)*(i==15);
-    o[i]-=c<<16;
+    o[(i+1)%16]+=(i<15?1:38)*(o[i]>>16);
+    o[i]&=0xffff;
   }
 }

 sv sel25519(gf p,gf q,int b)
 {
-  i64 t,i,c=~(b-1);
+  int i;
+  i64 t,c=~(b-1);
   FOR(i,16) {
     t= c&(p[i]^q[i]);
     p[i]^=t;
@@ -295,8 +293,8 @@
 sv pack25519(u8 *o,const gf n)
 {
   int i,j,b;
-  gf m,t;
-  FOR(i,16) t[i]=n[i];
+  gf t,m={0};
+  set25519(t,n);
   car25519(t);
   car25519(t);
   car25519(t);
@@ -309,7 +307,8 @@
     m[15]=t[15]-0x7fff-((m[14]>>16)&1);
     b=(m[15]>>16)&1;
     m[14]&=0xffff;
-    sel25519(t,m,1-b);
+    b=1-b;
+    sel25519(t,m,b);
   }
   FOR(i,16) {
     o[2*i]=t[i]&0xff;
@@ -353,7 +352,8 @@

 sv M(gf o,const gf a,const gf b)
 {
-  i64 i,j,t[31];
+  int i,j;
+  i64 t[31];
   FOR(i,31) t[i]=0;
   FOR(i,16) FOR(j,16) t[i+j]+=a[i]*b[j];
   FOR(i,15) t[i]+=38*t[i+16];
@@ -371,7 +371,7 @@
 {
   gf c;
   int a;
-  FOR(a,16) c[a]=i[a];
+  set25519(c,i);
   for(a=253;a>=0;a--) {
     S(c,c);
     if(a!=2&&a!=4) M(c,c,i);
@@ -394,8 +394,8 @@
 int crypto_scalarmult(u8 *q,const u8 *n,const u8 *p)
 {
   u8 z[32];
-  i64 x[80],r,i;
-  gf a,b,c,d,e,f;
+  int r,i;
```

```
82  +  gf x,a,b,c,d,e,f;
83     FOR(i,31) z[i]=n[i];
84     z[31]=(n[31]&127)|64;
85     z[0]&=248;
86  @@ -430,15 +430,9 @@
87       sel25519(a,b,r);
88       sel25519(c,d,r);
89     }
90  -  FOR(i,16) {
91  -    x[i+16]=a[i];
92  -    x[i+32]=c[i];
93  -    x[i+48]=b[i];
94  -    x[i+64]=d[i];
95  -  }
96  -  inv25519(x+32,x+32);
97  -  M(x+16,x+16,x+32);
98  -  pack25519(q,x+16);
99  +  inv25519(c,c);
100 +  M(a,a,c);
101 +  pack25519(q,a);
102    return 0;
103  }
```

In the following, we provide the explanations of the above changes to TweetNaCl's code.

- lines 7-8: We tell VST that `long long` are aligned on 8 bytes.
- lines 16-23: We remove the undefined behavior as explained in Section 7.6.
- lines 29-31; lines 60-62: VST does not support `for` loops over `i64`, we convert it into an `int`.
- lines 39 & 41: We initialize `m` with `0`. This change allows us to prove the functional correctness of `pack25519` without having to deal with an array containing a mix of uninitialized and initialized values. A hand iteration over the loop trivially shows that no uninitialized values are used.
- lines 40 & 42; lines 70 & 71: We replace the `FOR` loop by `set25519`. The code is the same once the function is inlined. This small change is purely cosmetic but stays in the spirit of TweetNaCl: keeping a small code size while being auditable.
- lines 50-52: VST does not allow computation in the argument before a function call. Additionally, `clightgen` does not extract the computation either. We add this small step to allow VST to carry through the proof.
- lines 79-82: VST does not support `for` loops over `i64`, we convert the loop indexes into an `int`.
  In the function calls of `sel25519`, the specifications requires the use of `int`, the value of `r` being either `0` or `1`, we consider this change safe.
- Lines 90-101: The `for` loop does not add any benefits to the code. By removing it we simplify the source and the verification steps as we do not need to deal with pointer arithmetic. As a result, `x` can be limited to only 16 `i64`, i. e., `gf`.

7.B COQ DEFINITIONS

7.B.1 *Montgomery Ladder*

Generic definition of the ladder:

```
(* Typeclass to encapsulate the operations *)
Class Ops (T T': Type) (Mod: T → T):=
{
  A   : T → T → T;           (* Add           *)
  M   : T → T → T;           (* Mult          *)
  Zub : T → T → T;           (* Sub           *)
  Sq  : T → T;               (* Square        *)
  C_0 : T;                   (* Constant 0    *)
  C_1 : T;                   (* Constant 1    *)
  C_121665: T;               (* const (a-2)/4 *)
  Sel25519: Z → T → T → T; (* CSWAP         *)
  Getbit: Z → T' → Z;       (* ith bit       *)
}.

Local Notation "X + Y" := (A X Y) (only parsing).
Local Notation "X - Y" := (Zub X Y) (only parsing).
Local Notation "X * Y" := (M X Y) (only parsing).
Local Notation "X ²" := (Sq X) (at level 40,
  only parsing, left associativity).

Fixpoint montgomery_rec_swap (m : nat) (z : T')
(a: T) (b: T) (c: T) (d: T) (e: T) (f: T) (x: T) (swap:Z) :
(* a:    x₂                  *)
(* b:    x₃                  *)
(* c:    z₂                  *)
(* d:    z₃                  *)
(* e:    temporary  var      *)
(* f:    temporary  var      *)
(* x:    x₁                  *)
(* swap: previous bit value *)
(T * T * T * T * T * T) :=
match m with
| S n ⇒
  let r := Getbit (Z.of_nat n) z in
    (* k_t = (k >> t) & 1                                  *)
  let swap := Z.lxor swap r in
    (* swap ^= k_t                                         *)
  let (a, b) := (Sel25519 swap a b, Sel25519 swap b a) in
    (* (x₂, x₃) = cswap(swap, x₂, x₃)                      *)
  let (c, d) := (Sel25519 swap c d, Sel25519 swap d c) in
    (* (z₂, z₃) = cswap(swap, z₂, z₃)                      *)
  let e := a + c in          (* A  = x₂ + z₂               *)
  let a := a - c in          (* B  = x₂ - z₂               *)
  let c := b + d in          (* C  = x₃ + z₃               *)
  let b := b - d in          (* D  = x₃ - z₃               *)
  let d := e²    in          (* AA = A²                    *)
  let f := a²    in          (* BB = B²                    *)
  let a := c * a in          (* CB = C * B                 *)
  let c := b * e in          (* DA = D * A                 *)
  let e := a + c in          (* x₃= (DA + CB)²             *)
  let a := a - c in          (* z₃= x₁* (DA - CB)²         *)
  let b := a²    in          (* z₃= x₁* (DA - CB)²         *)
  let c := d - f in          (* E  = AA - BB               *)
  let a := c * C_121665 in (* z₂ = E * (AA + a24 * E)     *)
  let a := a + d in          (* z₂ = E * (AA + a24 * E)     *)
  let c := c * a in          (* z₂ = E * (AA + a24 * E)     *)
```

```
  let a := d * f in        (* x2 = AA * BB                  *)
  let d := b * x in        (* z3 = x1* (DA - CB)^2          *)
  let b := e^2    in       (* x3 = (DA + CB)^2              *)
  montgomery_rec_swap n z a b c d e f x r
    (* swap = k_t                                           *)

| 0%nat ⇒
  let (a, b) := (Sel25519 swap a b, Sel25519 swap b a) in
    (* (x2, x3) = cswap(swap, x2, x3)                       *)
  let (c, d) := (Sel25519 swap c d, Sel25519 swap d c) in
    (* (z2, z3) = cswap(swap, z2, z3)                       *)
  (a,b,c,d,e,f)
end.


Definition get_a (t:(T * T * T * T * T * T)) : T :=
match t with
  (a,b,c,d,e,f) ⇒ a
end.
Definition get_c (t:(T * T * T * T * T * T)) : T :=
match t with
  (a,b,c,d,e,f) ⇒ c
end.
```

7.B.2 *RFC in Coq*

Instantiation of the Class `Ops` with operations over $\mathbb{Z}$ and modulo $2^{255} - 19$.

```
Definition modP (x:Z) : Z :=
  Z.modulo x (Z.pow 2 255 - 19).

(* Encapsulate in a module. *)
Module Mid.
  (* shift to the right by n bits *)
  Definition getCarry (n: Z) (m: Z) : Z :=
    Z.shiftr m n.

  (* logical and with n ones *)
  Definition getResidue (n: Z)  (m: Z) : Z :=
    Z.land n (Z.ones n).

  Definition car25519 (n: Z) : Z  :=
    38 * getCarry 256 n +  getResidue 256 n.
  (* The carry operation is invariant under modulo *)
  Lemma Zcar25519_correct:
    forall (n: Z), n:GF = (Mid.car25519 n) :GF.

  (* Define Mid.A, Mid.M ... *)
  Definition A a b := Z.add a b.
  Definition M a b :=
    car25519 (car25519 (Z.mul a b)).
  Definition Zub a b := Z.sub a b.
  Definition Sq a := M a a.
  Definition C_0 := 0.
  Definition C_1 := 1.
  Definition C_121665 := 121665.
  Definition Sel25519 (b p q: Z) :=
    if (Z.eqb b 0) then p else q.

  Definition getbit (i:Z) (a: Z) :=
```

```
    if (Z.ltb a 0) then      (* a < 0*)
      0
    else if (Z.ltb i 0) then (* i < 0 *)
        Z.land a 1
    else                     (* 0 ≤ a & 0 ≤ i *)
        Z.land (Z.shiftr a i) 1.
End Mid.

(* Clamping *)
Definition clamp (n: list Z) : list Z :=
  (* set last 3 bits to 0 *)
  let x := nth 0 n 0 in
  let x' := Z.land x 248 in
  (* set bit 255 to 0 and bit 254 to 1 *)
  let t := nth 31 n 0 in
  let t' := Z.lor (Z.land t 127) 64 in
  (* update the list *)
  let n' := upd_nth 31 n t' in
    upd_nth 0 n' x'.

(* x^{p - 2} *)
Definition ZInv25519 (x: Z) : Z :=
  Z.pow x (Z.pow 2 255 - 21).

(* reduction modulo P *)
Definition ZPack25519 (n: Z) : Z :=
  Z.modulo n (Z.pow 2 255 - 19).


(* instantiate over Z *)
Instance Z_Ops : (Ops Z Z modP) := {}.
Proof.
  apply Mid.A.         (* instantiate +       *)
  apply Mid.M.         (* instantiate *       *)
  apply Mid.Zub.       (* instantiate -       *)
  apply Mid.Sq.        (* instantiate x^2     *)
  apply Mid.C_0.       (* instantiate Const 0 *)
  apply Mid.C_1.       (* instantiate Const 1 *)
  apply Mid.C_121665.  (* instantiate (a-2)/4 *)
  apply Mid.Sel25519.  (* instantiate CSWAP   *)
  apply Mid.getbit.    (* instantiate ith bit *)
Defined.

Definition decodeScalar25519 (l: list Z) : Z :=
  ZofList 8 (clamp l).

Definition decodeUCoordinate (l: list Z) : Z :=
  ZofList 8 (upd_nth 31 l (Z.land (nth 31 l 0) 127)).

Definition encodeUCoordinate (x: Z) : list Z :=
  ListofZ32 8 x.

(* instantiate montgomery_rec_swap with Z_Ops *)
Definition RFC (n: list Z) (p: list Z) : list Z :=
  let k := decodeScalar25519 n in
  let u := decodeUCoordinate p in
  let t := montgomery_rec_swap
    255  (* iterate 255 times  *)
    k    (* clamped n          *)
    1    (* x_2                *)
    u    (* x_3                *)
    0    (* z_2                *)
    1    (* z_3                *)
    0    (* dummy              *)
```

```
    0    (* dummy              *)
    u    (* x1                 *)
    0    (* previous bit = 0   *) in
  let a := get_a t in
  let c := get_c t in
  let o := ZPack25519 (Z.mul a (ZInv25519 c))
  in encodeUCoordinate o.
```

7.C ORGANIZATION OF THE PROOF FILES

REQUIREMENTS. Our proofs require the use of *Coq 8.8.2* for the proofs and *Opam 2.0* to manage the dependencies. We are aware that there exists more recent versions of Coq; VST; CompCert etc. However, as updating those often lead to breaking our proofs, after spending a significant amount of hours on compilation and debugging, we decided to freeze our dependencies.

ASSOCIATED FILES. The archive containing the proof is composed of two folders `packages` and `proofs`. The content is used at the same time as an *opam* repository to manage the dependencies of the proof and to provide the Coq code.

The actual proofs are found in the `proofs` folder in which the reader will find the directories `spec` and `vst`.

`packages/` This folder provides all the required Coq dependencies: ssreflect (1.7), VST (2.0), CompCert (3.2), the elliptic-curves library by Bartzia & Strub, and the theorem of quadratic reciprocity. For a future proof approach, the source code of those frozen dependencies is directly provided and therefore does not rely on external repositories.

`proofs/spec/` In this folder the reader will find multiple levels of implementation of X25519.

- `Libs/` contains basic libraries and tools to help use reason with lists and decidable procedures.
- `ListsOp/` defines operators on list such as `ZofList` and related lemmas using e. g., `Forall`.
- `Gen/` defines a generic Montgomery ladder which can be instantiated with different operations. This ladder is the stub for the following implementations.
- `High/` contains the theory of Montgomery curves, twists, quadratic extensions and ladder. It also proves the correctness of the ladder over $\mathbb{F}_{2^{255}-19}$.
- `Mid/` provides a list-based implementation of the basic operations `A`, `Z`, `M` ... and the ladder. It makes the link with the theory of Montgomery curves.
- `Low/` provides a second list-based implementation of the basic operations `A`, `Z`, `M` ... and the ladder. Those functions are proven to provide the same results as the ones in `Mid/`, however their implementation are closer to `C` in order facilitate the proof of equivalence with TweetNaCl code.

- `rfc/` provides our rfc formalization. It uses integers for the basic operations `A`, `Z`, `M` ... and the ladder. It specifies the decoding/encoding of/to byte arrays (seen as list of integers) as in RFC 7748.

`proofs/vst/` Here the reader will find four folders.

- `c/` contains the C Verifiable implementation of TweetNaCl. `clightgen` will generate the appropriate translation into Clight.
- `init/` contains basic lemmas and memory manipulation shortcuts to handle the aliasing cases.
- `spec/` defines as Hoare triple the specification of the functions used in `crypto_scalarmult`.
- `proofs/` contains the proofs of the above Hoare triples, and therefore the proof that TweetNaCl code is sound and correct.

7.D PROOF BY REFLECTION OF THE MULTIPLICATIVE INVERSE IN GF

In this section we provide a deeper view of the proof of correctness of `inv25519`, namely:

The implementation of the multiplicative inverse in TweetNaCl (`inv25519`) computes an inverse in $\mathbb{Z}_{2^{255}-19}$.

This proof takes part in three steps, with the last one implementing the proof by reflection.

1. We prove with VST that `inv25519` matches a list implementation of the inversion `Inv25519`.
2. We prove that `Inv25519` over lists is equivalent to `Inv25519_Z` over integers in $\mathbb{Z}_{2^{255}-19}$.
3. We prove that `Inv25519_Z` actually computes an inverse in $\mathbb{Z}_{2^{255}-19}$.

STEP 1. In order to compute an inverse in $\mathbb{Z}_{2^{255}-19}$, `inv25519` uses Fermat's little theorem by raising to the power of $2^{255}-21$ with a square-and-multiply algorithm. The binary representation of $p-2$ implies that every step does a multiplication except for bits 2 and 4 (see Code 7.1).

```
1  sv inv25519(gf o,const gf i)
2  {
3    gf c;
4    int a;
5    set25519(c,i);
6    for(a=253;a>=0;a--) {
7      S(c,c);
8      if(a!=2&&a!=4) M(c,c,i);
9    }
10   FOR(a,16) o[a]=c[a];
11 }
```

Code 7.4: Inverse modulo $2^{255}-19$ in TweetNaCl

To speed up proofs with the VST, we define `Inv25519` with two functions: a recursive `pow_fn_rev` to simulate the `for` loop and a simple `step_pow` containing the body.

```
Definition step_pow (a:Z)
  (c:list Z) (g:list Z) : list Z :=
  let c := Sq c in
    if a ≠? 2 && a ≠? 4
    then M c g
    else c.

Function pow_fn_rev (a:Z) (b:Z)
  (c: list Z) (g: list Z)
  {measure Z.to_nat a} : (list Z) :=
  if a ≤ ? 0
    then c
    else
      let prev := pow_fn_rev (a - 1) b c g in
        step_pow (b - a) prev g.
```

Note that this `Function` requires a proof of termination. It is done by proving the well-foundedness of the decreasing argument: "`measure Z.to_nat a`". As a result, calling `pow_fn_rev` 254 times allows us to reproduce the same behavior as the C definition in Code 7.4.

```
Definition Inv25519 (x:list Z) : list Z :=
  pow_fn_rev 254 254 x x.
```

Using the VST, we prove that `inv25519` matches a list implementation of the inversion `Inv25519`. This theorem is found in the associated materials in `proofs/vst/proofs/verif_inv25519.v`.

STEP 2. We now want to prove that the operations on lists in `Inv25519` are equivalent as if being done on integers modulo $2^{255} - 19$. Therefore, we define the same functions over integers.

```
Definition step_pow_Z (a:Z) (c:Z) (g:Z) : Z :=
  let c := c * c in
  if a ≠? 2 && a ≠? 4
    then c * g
    else c.

Function pow_fn_rev_Z (a:Z) (b:Z) (c:Z) (g: Z)
  {measure Z.to_nat a} : Z :=
  if (a ≤ ? 0)
    then c
    else
      let prev := pow_fn_rev_Z (a - 1) b c g in
        step_pow_Z (b - a) prev g.

Definition Inv25519_Z (x:Z) : Z :=
  pow_fn_rev_Z 254 254 x x.
```

By using Lemma 7.4.1 (`M` correctly implements a multiplication over $\mathbb{Z}_{2^{255}-19}$), we prove their equivalence in $\mathbb{Z}_{2^{255}-19}$.

LEMMA 7.D.1. *The function `Inv25519` over list of integers computes the same result at `Inv25519_Z` over integers in* $\mathbb{Z}_{2^{255}-19}$.

This is formalized in Coq as follows and is found in the associated materials in `proofs/spec/Low/Inv25519.v`.

```
Lemma Inv25519_Z_GF : forall (g:list Z),
  length g = 16 →
  (ℤ16.lst (Inv25519 g)) :GF =
  (Inv25519_Z (ℤ16.lst g)) :GF.
```

STEP 3. Finally, we need to prove that `Inv25519_Z` is computing $x^{2^{255}-21}$. To do so we use a proof by reflection.

Figure 7.8 provides a brief intuition of this technique. First we reify our goal over a DSL, second we apply a decision procedure on the expression, and then we infer the desired proof by soundness.

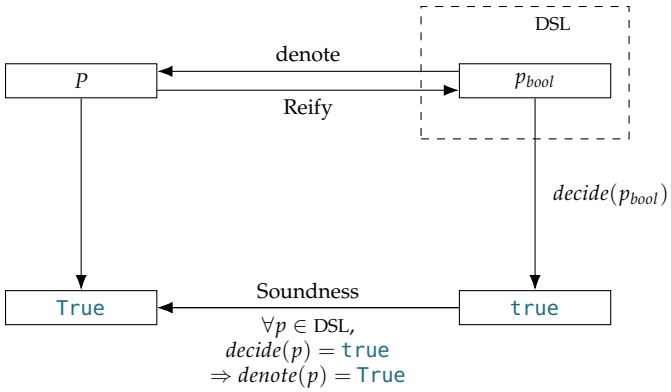


Figure 7.8: Overview of a proof by reflection.

We define a DSL to express monomials.

DEFINITION 7.D.2. *`expr_inv` denotes an expression which is either a term, a product of expressions, a square of an expression, or a power of an expression; By extension `formula_inv` denotes an equality between two expressions.*

```
Inductive expr_inv :=
  | R_inv : expr_inv
  | M_inv : expr_inv → expr_inv → expr_inv
  | S_inv : expr_inv → expr_inv
  | P_inv : expr_inv → positive → expr_inv.

Inductive formula_inv :=
  | Eq_inv : expr_inv → expr_inv → formula_inv.
```

We introduce an environment, i. e., a partial function `vars` which given a positive value returns an integer. This allows us to encapsulate simple variable such as x, but also more complex expressions as $57xyz$.

```
Record environment := { vars : positive → Z }.
```

The *denote* functions take the environment as argument and are defined as follows:

```
Fixpoint e_inv_denote (env:environment) (m:expr_inv) : Z :=
  match m with
  | R_inv     ⇒
    vars env 1
  | M_inv x y ⇒
    (e_inv_denote env x) * (e_inv_denote env y)
  | S_inv x ⇒
    (e_inv_denote env x) * (e_inv_denote env x)
  | P_inv x p ⇒
    pow (e_inv_denote env x) (Z.pos p)
  end.

Definition f_inv_denote (env:environment) (t:formula_inv) : Prop :=
  match t with
  | Eq_inv x y ⇒  e_inv_denote env x = e_inv_denote env y
  end.
```

In our use case, as we work on a very simple monomial and we only have one variable x, we were able to simplify the denote function. A simple example of this DSL is depicted with the following equality:

```
Variable x:Z.
Definition env0 := {| vars := fun _:positive ⇒  x |} : environment.

Goal f_inv_denote env0
 (Eq_inv (M_inv R_inv (S_inv R_inv))
         (P_inv R_inv 3))
  = (x * x^2 = x^3).
```

On the right side of the goal (`x * x^2 = x^3`) depends on x while the left side (`Eq_inv (M_inv R_inv (S_inv R_inv)) (P_inv R_inv 3)`) does not. This allows us to use efficient computations (`compute`) in our decision procedure to prove formulas in `formula_inv`. We use the following simple decidable steps: (1) we compute the power of an expression with `compute_pow_expr_inv`, (2) we compare the values of both side and decide over their equality.

```
Fixpoint compute_pow_expr_inv (t:expr_inv) : Z :=
  match t with
  | R_inv   ⇒ 1       (* power of a term is 1. *)
  | M_inv x y ⇒       (* x^a * x^b = x^{a+b}. *)
    (compute_pow_expr_inv x) + (compute_pow_expr_inv y)
  | S_inv x ⇒         (* (x^a)^2 = x^{2a}. *)
    2 * (compute_pow_expr_inv x)
  | P_inv x p ⇒       (* (x^b)^a = x^{a*b}. *)
    (Z.pos p) * (compute_pow_expr_inv x)
  end.

Definition decide_e_inv (l1 l2:expr_inv) : bool :=
  (compute_pow_expr_inv l1) ==? (compute_pow_expr_inv l2).

Definition decide_f_inv (f:formula_inv) : bool :=
  match f with
  | Eq_inv x y ⇒  decide_e_inv x y
  end.
```

We prove our decision procedure correct:

LEMMA 7.D.3 (SOUNDNESS). *For all formulas f, if the decision over f returns `true`, then the denoted equality by f is true.*

```
Lemma decide_formula_inv_impl :
  forall (f:formula_inv),
  decide_f_inv f = true → f_inv_denote f.
```

Our DSL captures simple monomials, however we also need to compute the value of the exponent used by the square-and-multiply algorithm in `Inv25519_Z`. To do so, we define `step_inv` and `pow_inv` to mirror the behavior of `step_pow_Z` and respectively `pow_fn_rev_Z` over our DSL. By using the *denote* function we ensure the equality.

```
Definition step_inv a c g : expr_inv :=
  let c := (S_inv c) in
  if a ≠? 2 && a ≠? 4
    then M_inv (S_inv c) g
    else (S_inv c).

(* Similar definition of pow_inv *)
Lemma step_inv_step_pow_eq :
  forall (a:Z) (c:expr_inv) (g:expr_inv),
  e_inv_denote (step_inv a c g) =
  step_pow_Z a (e_inv_denote c) (e_inv_denote g).

Lemma pow_inv_pow_fn_rev_eq :
  forall (a:Z) (b:Z) (c:expr_inv) (g:expr_inv),
  e_inv_denote (pow_inv a b c g) =
  pow_fn_rev_Z a b (e_inv_denote c) (e_inv_denote g).
```

We then derive the following:

LEMMA 7.D.4. *With an appropriate choice of variables, `pow_inv` denotes `Inv25519_Z`.*

By reification to our DSL (Lemma 7.D.4) and by applying our decision (Lemma 7.D.3), we prove the subsequent lemma:

LEMMA 7.D.5. *`Inv25519_Z` computes an inverse in $\mathbb{Z}_{2^{255}-19}$.*

This is formalized as (using Fermat's little theorem):

```
Theorem Inv25519_Z_correct :
  forall (x:Z),
  Inv25519_Z x = pow x (2^255 − 21).
```

From Lemma 7.D.1 and Lemma 7.D.5, we conclude the functional correctness of the inversion over $\mathbb{Z}_{2^{255}-19}$.

COROLLARY 7.D.6. *`Inv25519` computes an inverse in $\mathbb{Z}_{2^{255}-19}$.*

```
Corollary Inv25519_Zpow_GF :
  forall (g:list Z),
  length g = 16 →
  ℤ16.lst (Inv25519 g) :GF  =
  (pow (ℤ16.lst g) (2^255 − 21)) :GF.
```

Part IV

STANDARDIZING

8

KANGAROOTWELVE: FAST HASHING BASED ON KECCAK-p

Most cryptography involves careful trade-offs between performance and security. The performance of a cryptographic function can be objectively measured, although it can yield a wide spectrum of figures depending on the variety of hardware and software platforms that the users may be interested in. Out of these, performance on widespread processors is easily measurable and naturally becomes the most visible feature. Security on the other hand cannot be measured. The best one can do is to obtain security assurance by relying on public scrutiny by skilled cryptanalysts. This is a scarce resource and the gaining of insight requires time and reflection. With the growing emphasis on provable security reduction of modes, the fact that the security of the underlying primitives is still based on public scrutiny should not be overlooked.

In this chapter we present the hash function KangarooTwelve. We give its specifications in Section 8.2 and our design rationale in Section 8.3. In Section 8.4 we introduce a closely related variant called MarsupilamiFourteen. Furthermore, in Section 8.5 we discuss implementation aspects and display benchmarks for recent processors.

8.1 INTRODUCTION

In a few words, KangarooTwelve is a hash function —or more exactly an eXtendable Output Function (XOF)— that makes use of a tree hash mode with Sakura encoding [NIS15; Ber+14a] and the sponge construction [Ber+08a], both proven secure. Its underlying permutation is a member of the Keccak-$p[1600, n_r]$ family, differing from that of Keccak only in the number of rounds. Since its publication in 2008, the round function of Keccak was never tweaked [Ber+08b]. Moreover, as for most symmetric cryptographic primitives, third-party cryptanalysis has been applied to reduced-round versions of Keccak. Hence KangarooTwelve's security assurance directly benefits from more than twelve years of public scrutiny, including all cryptanalysis during and after the SHA-3 competition [Ber+17b].

KangarooTwelve gets its low computational workload per bit from using the Keccak-$f[1600]$ permutation reduced to 12 rounds. Clearly, 12 rounds provide less safety margin than the full 24 rounds in SHA-3 and SHAKE functions [NIS15]. Still, the safety margin provided by 12 rounds is comfortable as, e. g., the best published collision attacks at time of writing break Keccak only up to 6 rounds [DDS13; DDS14; SLG17a; SLG17b].

The other design choice that gives KangarooTwelve great speed for long messages is the use of a tree hash mode. This mode is transparent for the user in the sense that the message length fully determines the tree topology. Basically, the mode calls an underlying sponge-based compression function for each 8192-byte chunk of message and finally hashes the concatenation of the resulting digests. We call this the *final node growing* approach. Clearly, the chunks can be hashed in parallel.

The main advantage of the final node growing approach is that implementors can decide on the degree of parallelism their programs support. A simple implementation could compute everything serially, while another would process two, four or more branches in parallel using multiple cores, or more simply, a SIMD instruction set such as the Intel AVX2. Future processors can even contain an increasing number of cores, or wider SIMD registers as exemplified by the recent AVX-512 instruction set, and KangarooTwelve will be readily able to exploit them. The fixed length of the chunks and the fact that the tree topology is fully determined by the message length improve interoperability: the hash result is independent of the amount of parallelism exploited in the implementation.

KangarooTwelve is not the only Keccak-based parallel hash mode. In late 2016, NIST published the SP 800-185 standard, including a parallelized hash mode called ParallelHash [NIS16]. Compared to ParallelHash, KangarooTwelve improves on the speed for short messages. ParallelHash compresses message chunks to digests in a first stage and compresses the concatenation of the digests in a second stage. This two-stage hashing introduces an overhead that is costly for short messages. In KangarooTwelve we apply a technique called *kangaroo hopping*: It merges the hashing of the first chunk of the message and that of the chaining values of the remaining chunks [Ber+14a]. As a result, the two stages reduce to one if the input fits in one chunk with no overhead whatsoever.

Finally, KangarooTwelve is a concrete application of the Sakura encoding, which yields secure tree hash modes by construction [Ber+14a].

8.2 SPECIFICATIONS OF KANGAROOTWELVE

KangarooTwelve is an eXtendable Output Function (XOF) [Per14]. It takes as input a message M and an optional customization string C, both byte strings of variable length.

KangarooTwelve produces unrelated outputs on different couples (M, C). The customization string C is meant to provide *domain separation*, namely, for two different customization strings $C_1 \neq C_2$, KangarooTwelve gives two independent functions of M. In practice, C is typically a short string; such as a name, an address, or an identifier (e. g., URI, OID). KangarooTwelve naturally maps to a XOF with a single input string M by setting the customization string input C to

the empty string. This allows implementing it with a classical hash function Application Programming Interface (API).

As a XOF, the output of KangarooTwelve is unlimited, and the user can request as many output bits as desired. It can be used for traditional hashing simply by generating outputs of the desired digest size.

We provide a reference implementation in Section 8.A and in the associated materials of this thesis (Section 1.3).

8.2.1 *The inner compression function F*

The core of KangarooTwelve is the Keccak-$p[1600, n_r = 12]$ permutation, i. e., a version of the permutation used in SHAKE and SHA-3 instances reduced to $n_r = 12$ rounds [NIS15]. We build a sponge function F on top of this permutation with capacity set to $c = 256$ bits and therefore with rate $r = 1600 - c = 1344$. It makes use of multi-rate padding, indicated by pad10*1. Following [NIS15], this is expressed formally as:

$$F = \text{SPONGE}[\text{Keccak-}p[1600, n_r = 12], \text{pad10*1}, r = 1344].$$

On top of the sponge function F, KangarooTwelve uses a Sakura-compatible tree hash mode, which we now describe shortly.

8.2.2 *The merged input string S*

First, we merge M and C to a single input string S in a reversible way by concatenating:

1. the input message M;
2. followed by the customization string C;
3. and followed by the length in bytes of C encoded using `length_encode` $(\|C\|)$ as in Algorithm 8.

Then, the input string S is cut into chunks of $B = 8192$ bytes, i. e.,

$$S = S_0 \| S_1 \| \dots \| S_{n-1},$$

with $n = \left\lceil \frac{\|S\|}{B} \right\rceil$ and where all chunks except the last one must have exactly B bytes. Note that there is always one block as S consists of at least one byte.

8.2.3 *The tree hash mode*

When $\|S\| > B$, we have $n > 1$ and KangarooTwelve builds a tree with the following final node $\texttt{Node}_*$ and inner nodes $\texttt{Node}_i$ with $1 \leq i \leq n - 1$:

$$\texttt{Node}_i \leftarrow S_i \| \texttt{'110'}$$

ALGORITHM 8. The function `length_encode`(x).

`Input:` an integer x in the range $0 \leq x \leq 256^{255} - 1$
`Output:` a byte string

Let l be the smallest integer in the range $0 \leq l \leq 255$ such that $x < 256^l$
Let $x = \sum_{i=0}^{l-1} x_i 256^i$ with $0 \leq x_i \leq 255$ for all i return $\text{enc}_8(x_{l-1}) \| \dots \| \text{enc}_8(x_1) \| \text{enc}_8(x_0) \| \text{enc}_8(l)$

EXAMPLES:
`length_encode`(0) returns `0x00`
`length_encode`(12) returns `0x0C`$\|$`0x01`
`length_encode`(65538) returns `0x01`$\|$`0x00`$\|$`0x02`$\|$`0x03`

$$\begin{aligned} \text{CV}_i &\leftarrow \lfloor F(\texttt{Node}_i) \rfloor_{256} \\ \texttt{Node}_* &\leftarrow S_0 \| \text{`110}^{62}\text{'} \\ &\quad \| \text{CV}_1 \| \dots \| \text{CV}_{n-1} \\ &\quad \| \texttt{length\_encode}(n-1) \\ &\quad \| \texttt{0xFF} \| \texttt{0xFF} \| \text{`01'} \\ \text{KangarooTwelve}(M, C) &= F(\texttt{Node}_*). \end{aligned}$$

The chaining values CV_i have length $c = 256$ bits. This is illustrated in Figure 8.1.

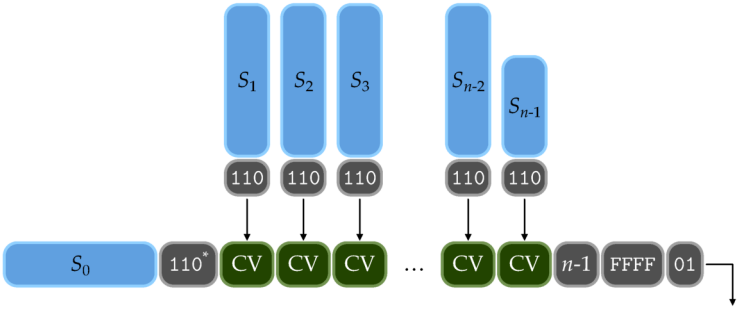


Figure 8.1: Schematic of KangarooTwelve for $\|S\| > B$, with arrows denoting calls to F.

When $\|S\| \leq B$, we have $n = 1$ and the tree reduces to its single final node $\texttt{Node}_*$ and KangarooTwelve becomes:

$$\begin{aligned} \texttt{Node}_* &\leftarrow S \| \text{`11'} \\ \text{KangarooTwelve}(M, C) &= F(\texttt{Node}_*). \end{aligned}$$

8.2.4 *Security claim*

We make a flat sponge claim [Ber+07] with 255 bits of claimed capacity in Claim 8.2.1. Informally, it means that KangarooTwelve shall offer the same security strength as a random oracle whenever that offers a strength below 128 bits and a strength of 128 bits in all other cases. We discuss the implications of the claim more in depth in Section 8.3.1.

CLAIM 8.2.1 (FLAT SPONGE CLAIM [BER+07]). *The success probability of any attack on* KangarooTwelve *shall not be higher than the sum of that for a random oracle and*

$$1 - \mathrm{e}^{-\frac{N^2}{2^{256}}},$$

with N the attack complexity in calls to Keccak-$p[1600, n_\mathrm{r} = 12]$ *or its inverse. We exclude from the claim weaknesses due to the mere fact that the function can be described compactly and can be efficiently executed, e. g., the so-called random oracle implementation impossibility [MRH04], as well as properties that cannot be modeled as a single-stage game [RSS11].*

Note that $1 - \mathrm{e}^{-\frac{N^2}{2^{256}}} < \frac{N^2}{2^{256}}$.

8.3 RATIONALE

In this section, we provide some more in-depth explanations on the design choices in KangarooTwelve.

8.3.1 *Implications of the security claim*

The flat sponge claim covers all attacks up to a given *security strength* of 128 bits. Informally, saying that a cryptographic function has a security strength of s bits means that no attacks exist with complexity N and success probability p such that $N/p < 2^s$ [MW18].

The claim covers quasi all practically relevant security of KangarooTwelve including that of traditional hashing: collision, preimage and second preimage resistance. To achieve 128-bit security strength, the output n must be chosen long enough so that there are no generic attacks (i. e., also applicable to a random oracle) that violate 128-bit security. So for 128-bit (second) preimage security the output should be at least 128 bits, and for 128-bit collision security the output should be at least 256 bits.

For many primitives the security strength that can be claimed degrades under multi-target attacks by $\log_2 M$ bits with M the number of targets. This is not the case for the flat sponge claim. As an example, let us take the case of a multi-target preimage attack versus a single-target preimage attack.

- In a (single-target) preimage attack, the adversary is given a n-bit challenge y and has to find an input x such that $\lfloor f(x) \rfloor_n = y$. A random oracle offers n bits of security strength: After N attempts, the total success probability is p with $p \approx N2^{-n}$. So we have that $N/p \approx 2^n$ for $N < 2^n$ and the security strength for a random oracle is n. For KangarooTwelve we claim security strength $\min(n, 128)$ bits in this case.
- In an M-target preimage attack, the adversary is given M challenges, y_1 to y_M, and she succeeds if she finds an input x such that $\lfloor f(x) \rfloor_n = y_i$ for any of the challenges. A random oracle with N attempts has a success probability p with of $p \approx MN2^{-n}$, and hence $N/p \approx 2^n/M$. So the security strength for the random oracle reduces to $n - \log M$ bits. For KangarooTwelve we claim security strength $\min(n - \log M, 128)$ bits in this case.

Clearly, the reduction in security due to M targets is generic and independent of the security strength. It can be compensated for by increasing the output length n by $\log M$ bits.

8.3.2 *Security of the mode*

The security of the mode, or the generic security, relies on both the sponge construction and on the tree hash mode. The latter is Sakura-compatible so that it automatically satisfies the conditions of soundness and guarantees security against generic attacks, see [Ber+14a, Theorem 1] and [Ber+14b, Theorem 1]. In both cases, the bottleneck is the ability to generate collisions in the chaining values, or equivalently, collisions of the inner hash function.

The probability of inner collisions in the sponge construction is $N^2/2^{c+1}$, with N the number of blocks [Ber+08a]. Regarding the collisions in the chaining values of the tree hash mode, the probability is at most $q^2/2^{c+1}$ [Ber+14b, Theorem 1] with q the number of queries to F. Since each query to F implies at least one block be processed by the sponge construction, we have $q \leq N$ and we can bound the sum of the two probabilities as $N^2/2^{c+1} + q^2/2^{c+1} \leq N^2/2^{(c-1)+1}$. This expression is equivalent as if c was one bit less than with a single source of collisions, and Claim 8.2.1 takes this into account by setting the claimed capacity to $c - 1 = 255$ bits.

We formalize the security of KangarooTwelve's mode of operation in the following theorem. We can see the combination of the tree hash mode and the sponge construction as applied in KangarooTwelve as a mode of operation of a permutation and call it $\mathcal{K}$.

THEOREM 8.3.1. *The advantage of differentiating $\mathcal{K}$, where the underlying permutation is uniformly chosen among all the possible* 1600*-bit permutations, is upper bounded by*

$$\frac{2N^2 + N}{2^{c+1}},$$

with N the number of calls to the underlying permutation.

Proof. By the triangle inequality, the advantage in distinguishing $\mathcal{K}$ calling a random permutation from a random oracle is upper bounded by the sum of two advantages:

- that of distinguishing the tree hash mode calling as inner function a random function $\mathcal{F}$ from a random oracle;
- that of distinguishing the sponge construction calling a random permutation from a random function.

The former advantage is upper bounded by $q^2/2^{c+1}$, where q is the number of calls to $\mathcal{F}$. This follows from Theorem 1 of [Ber+14b] for any sound tree hash mode, and from Theorem 1 of [Ber+14a] that says that any SAKURA-compatible tree hash mode is sound. We show that the tree hash mode is indeed SAKURA-compatible in Section 8.3.3.

Following Theorem 2 of [Ber+08a], the latter advantage is upper bounded by $(N^2 + N)/2^{c+1}$. Adding the two bounds and using $q \leq N$ proves our theorem. □

8.3.3 SAKURA *compatibility*

To show SAKURA-compatibility, we use the following terminology. The inputs to the underlying hash function are called *nodes*. Each node consists of one or more *hops*, and a hop is either a chunk of the message or a sequence of chaining values.

The encoding of the nodes follows [Ber+14a, Section 3.1]:

- When $n = 1$, the tree reduces to a single node. This is the final node, and it contains a single message hop consisting of the input string S followed by the frame bits "message hop" '1' and "final node" '1'.
- When $n > 1$, there are inner nodes and the final node.
  - Each inner node contains a message hop consisting of a chunk S_i followed by the frame bit "message hop" '1'; a simple padding bit '1' and "inner node" '0'.
  - The final node contains two hops: a message hop followed by a chaining hop. The message hop is the first chunk of the input string S_0 followed by the frame bit "message hop" '1' and a padding string '1'$\|$'0^{62}' to align the chaining hop to 64-bit boundaries. The chaining hop consists of the concatenation of the chaining values, the coded number of chaining values (`length_encode`$(n-1)$), the indication that there was no interleaving ($I = \infty$, coded with the bytes `0xFF`$\|$`0xFF`), and the frame bits "chaining hop" '0' and "final node" '1'.

8.3.4 *Choice of B*

We fix the size of the message chunks to make KangarooTwelve a function without parameters. This frees the user from the burden of this technical choice and facilitates interoperability.

In particular, we chose $B = 8192$. First, we chose a power of two as this can help to fetch bulk data in time-critical applications. For instance, when hashing a large file, we expect the implementation to be faster and easier if the chunks contain a whole number of disk sectors.

As for the order of magnitude of B, we took into account following considerations. For each B-byte block there is a 32-byte chaining value in the final node, giving rise to a relative processing overhead of about $32/B$. Choosing $B = 2^{13}$, this is only $2^{-8} \approx 0.4\%$.

Another concern is the number of *unused bytes* in the last r-bit block of the input to F. We have $r = 1344$ bits or $R = r/8 = 168$ bytes. When cutting the chunk S_i into blocks of R bytes, it leaves $W = -(B+1) \bmod R$ unused bytes in the last block. It turns out that W reaches a minimum for $B = 2^{7+6n}$ with $n \geq 0$ an integer. Its relative impact, $\frac{W}{B}$, decreases as B increases. For small values, e. g., $B \in \{128, 256, 512\}$, this is about 30%, while for $B = 8192$ it drops below 0.5%.

There is a tension between a larger B and the exploitable parallelism. Increasing B would further reduce these two overhead factors, but it would also delay the benefits of parallelism to longer messages.

Finally, the choice of B bounds the degree of parallelism that an implementation can fully exploit. An implementation can in principle compute the final node and leaves in parallel, but if more than $B/32$ leaves are processed at once, the final node grows faster than B bytes at a time. The chosen value of B allows a parallelism up to degree $B/32 = 256$.

8.3.5 *Choice of the number of rounds*

Opting for the Keccak-$p[1600, n_r = 12]$ permutation is a drastic reduction in the number of rounds compared to the nominal Keccak and to the SHA-3 standard. Still, there is ample evidence from third-party cryptanalysis that the switch to Keccak-$p[1600, n_r = 12]$ leaves a safety margin similar to the one in the SHA-2 functions.

Currently, the best collision attack applicable to KangarooTwelve or any SHA-3 instance works only when the permutation is reduced to 5 rounds [SLG17a]. The attack extends to 6 rounds if more degrees of freedom are available and requires a reduction of the capacity from 256 to 160 bits. Preimage attacks reach an even smaller number of rounds [GLS16]. Hence, our proposal has a safety margin of 7 out of 12 rounds with respect to collision and (second) preimage resistance.

Structural distinguishers is the term used for properties of a specific function that are very unlikely to be present in a random function. Zero-sum distinguishers were applied to the KECCAK-$p[1600, n_r]$ family of permutations in a number of publications [AM09; BCC11; GLS16]. They allow producing a set of input and of output values that both sum to zero, and this in about half the time it would be needed on a random permutation with only black-box access. These structural distinguishers are of nice theoretical interest, but they do not pose a threat as they do not extend to distinguishers on sponge functions that use KECCAK-$p[1600, n_r]$, see, e. g., [SKC17].

The structural distinguisher on the KECCAK sponge function that does reach the highest number of rounds is the keystream prediction by Dinur et al. [Din+15]. It works when the permutation is reduced to 9 rounds, with a time and data complexity of 2^{256}, and allows them to predict one block of output. This is above the security claim of KANGAROOTWELVE, but the same authors propose a variant that works on 8 rounds with a time and data complexity of 2^{128}, leaving a safety margin of 4 rounds or 33% for KANGAROOTWELVE against this rather academic attack. Examples of structural distinguishers for the KECCAK sponge function with practical complexity and reaching the highest number of rounds are reported by Huang et al. and work up to 7 rounds [Hua+17].

In comparison, SHA-256 has a collision attack on 31 (out of 64) steps and its compression function on 52 steps [LIS12; MNS13]. SHA-512's compression function admits collision attacks with practical complexities for more than half of its steps [DEM15].

8.4 MARSUPILAMIFOURTEEN

While KANGAROOTWELVE claims a strong notion of 128-bit security strength, and we believe any security beyond it is purely of theoretical interest, some users may wish to use a XOF or hash function with higher security strength. In particular, when defining a cipher suite or protocol aiming for 256-bit security strength, all cryptographic functions shall have at least 256-bit security. Coming forward to such requests, in this section we present a variant of KANGAROOTWELVE with 511-bit claimed capacity.

Addressing a claimed capacity of 511 bits requires an increase of both the capacity in F and the length of chaining values in the tree hash mode to at least 512 bits. Taking exactly $c = 512$ bits is sufficient for resisting generic attacks below the claim. As for KECCAK-p-specific attacks, the increase of the claimed capacity to 511 bits increases the available budget of attackers and hence reduces the safety margin. In many types of attack, adding a round in the primitive (permutation or block cipher) increases the attack complexity by a large factor. Or the other way round, if one wishes to keep the same safety margin,

an increase of the attack complexity must be compensated by adding rounds.

We did the exercise and the result is MARSUPILAMIFOURTEEN. It has the same specifications as KANGAROOTWELVE, with the following exceptions:

- The capacity and chaining values are 64-byte long instead of 32 bytes. This reduces the sponge rate in F to 136 bytes.
- The number of rounds of KECCAK-$p[1600, n_r]$ is 14 instead of 12.
- The claimed capacity in the flat sponge claim is 511 bits instead of 255.

The computational workload per bit is roughly 45% higher than that of KANGAROOTWELVE.

Naturally, even thicker safety margins are achieved with the standard FIPS 202 instances or ParallelHash [NIS15; NIS16].

8.5 IMPLEMENTATION

We implemented KANGAROOTWELVE in C and made it available in the KECCAK code package (KCP) [Ber+16]. We now review different aspects of this implementation and its performance.

8.5.1 *Byte representation*

KANGAROOTWELVE assumes that its inputs M and C are byte strings. SAKURA encoding works at the bit level and adds padding and suffixes so that the input to the function F is a string of bits whose length is in general not a multiple of 8.

It is common practice in implementations of KECCAK to represent the last few bits of a string as a *delimited suffix* [Ber+16]. The delimited suffix is a byte that contains the last $|X| \bmod 8$ bits of a string X, with $|X|$ the length of X in bits, followed by the delimiter bit '`1`', and ending with the necessary number of bits '`0`' to reach a length of 8 bits. When absorbing the last block in the sponge function F, the delimiter bit coincides with the first bit '`1`' of the pad10*1 padding rule. An implementation can therefore process the first $\lfloor |X|/8 \rfloor$ bytes of the string S and, in the last block, simply add the delimited suffix and the second bit of the pad10*1 padding rule at the last position of the outer part of the state (i. e., at position $r - 1$, with r the rate).

Following the convention in Section 2.1, the delimited suffix of a string with last bits $(s_0, \ldots, s_{n-1})$ can be represented by the value $2^n + \sum_{i=0}^{n-1} s_i 2^i$ in hexadecimal. For KANGAROOTWELVE, this concretely means that the final node with $\|S\| \leq B$ has suffix '`11`' and delimited suffix `0x07`. With $\|S\| > B$ the intermediate nodes with trailing bits '`110`' use `0x0B` (as depicted in Figure 8.2), and the final node ending with '`01`' will have `0x06` as delimited suffix.

Figure 8.2: Example of delimited suffix.

On a similar note, the 64-bit string '$\texttt{110}^{62}$' in the final node is represented by the bytes $\texttt{0x03}\|(\texttt{0x00})^7$, still following the convention in Section 2.1.

This approach is taken by the Internet Research Task Force (IRTF) RFC draft describing KangarooTwelve [Vig18] and in the reference source code in Section 8.A.

8.5.2 *Structuring the implementation*

The implementation has an interface that accepts the input message M in pieces of arbitrary sizes. This is useful if a file, larger than the memory size, must be processed. The customization string C can be given at the end.

We have integrated the KangarooTwelve code in KCP as illustrated on Figure 8.3. In particular, we instantiate the sponge construction on top of Keccak-$p[1600, n_r = 12]$ to implement the function F, at least to compute the final node. The function F on the leaves is computed as much as possible in parallel, i. e., if at least $8B$ input bytes are given by the caller, it uses a function that computes 8 times Keccak-$p[1600, n_r = 12]$ in parallel; if it is not available and if at least $4B$ bytes are given, it computes $4 \times$ Keccak-$p[1600, n_r = 12]$ in parallel; and so on. If no parallel implementation exists for the given platform, or if not enough bytes are given by the caller, it falls back on a serial implementation like for the final node.

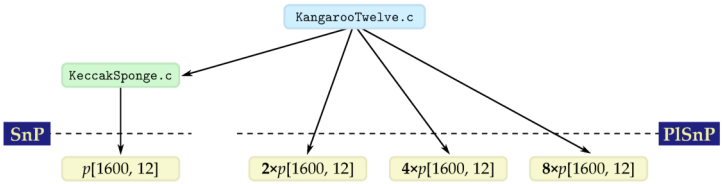


Figure 8.3: The structure of the code implementing KangarooTwelve in the KCP.

The KCP foresees that the serial and parallel implementations of the Keccak-p permutation can be optimized for a given platform. In contrast, the code for the tree hash mode and the sponge construction is generic C, without optimizations for specific platforms, and it accesses

the optimized permutation-level functions through an interface called *SnP* (for a single permutation) or *PlSnP* (for permutations computed in parallel) [Ber+16].

To input large messages M, the state to maintain between two calls internally uses two queues: one for the final node and one for the current leaf. To save memory, the input bytes are absorbed directly into the state of F as they arrive. Hence, the state reduces to two times the state of F. Of course, if a message is known to be smaller than or equal to B bytes, one could further save one queue.

8.5.3 *256-bit SIMD*

Current mainstream PC processors, in the Intel Haswell and Skylake families, support a 256-bit SIMD instruction set called AVX2. We can exploit it to compute $4 \times$ KECCAK-$p[1600, n_r = 12]$ efficiently, even on a single core.

On an Intel Core i5-6500 (Skylake), we measured that one evaluation of KECCAK-$p[1600, n_r = 12]$ takes about 450 cycles, while 2 in parallel about 730 cycles and $4 \times$ KECCAK-$p[1600, n_r = 12]$ about 770 cycles. This does not include the time needed to add the input bytes to the state. Yet, this clearly points out that the time per byte decreases with the degree of parallelism.

Figure 8.4 displays the number of cycles for input messages up to $150,000$ bytes. Microscopically, the computation time steps up for every additional $R = 168$ bytes, but this is not visible on the figure. The time needed to hash messages of length smaller than 168 bytes thus represents the smallest granularity and is reported in Table 8.1. Note that if many very short messages have to be processed, they can be batched to use a parallel implementation. This case is also reported in Table 8.1.

Macroscopically, when $\|S\| < B$, the time is a straight line with a slope of about 2.89 cycles/byte, i. e., the speed for F implemented serially. At $\|S\| = B = 8192$, there is a slight bump (a) as the tree gets a leaf, which causes an extra evaluation of KECCAK-$p[1600, n_r = 12]$. When $\|S\| = 3B = 24,576$, two leaves can be computed in parallel and the number of cycles drops. When $\|S\| = 5B = 40,960$, four leaves can be computed in parallel and we see another drop. From then on, the same pattern repeats and one can easily identify the slopes of serial, $\times 2$ and $\times 4$ parallel implementations of KECCAK-$p[1600, n_r = 12]$.

In our implementation, the final node is always processed with a serial implementation. In principle, a more advanced implementation could process the final node in parallel with the leaves. In more details, it would process the first chunk S_0 in parallel with the first few leaves, and it would buffer about B bytes of chaining values and to process them in parallel with the next leaves. However, at this point, we preferred code simplicity over speed optimization. Similarly, one could in principle remove the peaks of Figure 8.4 and make it monotonous. It

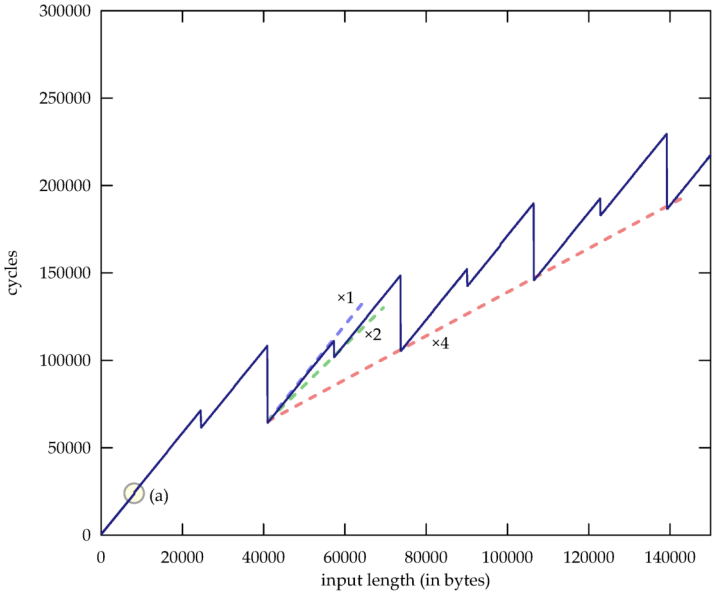


Figure 8.4: The number of cycles of KangarooTwelve on an Intel Core i5-6500 (Skylake) as a function of the input message size.

could be achieved by using, e. g., the fast $4 \times$ Keccak-$p[1600, n_r = 12]$ implementation even if there are less than $4B$ bytes available, with some dummy input bytes.

Figure 8.5 shows the implementation cost in cycles per bytes. To determine the speed in cycles per byte for long messages in our implementation, we need to take into account both the time to process $4B$ input bytes in 4 leaves (or a multiple thereof) and 4 chaining values in the final node. Regarding the latter, 21 chaining values fit in exactly 4 blocks of $R = 168$ bytes. Hence, we measure the time taken to process an extra $84B = \text{lcm}(4B, 21B)$ bytes. These results are reported in Table 8.1, together with measurements on short messages.

8.5.4 512-*bit SIMD*

Recently, Intel started shipping processors with the AVX-512 instruction set. It supports 512-bit SIMD instructions, enabling efficient implementations of $8 \times$ Keccak-$p[1600, n_r = 12]$. In addition to a higher degree of parallelism, some new features of AVX-512 benefit to the implementation of KangarooTwelve, of ParallelHash and of Keccak in general.

- *Rotation instructions.* With the exception of AMD's eXtended Operations (XOP), earlier SIMD instruction sets did not include a rotation instruction. This means that the cyclic shifts in θ and

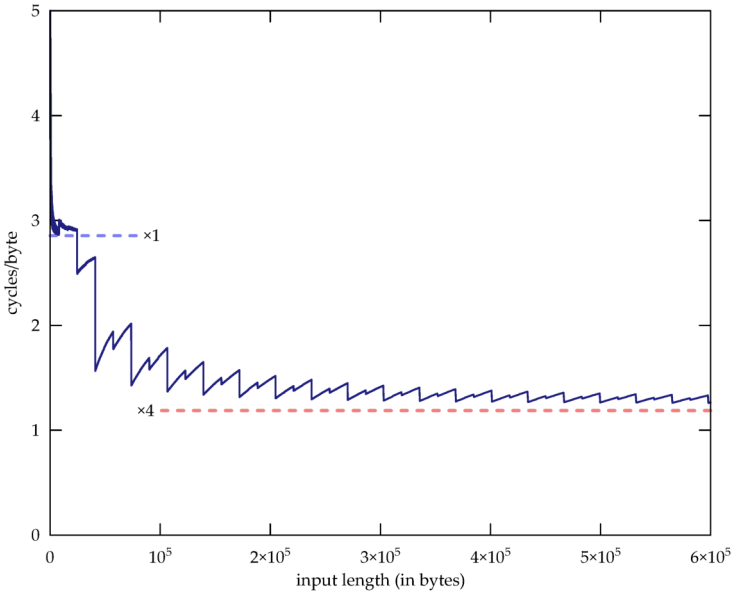


Figure 8.5: The number of cycles per byte of KangarooTwelve on an Intel Core i5-6500 (Skylake) as a function of the input message size.

ρ had to be implemented with a sequence of three instructions (shift left, shift right, XOR). With a rotation instruction, cyclic shifts are thus reduced from three to one instruction.

- *Three-input binary functions.* AVX-512 offers an instruction that produces an arbitrary bitwise function of three binary inputs. In θ, computing the parity takes four XORs, which can be reduced to two applications of this new instruction. Similarly, the non-linear function χ can benefit from it to directly compute $a_x \oplus (a_{x+1} \oplus \text{'1'}) \wedge a_{x+2}$.
- *32 registers.* Compared to AVX2, the new processors increase the number of registers from 16 to 32. As Keccak-p has 25 lanes, this significantly decreases the need to move data between memory and registers.

We report in Table 8.1 the speed of our current implementation on a machine equipped with a processor in the Intel SkylakeX family, supporting this instruction set [Ber+16].

8.5.5 *Comparison with other functions*

To put the speed of KangarooTwelve in perspective, we compare it to typical hash functions, including the traditional standards MD5, SHA-1 and SHA-2 [Riv92; NIS95; NIS02], the SHA-3 finalists [Aum+08; Gau+11; Wu11; Fer+09], the popular Blake2 functions [Aum+13] and

Table 8.1: The overall speed for very short messages ($\|S\| < 168$) in cycles, very short messages when batched in cycles/message, for short messages ($\|S\| \leq 8192$) and for long ($\|S\| \gg 8192$) messages in cycles/byte. The figures assume a single core in each case.

	Messages			
Processor Intel	Very short	Batched v.s.	Short	Long
i5-4570 (Haswell)	618 c	242 c/m	3.68 c/b	1.44 c/b
i5-6500 (Skylake)	486 c	205 c/m	2.89 c/b	1.22 c/b
i7-7800X (SkylakeX)	395 c	92 c/m	2.35 c/b	0.55 c/b

some SHA-3 instances [NIS15; NIS16]. For consistency, wherever possible we performed benchmarks on three machines in our possession. Moreover, we cross-checked with the publicly available eBACS results [BL] and in case of discrepancy, we selected the fastest. For the traditional hash functions, the fastest implementation often came from OpenSSL [Opea]. For BLAKE2, we included some specific AVX2 code by Samuel Neves [Nev]. Note that the comparison on SkylakeX must be taken with care, as not all implementations available at the time of this benchmarking are fully optimized for the AVX-512 instruction set.

Table 8.2 shows the results. We first list hash functions that explicitly exploit SIMD instructions with a built-in tree hash mode, such as PARALLELHASH and BLAKE2{B,S}P, and compare them to KANGAROOTWELVE for long messages (or when it is used for hashing multiple messages in parallel).

It is interesting to compare the other hash functions to KANGAROOTWELVE when it is restricted to serial processing (as for short messages), to see its speed gain already before the parallelism kicks in. Of course, such a restriction does not exist when hashing a large file, and in practice the comparison should also be made with KANGAROOTWELVE for long messages.

8.6 CONCLUSION

KANGAROOTWELVE can be seen as a new member of the KECCAK family. It inherits all the properties of the family such as suitability in hardware and resistance against side-channel attacks, but grew up with a strong focus on software performance and interoperability. We tuned the mode and the primitive to offer a tremendous computational speedup in many applications while keeping a comfortable security margin. The latter is confirmed by the cryptanalysis results on KECCAK accumulated over the last ten years, which are directly applicable to the new sibling. Also, all existing KECCAK implementations can be reused with minimal effort thanks to the layered approach in the design. For instance,

Table 8.2: Speed comparison. All figures are in cycles per byte for long messages, unless otherwise specified.

Function	SkylakeX	Skylake	Haswell
KangarooTwelve	0.55	1.22	1.44
KangarooTwelve ($\leq$ 8KiB)	2.35	2.89	3.68
ParallelHash128	0.96	2.31	2.73
Blake2bp	1.39	1.34	1.37
Blake2sp	1.22	1.29	1.39
shake128	4.28	5.56	7.09
MD5	4.33	4.54	4.93
SHA-1	3.05	3.07	4.15
SHA-256	6.65	6.91	9.27
SHA-512	4.44	4.64	6.54
Blake2b	2.98	3.04	3.08
Blake2s	4.26	4.85	5.34
Blake-256	5.95	6.76	7.52
Blake-512	4.48	5.19	5.68
Grøstl-256	7.24	8.13	9.35
Grøstl-512	9.95	11.31	13.51
JH	13.04	15.14	15.09
Skein	4.48	5.18	5.34

KangarooTwelve benefits immediately from the new SHA-3 hardware support recently introduced in the ARMv8.2 instruction set [ARM].

The speedup benefits to both low-end and high-end processors. For the low end, one immediately benefits from the reduction in the number of rounds, and care was taken not to add overhead in the case of short messages.

At the high end, we observed that KangarooTwelve gets significant performance improvements in recent processors, which go beyond the mere gain due to parallelism. Part of these improvements come from the choice of low-latency Boolean operations in the primitive that superscalar architectures can implement efficiently, as demonstrated in the latest Intel's SkylakeX processors with the introduction of three-input binary functions.

On such a processor, KangarooTwelve processes long messages at 0.55 cycles/byte. At this speed, it would require only one of its cores to process, in real-time, the output of 10 high-speed solid-state drives (SSD), i. e., a cumulated bandwidth of more than 7 GB/s per core (assuming a clock frequency of 4 GHz). This simply illustrates that with KangarooTwelve the speed of hashing is no longer a bottleneck in software applications.

APPENDIX OF CHAPTER 8

8.A KANGAROOTWELVE CODE

We give the Python 3 reference code of KangarooTwelve from the XKCP [Ber+18; Ber+]. Code 8.1 implements Keccak-$p[1600, n_r]$, which is then used in Code 8.2 to build the sponge function F. The length encode function and KangarooTwelve are displayed in Code 8.3.

```
def ROL64(a, n):
  return ((a >> (64-(n%65))) + (a << (n%64))) % (1 << 64)

def KeccakP1600onLanes(st, nrRounds):
  R = 1
  for round in range(24):
    if (round + nrRounds >= 24):
      # Theta
      C = [st[x][0] ^ st[x][1] ^ st[x][2] ^ st[x][3] ^ st[x][4] \
        for x in range(5)]
      D = [C[(x+4)%5] ^ ROL64(C[(x+1)%5], 1) for x in range(5)]
      st = [[st[x][y]^D[x] for y in range(5)] for x in range(5)]
      # Pi
      (x, y) = (1, 0)
      current = st[x][y]
      for t in range(24):
        (x, y) = (y, (2*x+3*y)%5)
        (current, st[x][y]) = \
          (st[x][y], ROL64(current, (t+1)*(t+2)//2))
      # Chi
      for y in range(5):
        T = [st[x][y] for x in range(5)]
        for x in range(5):
          st[x][y] = T[x] ^((~T[(x+1)%5]) & T[(x+2)%5])
      # Iota
      for j in range(7):
        R = ((R << 1) ^ ((R >> 7)*0x71)) % 256
        if (R & 2):
          st[0][0] = st[0][0] ^ (1 << ((1<<j)-1))
    else:
      for j in range(7):
        R = ((R << 1) ^ ((R >> 7)*0x71)) % 256
  return st

def load64(b):
  return sum((b[i] << (8*i)) for i in range(8))

def store64(a):
  return bytes((a >> (8*i)) % 256 for i in range(8))

def KeccakP1600(state, nrRounds):
  lanes = [[load64(state[8*(x+5*y):8*(x+5*y)+8])
        for y in range(5)] for x in range(5)]
  lanes = KeccakP1600onLanes(lanes, nrRounds)
  state = b''.join([store64(lanes[x][y])
        for y in range(5) for x in range(5)])
  return bytearray(state)
```

Code 8.1: The Keccak-$p[1600, n_r]$ permutations.

```
def F(inputBytes, delimitedSuffix , outputByteLen):
  outputBytes = b''
  state = bytearray([0 for i in range(200)])
  rateInBytes = 1344//8
  blockSize = 0
  inputOffset = 0
  # === Absorb all the input blocks ===
  while(inputOffset < len(inputBytes)):
    blockSize = min(len(inputBytes)-inputOffset, rateInBytes)
    for i in range(blockSize):
      state[i] = state[i] ^ inputBytes[i+inputOffset]
    inputOffset = inputOffset + blockSize
    if (blockSize == rateInBytes):
      state = KeccakP1600(state, 12)
      blockSize = 0
  # === Do the padding and switch to the squeezing phase ===
  state [blockSize ] = state[blockSize ] ^ delimitedSuffix
  if ((( delimitedSuffix & 0x80) != 0)
      and (blockSize == (rateInBytes-1))):
    state = KeccakP1600(state, 12)
  state [rateInBytes-1] = state[rateInBytes-1] ^ 0x80
  state = KeccakP1600(state, 12)
  # === Squeeze out all the output blocks ===
  while(outputByteLen > 0):
    blockSize = min(outputByteLen, rateInBytes)
    outputBytes = outputBytes + state[0:blockSize]
    outputByteLen = outputByteLen - blockSize
    if (outputByteLen > 0):
      state = KeccakP1600(state, 12)
  return outputBytes
```

Code 8.2: The function F.

```
def length_encode(x):
  S = b''
  while(x > 0):
    S = bytes([x % 256]) + S
    x = x//256
    S = S + bytes([len(S)])
  return S

def KangarooTwelve(inputMessage, customizationString, outputByteLen):
  B = 8192
  c = 256
  S = inputMessage + \
    customizationString + \
    length_encode(len(customizationString))
  # === Cut the input string into chunks of B bytes ===
  n = (len(S)+B-1)//B
  Si = [bytes(S[i*B:(i+1)*B]) for i in range(n)]
  if (n == 1):
    # === Process the tree with only a final node ===
    return F(Si[0], 0x07, outputByteLen)
  else:
    # === Process the tree with kangaroo hopping ===
    CVi = [F(Si[i+1], 0x0B, c//8) for i in range(n-1)]
    NodeStar = Si[0] + b'\x03\x00\x00\x00\x00\x00\x00\x00' \
      + b''.join(CVi) \
      + length_encode(n-1) + b'\xFF\xFF'
  return F(NodeStar, 0x06, outputByteLen)
```

Code 8.3: The function `length_encode` and KangarooTwelve.

9 THE IETF-IRTF STANDARDIZATION PROCESS

Designing an algorithm and ensuring its security strength does not necessarily translate into an immediate use by companies. They need to respect governments or international standards such as FIPS by the NIST or ISO by the International Organization for Standardization (ISO).

In this short chapter we first provide a brief overview of the IETF, then we describe the creation of an RFC, and we illustrate this process with timeline of the standardization of KangarooTwelve as defined in Chapter 8.

In the last section we list the RFC mentioned in this chapter, providing a quick-access reference to the reader.

9.1 THE IETF, THE IRTF, AND THE CFRG

The Internet Engineering Task Force (IETF) is the organization responsible for the Internet standards, and it is most notably known for the internet protocol suite: the Transmission Control Protocol (TCP) and the Internet Protocol (IP), defined in RFC 675, RFC 791, and RFC 793. It was originally supported by the government of the United States, and since 1993 it has been operating under the Internet Society as an international non-profit organization.

The membership is free and there are no formal requirements as the participants are all volunteers. The participation is done via mailing lists. Optionally, the members may join the meetings in person which are organized three times a year: in March, July, and November with locations rotating between America, Europe, and Asia.

While the IETF focuses on shorter term engineering issues and making standards, its sister organization the Internet Research Task Force (IRTF) works on more research oriented questions with regard to the internet, focusing mainly on protocols, architecture and new applications. In addition to the chairpersons, the governing body of the IETF —respectively IRTF— is the Internet Engineering Steering Group (IESG), respectively Internet Research Steering Group (IRSG).

Both the IETF and IRTF are organized into a large number of Working Groups (WG) —Research Groups (RG) for the IRTF— which are engaged on small focus point. For example, the working group Dynamic Host Configuration (dhc) is responsible for the protocol that assigns your IP address when connecting to a network. Similarly, the Transport Layer Security (tls) working group is responsible for 50 RFCs and has 14 active drafts at the time of this writing. Upon completion of their intended work on their topic, working groups are intended to disband.

Each working group is under the hood of a chairpersons (or multiple co-chairs) and a charter that describes its goal. While online communications and decisions are handled by email, in-person meetings use rough consensus to assert the interest of the group on a topic. Rather than using show of hands which easily identifies a person with their opinion, the process makes use of humming: a person will hum if they agree with the proposition. The chair of the session will then determine if a consensus is reached or not. This practice is described in more detail in RFC 2418 and in RFC 7282.

The working group of our interest is part of the IRTF: the Crypto Forum Research Group (CFRG). While most Working Group (WG) disband once their task is completed, this one is focused on a broader theme and aims to cover things worth standardizing that do not need to spin up work groups.

The goal of CFRG is to allow discussions and reviews about cryptographic mechanisms. It mainly focuses on network security and their use in the IETF. In the end, CFRG intends to provide informational documents to help engineers use cryptography.

9.2 WRITING AN RFC AND THE STANDARDIZATION PROCESS

Originally aimed at communications between researchers and encouraging discussions, the goal of a RFC shifted to propose standards by formalizing terms, giving guidelines, describing the best current practices, but also introducing new protocols and implementation advises.

For historical reasons, RFCs are presented in a 72-character wide text, however their writing is now done in XML and makes use of the `xml2rfc`[1] software to take care of the formatting. This change shifted a large part of the responsibility of formality and structure of the document from the reviewer and RFC editor to the authors, making it easier to work on for everyone.

The IETF also published the guidelines RFC 2360 and RFC 7322 to help authors getting used to the process of publication. Furthermore, RFC 2119 defines common keywords such as "SHOULD", "MAY", "MUST", and others to unify the requirement level across internet drafts while its update RFC 8174 aims to clarify some of the ambiguity.

While using a version control system is always advised, the recent RFC 8874 and RFC 8875 encourage the use of GitHub over GitLab or BitBucket due the large community of contributors available on the first. Additionally, the ease of enabling Continuous Integration (CI) and the ability to use *GitHub pages* to provide a preview of the compiled document strengthen that choice. To simplify the set up

1 `https://xml2rfc.tools.ietf.org/`

process, Thomson proposes a stub repository[2] providing compilation scripts and a pre-filled skeleton in XML for an RFC.

Drafts are submitted to the datatracker[3], this allows more visibility for someone casually following the conversation on the mailing list. Moreover, the philosophy is "version numbers are free", meaning that it is encouraged to update a draft quite often rather than having a low version number but with a large volume of modifications. That way readers get access to the updated version more easily.

The lifetime of a draft is six months. If not updated by then, it will be considered as expired. For this reason it is not unusual to see authors apply minor modifications and resubmit the document in order to prolong its lifetime as illustrated for example by `draft-moskowitz-ecdsa-pki`[4].

PUBLICATION OF CFRG DOCUMENTS. The IRTF publication stream is described in RFC 5743, however the document glances briefly over the details of the Research Group (RG) part. In the following, we outline the process of CFRG.

1. Alice writes a candidate document `draft-Alice-Title-00` and submits it to the datatracker. At this stage it is called a *"personal draft"*.

2. Alice continues to work on her draft and updates it multiple times; as a result the version number increases: `draft-Alice-Title-42`.

3. Alice contacts the chairs of the WG and asks for a time slot to present her draft at the CFRG session during the next IETF/IRTF meeting.

4. Other CFRG participants join the discussion on the mailing list and propose modification to the draft.

5. Alice modifies her draft with respect to the feedback she got and updates it as `draft-Alice-Title-43`.

6. Alice contacts the chairs of the RG to ask for a time slot to discuss her improvements at the next group meeting.

7. Alice requests the chairs of the RG to make an adoption call.

8. The chairs start a two-week call during which CFRG participants are asked to show their support for the draft.

9. Upon acceptance, the chairs change the status of the draft from personal to RG item. As a result the draft name

2 https://github.com/martinthomson/i-d-template
3 https://datatracker.ietf.org/submit/
4 https://datatracker.ietf.org/doc/draft-moskowitz-ecdsa-pki/

changes to `draft-irtf-cfrg-Title-00`. At that point Alice is invited to join the draft repository on Github at `github.com/cfrg/draft-irtf-cfrg-Title`. Users can propose changes to the draft directly by making pull-requests and opening issues on the repository.

Alice is then responsible for providing to the mailing list summaries of the communications happening on the repository. This is to ensure that nobody from the group is being left out for not using GitHub.

Similarly, as while working on a personal draft, the version number is incremented at each update: `draft-irtf-cfrg-Title-57`.

10. Once the document is stable, Alice requests that the chairs of the RG proceed forward. They ask feedback from the "Crypto Review Panel" which are responsible for the correctness and quality of the document.

11. After modification of the draft with respect to the feedback from the Review Panel, Alice asks the chairs for a second call which upon consensus leads to the publication of the subsequent steps described in RFC 5743, namely the IRSG review and approval, the IESG review and finally the publication by the RFC Editor.

TIMELINE OF THE KANGAROOTWELVE DRAFT. We briefly review in Table 9.1 the standardization efforts for the KangarooTwelve function presented in Chapter 8.

As illustrated by this record, we observe that the creation of an internet standard document over a well cryptanalyzed function is a long process. Additionally, it is good to keep in mind that due to the nature of how decisions by consensus are made, a low number of engagement implies that the other members agree with the draft. Indeed, people disagreeing with the document are expected to speak up or contact the chairs directly.

9.3 RFC REFERENCES

In the following, we reference the RFC mentioned in the previous sections.

RFC 675 [CDS74]: SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM
RFC 791 [Age81a]: INTERNET PROTOCOL
RFC 793 [Age81b]: TRANSMISSION CONTROL PROTOCOL
RFC 2014 [WP96]: IRTF RESEARCH GROUP GUIDELINES AND PROCEDURES
RFC 2119 [Bra97]: KEY WORDS FOR USE IN RFCS TO INDICATE REQUIREMENT LEVELS

Date	Event
August 10th, 2016	Original publication on eprint.iacr.org
June 14th, 2017	`draft-viguier-kangarootwelve-00`
July 18th, 2017	Presentation at the 99th IETF meeting.
December 12th, 2017	`draft-viguier-kangarootwelve-01`
March 19th, 2018	`draft-viguier-kangarootwelve-02`
March 19th, 2018	Presentation at the 101st IETF meeting.
July 3rd, 2018	Publication at ACNS 2018.
July 17th, 2018	Presentation at the 102st IETF meeting.
September 19th, 2018	`draft-viguier-kangarootwelve-03`
February 7th, 2019	`draft-viguier-kangarootwelve-04`
February 13th, 2019	Beginning of call for adoption by the RG.
March 13th, 2019	Draft adopted by the RG.
August 6th, 2019	`draft-irtf-cfrg-kangarootwelve-00`
January 24th, 2020	`draft-irtf-cfrg-kangarootwelve-01`
March 12th, 2020	`draft-irtf-cfrg-kangarootwelve-02`
September 1st, 2020	`draft-irtf-cfrg-kangarootwelve-03`
September 21st, 2020	`draft-irtf-cfrg-kangarootwelve-04`
October 27th, 2020	Beginning of last adoption call as an RFC.
November 11st, 2020	Presentation at FSE 2020 rump session.
February 19st, 2021	`draft-irtf-cfrg-kangarootwelve-05`

Table 9.1: Timeline of the KangarooTwelve Internet-Draft.

RFC 2360 [Rfca]: GUIDE FOR INTERNET STANDARDS WRITERS
RFC 2418 [Rfcb]: IETF WORKING GROUP GUIDELINES AND PROCEDURES
RFC 5743 [Falo9]: DEFINITION OF AN INTERNET RESEARCH TASK FORCE (IRTF) DOCUMENT STREAM
RFC 7282 [Res14]: ON CONSENSUS AND HUMMING IN THE IETF
RFC 7322 [FG14]: RFC STYLE GUIDE
RFC 7776 [RF16]: IETF ANTI-HARASSMENT PROCEDURES
RFC 8174 [Lei17]: AMBIGUITY OF UPPERCASE VS LOWERCASE IN RFC 2119 KEY WORDS
RFC 8729 [Rfcc]: THE RFC SERIES AND RFC EDITOR
RFC 8874 [TS20]: WORKING GROUP GITHUB USAGE GUIDANCE
RFC 8875 [CH20]: WORKING GROUP GITHUB ADMINISTRATION

Part V

APPENDIX

[ARM] ARM corporation. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. document ARM DDI 0487C.a (ID121917), http://www.arm.com/ (cited on page 246).

[Age81a] Defense Advanced Research Projects Agency. *RFC 791 – INTERNET PROTOCOL*. https://datatracker.ietf.org/doc/html/rfc791. Sept. 1981 (cited on page 252).

[Age81b] Defense Advanced Research Projects Agency. *RFC 793 – TRANSMISSION CONTROL PROTOCOL*. https://datatracker.ietf.org/doc/html/rfc793. Sept. 1981 (cited on page 252).

[AlF+13] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. "On the Security of RC4 in TLS." In: *USENIX Security Symposium 2013*. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan. USENIX Association, 2013, pp. 305–320 (cited on pages 162, 170).

[Alk+16] 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. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_alkim.pdf. USA: USENIX Association, 2016, pp. 327–343 (cited on page 133).

[Alk+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. "ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (June 2020). https://tches.iacr.org/index.php/TCHES/article/view/8589, pp. 219–242 (cited on page 115).

[Alm+16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying Constant-Time Implementations." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_almeida.pdf. Association for Computing Machinery, 2016, pp. 53–70 (cited on page 183).

[App11] Andrew W. Appel. "Verified Software Toolchain." In: *Programming Languages and Systems*. Ed. by Gilles Barthe. Vol. 6602. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-642-19718-5_1. Berlin, Heidelberg: Springer, 2011, pp. 1–17 (cited on page 50).

[App12] Andrew W. Appel. "Verified Software Toolchain." In: *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Ed. by Alwyn Goodloe and Suzette Person. Vol. 7226. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-642-28891-3_2. Springer, 2012, p. 2 (cited on page 181).

[App15] Andrew W. Appel. "Verification of a Cryptographic Primitive: SHA-256." In: *ACM Transactions on Programming Languages and Systems* 37.2 (2015). http://doi.acm.org/10.1145/2701415, 7:1–7:31 (cited on pages 184, 213).

[App+14] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. https://dl.acm.org/doi/book/10.5555/2670099. USA: Cambridge University Press, 2014 (cited on page 50).

[App21] Andrew Appel. *[Coq-Club] Proof automation for large terms?* Posting to Coq-club mailing list. https://sympa.inria.fr/sympa/arc/coq-club/2021-02/msg00023.html. Feb. 2021 (cited on page 199).

[Asc] *Ascon C repository on GitHub*. https://github.com/ascon/ascon-c (cited on page 124).

[AR16] Tomer Ashur and Vincent Rijmen. "On Linear Hulls and Trails." In: *Progress in Cryptology – INDOCRYPT 2016*. Ed. by Orr Dunkelman and Somitra Kumar Sanadhya. Vol. 10095. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-49890-4_15, see also https://eprint.iacr.org/2016/088. Springer, 2016, pp. 269–286 (cited on page 152).

[AK16] Gilles Van Assche and Ronny Van Keer. *Structuring and optimizing Keccak software*. http://ccccspeed.win.tue.nl/papers/KeccakSoftware.pdf. 2016 (cited on page 91).

[Aum+08] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. *SHA-3 proposal BLAKE*. Submission to NIST. http://131002.net/blake/blake.pdf. 2008 (cited on page 244).

[AJN14a] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. "Analysis of NORX: Investigating Differential and Rotational Properties." In: *Progress in Cryptology – LATINCRYPT 2014*. Ed. by Diego F. Aranha and Alfred Menezes. Vol. 8895. Lecture Notes in Computer Science (LNCS). https://eprint.iacr.org/2014/317.pdf. Springer, 2014, pp. 306–324 (cited on pages 84, 86).

[AJN14b] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. "NORX: Parallel and Scalable AEAD." In: *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*. Ed. by Miroslaw Kutylowski and Jaideep Vaidya. Vol. 8713. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-11212-1_2. Springer, 2014, pp. 19–36 (cited on page 76).

[AKM12] Jean-Philippe Aumasson, Simon Knellwolf, and Willi Meier. "Heavy Quark for secure AEAD." In: *DIAC 2012: Directions in Authenticated Ciphers*. https://www.aumasson.jp/data/papers/AKM12.pdf. 2012 (cited on page 77).

[AM09] Jean-Philippe Aumasson and Willi Meier. *Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi*. Available online. http://131002.net/data/papers/AM09.pdf. 2009 (cited on page 239).

[Aum+14] Jean-Philippe Aumasson, Willi Meier, Raphael C.-W. Phan, and Luca Henzen. *The Hash Function BLAKE*. https://doi.org/10.1007/978-3-662-44757-4. 2014 (cited on page 78).

[Aum+13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. "BLAKE2: Simpler, Smaller, Fast as MD5." In: *Applied Cryptography and Network Security: 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*. Ed. by Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-642-38980-1_8. Springer, 2013, pp. 119–135 (cited on page 244).

[Bak+20] Anubhab Baksi, Jakub Breier, Xiaoyang Dong, and Chen Yi. "Machine Learning Assisted Differential Distinguishers For Lightweight Ciphers." In: *IACR Cryptology ePrint Archive* 2020 (2020). https://eprint.iacr.org/2020/571, p. 571 (cited on page 100).

[Bal+12] Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoit Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. *Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices*. Cryptology ePrint Archive: Report 2012/507. https://eprint.iacr.org/2012/507/. 2012 (cited on page 90).

[Bar+19] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. *SoK: Computer-Aided Cryptography*. Cryptology ePrint Archive, Report 2019/1393. https://eprint.iacr.org/2019/1393. 2019 (cited on page 183).

[BS14] Evmorfia-Iro Bartzia and Pierre-Yves Strub. "A Formal Library for Elliptic Curves in the Coq Proof Assistant." In: *Interactive Theorem Proving*. Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. Lecture Notes in Computer Science (LNCS). https://hal.inria.fr/hal-01102288. Springer, 2014, pp. 77–92 (cited on pages 182, 201, 214).

[Bei+20] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. "Lightweight AEAD and Hashing using the Sparkle Permutation Family." In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (June 2020). https://tosc.iacr.org/index.php/ToSC/article/view/8627, pp. 208–261 (cited on page 121).

[Bel11] Steven M. Bellovin. "Frank Miller: Inventor of the One-Time Pad." In: *Cryptologia* 35.3 (July 2011). https://doi.org/10.1080/01611194.2011.583711, pp. 203–222 (cited on page 4).

[Ber+15a] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. "Verified Correctness and Security of OpenSSL HMAC." In: *Proceedings of the 24th USENIX Security Symposium*. https://www.cs.cmu.edu/~kqy/resources/verified-hmac.pdf. USENIX Association, 2015, pp. 207–221 (cited on page 184).

[Ber05a] Daniel J. Bernstein. *Cache-timing attacks on AES*. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 2005 (cited on page 132).

[Ber05b] Daniel J. Bernstein. "The Poly1305-AES Message-Authentication Code." In: *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*. Ed. by Henri Gilbert and

Helena Handschuh. Vol. 3557. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/11502760_3. Springer, Feb. 2005, pp. 32–49 (cited on page 25).

[Ber06a] Daniel J. Bernstein. "Curve25519: new Diffie-Hellman speed records." In: *Public Key Cryptography – PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Lecture Notes in Computer Science (LNCS). https://cr.yp.to/papers.html#curve25519. Springer, 2006, pp. 207–228 (cited on pages 5, 12).

[Ber06b] Daniel J. Bernstein. "Curve25519: new Diffie-Hellman speed records." In: *Public Key Cryptography – PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Lecture Notes in Computer Science (LNCS). http://cr.yp.to/papers.html#curve25519. Springer, 2006, pp. 207–228 (cited on pages 181, 182, 184, 210, 214).

[Ber08a] Daniel J. Bernstein. *25519 naming*. Posting to the CFRG mailing list. https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html. Aug. 2008 (cited on pages 181, 185).

[Ber08b] Daniel J. Bernstein. *ChaCha, a variant of Salsa20*. SASC 2008: The State of the Art of Stream Ciphers. https://cr.yp.to/chacha/chacha-20080128.pdf. 2008 (cited on pages 19, 67).

[Ber08c] Daniel J. Bernstein. "The Salsa20 Family of Stream Ciphers." In: *New Stream Cipher Designs - The eSTREAM Finalists*. Ed. by Matthew J. B. Robshaw and Olivier Billet. Vol. 4986. Lecture Notes in Computer Science (LNCS). https://cr.yp.to/snuffle/salsafamily-20071225.pdf. Springer, 2008, pp. 84–97 (cited on page 67).

[Ber+15b] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. "TweetNaCl: A crypto library in 100 tweets." In: *Progress in Cryptology – LATINCRYPT 2014*. Ed. by Diego Aranha and Alfred Menezes. Vol. 8895. Lecture Notes in Computer Science (LNCS). http://cryptojedi.org/papers/#tweetnacl. Springer, 2015, pp. 64–83 (cited on page 181).

[Ber+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. "The SPHINCS+ Signature Framework." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. https://dl.acm.org/doi/10.1145/3319535.3363229. New York, NY, USA: Association for Computing Machinery, 2019, pp. 2129–2146 (cited on page 133).

[Ber+17a] 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*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-66787-4_15, see also https://eprint.iacr.org/2017/630. Springer, 2017, pp. 299–320 (cited on pages 119, 120, 129).

[BL] Daniel J. Bernstein and Tanja Lange. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. https://bench.cr.yp.to (cited on pages 91, 245).

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. "The security impact of a new cryptographic library." In: *Progress in Cryptology – LATINCRYPT 2012*. Ed. by Alejandro Hevia and Gregory Neven. Vol. 7533. Lecture Notes in Computer Science (LNCS). http://cryptojedi.org/papers/#coolnacl. Springer, 2012, pp. 159–176 (cited on page 181).

[BS08] Daniel J. Bernstein and Peter Schwabe. "New AES Software Speed Records." In: *Progress in Cryptology - INDOCRYPT 2008*. Ed. by Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das. Lecture Notes in Computer Science (LNCS). https://www.cryptojedi.org/papers/aesspeed-20080926.pdf. Berlin, Heidelberg: Springer, 2008, pp. 322–336 (cited on page 132).

[BS12] Daniel J. Bernstein and Peter Schwabe. "NEON crypto." In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science (LNCS). https://cryptojedi.org/papers/#neoncrypto. Springer, 2012, pp. 320–339 (cited on pages 67, 98).

[Ber+07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Sponge functions." In: *ECRYPT Hash Workshop*. Vol. 2007. https://keccak.team/files/SpongeFunctions.pdf. 2007 (cited on pages 27, 235).

[Ber+08a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "On the Indifferentiability of the Sponge Construction." In: *Advances in Cryptology – EUROCRYPT 2008*. Lecture Notes in Computer Science (LNCS). http://dx.doi.org/10.1007/978-3-540-78967-3_11. Springer, 2008, pp. 181–197 (cited on pages 27, 81, 231, 236, 237).

[Ber+08b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. KECCAK *specifications*. NIST SHA-3 Submission. https://keccak.team/files/Keccak-submission-3.pdf. Oct. 2008 (cited on pages 32, 231).

[Ber+11a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Cryptographic sponge functions*. http://sponge.noekeon.org/CSF-0.1.pdf. Jan. 2011 (cited on pages 19, 28, 81).

[Ber+11b] 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 - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*. Ed. by Ali Miri and Serge Vaudenay. Vol. 7118. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-642-28496-0_19. Springer, 2011, pp. 320–337 (cited on pages 31, 82).

[Ber+11c] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "On the security of the keyed sponge construction." In: *Symmetric Key Encryption Workshop – SKEW 2011*. http://skew2011.mat.dtu.dk/proceedings/On%20the%20security%20of%20the%20keyed%20sponge%20construction.pdf. 2011 (cited on page 31).

[Ber+11d] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The* KECCAK *reference*. http://keccak.noekeon.org/. Jan. 2011 (cited on page 32).

[Ber+12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Permutation-based encryption, authentication and authenticated encryption." In: *DIAC 2012 – Directions in Authenticated Ciphers*. http://www.hyperelliptic.org/DIAC/slides/PermutationDIAC2012.pdf. July 2012 (cited on page 32).

[Ber+13a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Keccak." In: *Advances in Cryptology – EUROCRYPT 2013*. Lecture Notes in Computer Science (LNCS). http://dx.doi.org/10.1007/978-3-642-38348-9_19, see also http://keccak.noekeon.org/Keccak-slides-at-Eurocrypt-May2013.pdf. 2013, pp. 313–314 (cited on page 67).

[Ber+13b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Keccak." In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-

3-642-38348-9_19. Berlin, Heidelberg: Springer, 2013, pp. 313–314 (cited on pages 129, 133).

[Ber+14a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Sakura: A Flexible Coding for Tree Hashing." In: *ACNS*. Lecture Notes in Computer Science (LNCS). http://dx.doi.org/10.1007/978-3-319-07536-5_14. Springer, 2014, pp. 217–234 (cited on pages 231, 232, 236, 237).

[Ber+14b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Sufficient conditions for sound tree and sequential hashing modes." In: *International Journal of Information Security* 13 (2014). http://dx.doi.org/10.1007/s10207-013-0220-y, pp. 335–353 (cited on pages 236, 237).

[Ber+] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *XKCP-eXtended Keccak Code Package*. https://github.com/XKCP/XKCP (cited on page 247).

[Ber+13c] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Keccak implementation overview*. https://keccak.team/files/Keccak-implementation-3.2.pdf. 2013 (cited on pages 129, 133).

[Ber+16] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. KECCAK *code package*. https://github.com/gvanas/KeccakCodePackage. June 2016 (cited on pages 240, 242, 244).

[Ber+17b] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. KECCAK *third-party cryptanalysis*. https://keccak.team/third_party.html. 2017 (cited on page 231).

[Ber+18] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *XKCP-extracted code for KangarooTwelve*. https://github.com/XKCP/K12. 2018 (cited on page 247).

[Bey+19] Tim Beyne, Yu Long Chen, Christof Dobraunig, and Bart Mennink. *Elephant v1*. https://www.esat.kuleuven.be/cosic/elephant/. 2019 (cited on page 126).

[Bih04] Eli Biham. *Tutorial on Differential Cryptanalysis*. http://www.cs.technion.ac.il/~cs236506/04/slides/differentialcrypt-tutor-add.pdf. Feb. 2004 (cited on page 39).

[Bih05] Eli Biham. *Tutorial on Differential Cryptanalysis*. http://www.cs.technion.ac.il/~cs236506/04/slides/crypto-slides-19-dc-tutor.1x1.pdf. May 2005 (cited on page 39).

[BS90] Eli Biham and Adi Shamir. "Differential Cryptanalysis of DES-like Cryptosystems." In: *Advances in Cryptology-CRYPTO' 90*. Ed. by Alfred J. Menezes and Scott A. Vanstone. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/3-540-38424-3_1. Berlin, Heidelberg: Springer, 1990, pp. 2–21 (cited on pages 36, 37, 39).

[BS91] Eli Biham and Adi Shamir. "Differential cryptanalysis of DES-like cryptosystems." In: *Journal of CRYPTOLOGY*. Vol. 4. Lecture Notes in Computer Science (LNCS) 1. https://link.springer.com/article/10.1007/BF00630563. Springer, 1991, pp. 3–72 (cited on page 36).

[BS92] Eli Biham and Adi Shamir. "Differential Cryptanalysis of the Full 16-Round DES." In: *Advances in Cryptology – CRYPTO '92*. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/3-540-48071-4_34. Berlin, Heidelberg: Springer, 1992, pp. 487–496 (cited on pages 36, 39).

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. "Biclique Cryptanalysis of the Full AES." In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-642-25385-0_19. Berlin, Heidelberg: Springer, 2011, pp. 344–371 (cited on page 26).

[Bog+11] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. "SPONGENT: The Design Space of Lightweight Cryptographic Hashing." In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science (LNCS). https://eprint.iacr.org/2011/697. Springer, 2011, pp. 312–321 (cited on pages 77, 91).

[BM06] Joseph Bonneau and Ilya Mironov. "Cache-Collision Timing Attacks against AES." In: *CHES'06 – Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/11894063_16. Berlin, Heidelberg: Springer, 2006, pp. 201–215 (cited on page 132).

[Bor] *BoringSSL*. https://boringssl.googlesource.com/ (cited on page 8).

[Bos+16] Joppe Bos, Craig Costello, Leo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. "Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. https://dl.acm.org/doi/abs/10.1145/2976749.2978425. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1006–1018 (cited on page 133).

[BCC11] Christina Boura, Anne Canteaut, and Christophe De Cannière. "Higher-order differential properties of Keccak and Luffa." In: *Fast Software Encryption 2011*. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-642-21702-9_15, see also https://eprint.iacr.org/2010/589. Springer, 2011 (cited on page 239).

[Bra97] Scott Bradner. *RFC 2119 – Key words for use in RFCs to Indicate Requirement Levels*. https://datatracker.ietf.org/doc/html/rfc2119. Mar. 1997 (cited on page 252).

[Bro07a] Luitzen Egbertus Jan Brouwer. "On the foundations of mathematics." In: *Collected works* 1 (1907). Translation, pp. 11–101 (cited on page 44).

[Bro07b] Luitzen Egbertus Jan Brouwer. *Over de Grondslagen der Wiskunde*. Ed. by. Amsterdam: Maas & van Suchtelen, 1907 (cited on page 44).

[Bru+12] Billy B. Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. "Practical Realisation and Elimination of an ECC-Related Software Bug Attack." In: *Topics in Cryptology – CT-RSA 2012*. Ed. by Orr Dunkelman. https://doi.org/10.1007/978-3-642-27954-6_11. Berlin, Heidelberg: Springer, 2012, pp. 171–186 (cited on page 8).

[Bur14] Elie Bursztein. *Speeding up and strengthening HTTPS connections for Chrome on Android*. https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html. 2014 (cited on page 67).

[CAE13] CAESAR Committee. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. Call for Submissions*. http://competitions.cr.yp.to/caesar-call.html. 2013 (cited on page 143).

[Can+20] Anne Canteaut, Sébastien Duval, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Thomas Pornin, and André Schrottenloher. "Saturnin: a suite of lightweight symmetric algorithms for post-quantum security." In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (June 2020). https://tosc.iacr.org/index.php/ToSC/article/view/8621, pp. 160–207 (cited on pages 122, 123).

[Cao+18] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs." In: *Journal of Automated Reasoning* 61.1–4 (June 2018). https://doi.org/10.1007/s10817-018-9457-5, pp. 367–422 (cited on pages 50, 190, 196).

[Cau+20] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. "Constant-time foundations for the new spectre era." In: *Programming Language Design and Implementation – 2020*. https://dl.acm.org/doi/10.1145/3385412.3385970. Association for Computing Machinery, 2020, pp. 913–926 (cited on page 182).

[CDS74] Vinton Cerf, Yogen Dalal, and Carl Sunshine. *RFC 675 – SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM*. https://datatracker.ietf.org/doc/html/rfc675. Dec. 1974 (cited on page 252).

[Cha+18] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. "Beetle Family of Lightweight and Secure Authenticated Encryption Ciphers." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (May 2018). https://tches.iacr.org/index.php/TCHES/article/view/881, pp. 218–241 (cited on pages 83, 121).

[Che+14] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. "Verifying Curve25519 Software." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. https://cryptojedi.org/papers/#verify25519.pdf. Association for Computing Machinery, 2014, pp. 299–309 (cited on page 184).

[Chl10] Adam Chlipala. "An Introduction to Programming and Proving with Dependent Types in Coq." In: *Journal of Formalized Reasoning* 3(2) (2010). http://adam.chlipala.net/cpdt/, pp. 1–93 (cited on pages 184, 199).

[CH20] Alissa Cooper and Paul Hoffman. *RFC 8875 – Working Group GitHub Administration*. https://datatracker.ietf.org/doc/html/rfc8875. Aug. 2020 (cited on page 253).

[CH88] Thierry Coquand and Gérard Huet. "The calculus of constructions." In: *Information and Computation* 76.2 (1988). https://doi.org/10.1016/0890-5401(88)90005-3, pp. 95 –120 (cited on page 45).

[CS18] Craig Costello and Benjamin Smith. "Montgomery curves and their arithmetic: The case of large characteristic fields." In: *Journal of Cryptographic Engineering* 8.3 (2018). https://eprint.iacr.org/2017/212 (cited on page 186).

[Cru+16] Rafael Cruz, Tiago Reis, Diego F. Aranha, and Harsh Kupwade Patil. "Lightweight cryptography on ARM." In: *NIST Lightweight Cryptography Workshop. NIST*. http://www.africacrypt.com/presentations/lw-arm-speed.pdf. 2016 (cited on page 115).

[Dae95] Joan Daemen. "Cipher and hash function design, strategies based on linear and differential cryptanalysis, PhD Thesis." https://cs.ru.nl/~joan/papers/JDA_Thesis_1995.pdf. PhD thesis. K.U.Leuven, 1995 (cited on pages 37, 148, 149).

[Dae+18a] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. "The design of Xoodoo and Xoofff." In: *IACR Transactions on Symmetric Cryptology* 2018.4 (Dec. 2018). https://tosc.iacr.org/index.php/ToSC/article/view/7359, pp. 1–38 (cited on page 129).

[Dae+18b] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Xoodoo cookbook*. Cryptology ePrint Archive, Report 2018/767. https://eprint.iacr.org/2018/767. 2018 (cited on page 129).

[Dae+20] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. "Xoodyak, a lightweight cryptographic scheme." In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (June 2020). https://tosc.iacr.org/index.php/ToSC/article/view/8618, pp. 60–87 (cited on page 129).

[DMA17] Joan Daemen, Bart Mennink, and Gilles Van Assche. "Full-State Keyed Duplex with Built-In Multi-user Support." In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-70697-9_21, see also https://eprint.iacr.org/2017/498. Springer, 2017, pp. 606–637 (cited on page 32).

[DR02] Joan Daemen and Vincent Rijmen. "The Design of Rijndael: AES - The Advanced Encryption Standard." In: *Information Security and Cryptography*. https://www.springer.com/gp/book/9783540425809. Springer, 2002 (cited on pages 5, 20, 132).

[DH76] Whitfield Diffie and Martin Hellman. "New Directions in Cryptography." In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976). https://doi.org/10.1109/TIT.1976.1055638, pp. 644–654 (cited on page 5).

[Din+16a] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. "Design Strategies for ARX with Provable Bounds: SPARX and LAX." In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science (LNCS). https://eprint.iacr.org/2016/984.pdf. Springer, 2016, pp. 484–513 (cited on page 79).

[Din+16b] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. "Design Strategies for ARX with Provable Bounds: Sparx and LAX." In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-662-53887-6_18. Berlin, Heidelberg: Springer, 2016, pp. 484–513 (cited on page 121).

[DDS13] Itai Dinur, Orr Dunkelman, and Adi Shamir. "Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials." In: *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*. Lecture Notes in Computer Science (LNCS). http://dx.doi.org/10.1007/978-3-662-43933-3_12. Springer, 2013, pp. 219–240 (cited on page 231).

[DDS14] Itai Dinur, Orr Dunkelman, and Adi Shamir. "Improved Practical Attacks on Round-Reduced Keccak." In: *Journal of Cryptology*. Vol. 27. Lecture Notes in Computer Science (LNCS) 2. http://dx.doi.org/10.1007/s00145-012-9142-5. Springer, 2014, pp. 183–209 (cited on page 231).

[Din+15] Itai Dinur, Pawel Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michal Straus. "Cube Attacks and Cube-Attack-Like Cryptanalysis on the Round-Reduced Keccak Sponge Function." In: *Advances in Cryptology - EUROCRYPT 2015*. Lecture Notes in Computer Science (LNCS). http://dx.doi.org/10.1007/978-3-662-46800-5_28. Springer, 2015, pp. 733–761 (cited on page 239).

[DEM15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. "Analysis of SHA-512/224 and SHA-512/256." In: *Advances in Cryptology - ASIACRYPT*. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-662-48800-3_25. Springer, 2015, pp. 612–630 (cited on page 239).

[Dob+16a] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1. 2*. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf. 2016 (cited on page 124).

[Dob+16b] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2*. Submission to the CAESAR competition: https://competitions.cr.yp.to/round3/asconv12.pdf. http://ascon.iaik.tugraz.at. 2016 (cited on pages 75, 91).

[DR11] Thai Duong and Juliano Rizzo. *Here Come The ⊕ Ninjas*. Ekoparty. https://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf. 2011 (cited on page 162).

[Dwi+16] Ashutosh Dhar Dwivedi, Miloš Klouček, Pawel Morawiecki, Ivica Nikolić, Josef Pieprzyk, and Sebastian Wójtowicz. *SAT-based Cryptanalysis of Authenticated Ciphers from the CAESAR Competition*. Cryptology ePrint Archive, Report 2016/1053. https://eprint.iacr.org/2016/1053. 2016 (cited on page 144).

[DMW17] Ashutosh Dhar Dwivedi, Pawel Morawiecki, and Sebastian Wójtowicz. "Differential and Rotational Cryptanalysis of Round-reduced MORUS." In: *E-Business and Telecommunications – ICETE/SECRYPT 2017*. Ed. by Pierangela Samarati, Mohammad S. Obaidat, and Enrique Cabello. https://doi.org/10.5220/0006411502750284. SciTePress, 2017, pp. 275–284 (cited on page 144).

[Dwo15] Morris J. Dworkin. *FIPS 202: SHA-3 standard: Permutation-Based Hash and Extendable-Output Functions*. Tech. rep. https://doi.org/10.6028/NIST.FIPS.202. National Institute of Standards and Technology, 2015 (cited on page 133).

[Erb17] Andres Erbsen. "Crafting certified elliptic curve cryptography implementations in Coq." http://adam.chlipala.net/theses/andreser_meng.pdf. MA thesis. Massachusetts Institute of Technology, 2017 (cited on pages 8, 184).

[Erb+16] Andres Erbsen, Jade Philipoom, Jason Gross, Rober Sloan, and Adam Chlipala. *Systematic Synthesis of Elliptic Curve Cryptography Implementations*. https://people.csail.mit.edu/jgross/personal-website/papers/2017-fiat-crypto-pldi-draft.pdf. 2016 (cited on pages 8, 184, 214).

[Erb+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises." In: *2019 IEEE Symposium on Security and Privacy*. https://people.csail.mit.edu/jgross/personal-website/papers/2019-fiat-crypto-ieee-sp.pdf. 2019, pp. 73–90 (cited on pages 8, 184).

[Fal09] Aaron Falk. *RFC 5743 – Definition of an Internet Research Task Force (IRTF) Document Stream*. https://datatracker.ietf.org/doc/html/rfc5743. Dec. 2009 (cited on page 253).

[Fei73] Horst Feistel. "Cryptography and data security." In: *Scientific american* 228.5 (1973), pp. 15–23 (cited on page 4).

[Fer+09] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. *The Skein Hash Function Family*. Submission to NIST (Round 2). http://www.skein-hash.info/sites/default/files/skein1.2.pdf. 2009 (cited on page 244).

[FG14] Heather Flanagan and Sandy Ginoza. *RFC 7322 – RFC Style Guide*. https://datatracker.ietf.org/doc/html/rfc7322. Sept. 2014 (cited on page 253).

[Fló+20] Antonio Flórez-Gutiérrez, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, André Schrottenloher, and Ferdinand Sibleyras. "New Results on Gimli: Full-Permutation Distinguishers and Improved Collisions." In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-030-64837-4_2. Springer, 2020, pp. 33–63 (cited on pages 99, 100).

[Flo67] Robert W. Floyd. "Assigning meaning to programs." In: *Program Verification*. https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf. 1967 (cited on page 46).

[Flo93] Robert W. Floyd. "Assigning Meanings to Programs." In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. https://doi.org/10.1007/978-94-011-1793-7_4. Dordrecht: Springer Netherlands, 1993, pp. 65–81 (cited on page 46).

[FLG98] Electronic Frontier Foundation, Mike Loukides, and John Gilmore. *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*. http://cryptome.org/jya/cracking-des/cracking-des.htm. USA: O'Reilly & Associates, Inc., July 1998 (cited on pages 4, 36).

[FJM14] Pierre-Alain Fouque, Antoine Joux, and Chrysanthi Mavromati. "Multi-user Collisions: Applications to Discrete Logarithm, Even-Mansour and PRINCE." In: *Advances in Cryptology – ASIACRYPT 2014*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-662-45611-8_22, see also https://eprint.iacr.org/2013/761.pdf. Springer, 2014, pp. 420–438 (cited on page 77).

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepulveda. *RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography*. Cryptology ePrint Archive, Report 2020/446. https://eprint.iacr.org/2020/446. 2020 (cited on page 115).

[Gau+11] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. *Grøstl – a SHA-3 candidate*. Submission to NIST (round 3). http://www.groestl.info/Groestl.pdf. 2011 (cited on page 244).

[Gon08] Georges Gonthier. "Formal proof—the four-color theorem." In: *Notices of the AMS* 55.11 (2008). https://www.ams.org/notices/200811/tx081101382p.pdf, pp. 1382–1393 (cited on page 190).

[GMT16] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR–6455. https://hal.inria.fr/inria-00258384. Inria Saclay Ile de France, 2016 (cited on page 199).

[Gro96] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search." In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. https://doi.org/10.1145/237814.237866. New York, NY, USA: Association for Computing Machinery, 1996, pp. 212–219 (cited on page 6).

[Gro97] Lov K. Grover. "Quantum Mechanics Helps in Searching for a Needle in a Haystack." In: *Physical Review Letters* 79.2 (July 1997). http://dx.doi.org/10.1103/PhysRevLett.79.325, pp. 325–328 (cited on page 6).

[GLS16] Jian Guo, Meicheng Liu, and Ling Song. "Linear Structures: Applications to Cryptanalysis of Round-Reduced Keccak." In: *Advances in Cryptology - ASIACRYPT*. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-662-53887-6_9. Springer, 2016, pp. 249–274 (cited on pages 238, 239).

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann. "The PHOTON Family of Lightweight Hash Functions." In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-642-22792-9_13. Springer, 2011, pp. 222–239 (cited on page 77).

[Ham15] Mike Hamburg. *Ed448-Goldilocks, a new elliptic curve*. Cryptology ePrint Archive, Report 2015/625. https://eprint.iacr.org/2015/625. 2015 (cited on page 214).

[Hel02] Martin E. Hellman. "An overview of public key cryptography." In: *IEEE Communications Magazine* 40.5 (2002). https://doi.org/10.1109/MCOM.2002.1006971, pp. 42–49 (cited on page 5).

[Hoa69] Sir Charles Antony Richard Hoare. "An Axiomatic Basis for Computer Programming." In: *Association for Computing Machinery* 12.10 (Oct. 1969). http://doi.acm.org/10.1145/363235.363259, pp. 576–580 (cited on pages 46, 181).

[How95] William Alvin Howard. "The Formulæ-as-Types Notion of Construction." In: *The Curry-Howard Isomorphism*. Ed. by Philippe De Groote. https://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf. Academia, 1995 (cited on pages 44, 190).

[Hua+17] Senyang Huang, Xiaoyun Wangand Guangwu Xu, Meiqin Wang, and Jingyuan Zhao. "Conditional Cube Attack on Reduced-Round Keccak Sponge Function." In: *Advances in Cryptology - EUROCRYPT 2017*. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-56614-6_9. SV, 2017, pp. 259–288 (cited on page 239).

[HRS16] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. "ARMed SPHINCS – Computing a 41KB signature in 16KB of RAM." In: *Public Key Cryptography – PKC 2016*. Ed. by Giuseppe Persiano and Bo-Yin Yang. Vol. 9614. Lecture Notes in Computer Science (LNCS). https://cryptojedi.org/papers/#armedsphincs. Springer, 2016, pp. 446–470 (cited on page 91).

[HS13] Michael Hutter and Peter Schwabe. "NaCl on 8-bit AVR Microcontrollers." In: *Progress in Cryptology – AFRICACRYPT 2013*. Ed. by Amr Youssef and Abderrahmane Nitaj. Vol. 7918. Lecture Notes in Computer Science (LNCS). https://cryptojedi.org/papers/#avrnacl. Springer, 2013, pp. 156–172 (cited on page 91).

[IS] Ltd Imperas Software. *riscvOVPsim simulator*. https://github.com/riscv/riscv-ovpsim (cited on page 119).

[KEM17] Daniel Kales, Maria Eichlseder, and Florian Mendel. *Note on the Robustness of CAESAR Candidates*. IACR Cryptology ePrint Archive, Report 2017/1137. https://eprint.iacr.org/2017/1137. 2017 (cited on pages 144, 167).

[KS09] Emilia Käsper and Peter Schwabe. "Faster and Timing-Attack Resistant AES-GCM." In: *Cryptographic Hardware and Embedded Systems – CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-642-04138-9_1. Berlin, Heidelberg: Springer, 2009, pp. 1–17 (cited on page 133).

[Ker83] Auguste Kerckhoffs. "La cryptographie militaire." In: *Journal des sciences militaires*. Vol. 9. https://www.petitcolas.net/kerckhoffs/index.html. Paris: Librairie militaire de L. Baudoin, 1883, pp. 5–38 (cited on page 3).

[Kob87] Neal Koblitz. "Elliptic Curve Cryptosystems." In: *Association for Computing Machinery* 48.177 (Jan. 1987). https://doi.org/10.2307/2007884, pp. 203–209 (cited on page 5).

[KLT15] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. "Observations on the SIMON Block Cipher Family." In: *Advances in Cryptology – CRYPTO 2015*. Ed. by Rosario Gennaro and Matthew Robshaw. Vol. 9215. Lecture Notes in Computer Science (LNCS). https://eprint.iacr.org/2015/145.pdf. Springer, 2015, pp. 161–185 (cited on pages 84, 86).

[Köno8] Robert Könighofer. "A Fast and Cache-Timing Resistant Implementation of the AES." In: *Topics in Cryptology – CT-RSA 2008*. Ed. by Tal Malkin. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-540-79263-5_12. Berlin, Heidelberg: Springer, 2008, pp. 187–202 (cited on page 133).

[KR14] Ted Krovetz and Phillip Rogaway. "RFC 7253 – The OCB Authenticated-Encryption Algorithm." In: *RFC 7253* (2014). https://doi.org/10.17487/RFC7253, pp. 1–19 (cited on page 143).

[LMM91] Xuejia Lai, James L. Massey, and Sean Murphy. "Markov Ciphers and Differential Cryptanalysis." In: *EUROCRYPT'91 – Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*. Lecture Notes in Computer Science (LNCS) 547. https://doi.org/10.1007/3-540-46416-6_2. Berlin, Heidelberg: Springer, 1991, pp. 17–38 (cited on page 38).

[LHT] Adam Langley, Mike Hamburg, and Sean Turner. *RFC 7748 – Elliptic Curves for Security*. https://tools.ietf.org/html/rfc7748 (cited on pages 5, 12, 181, 185, 187, 190, 192, 193, 214).

[Lan+] Adam Langley, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. *RFC 7748 – ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*. https://tools.ietf.org/html/rfc7905 (cited on page 67).

[Lei17] Barry Leiba. *RFC 8174 – Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*. https://datatracker.ietf.org/doc/html/rfc8174. May 2017 (cited on page 253).

[Ler09a] Xavier Leroy. "A formally verified compiler back-end." In: *Journal of Automated Reasoning*. Vol. 43. 4. http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf. Springer, 2009, pp. 363–446 (cited on pages 49, 190).

[Ler09b] Xavier Leroy. "Formal Verification of a Realistic Compiler." In: *Association for Computing Machinery* 52.7 (July 2009). https://doi.org/10.1145/1538788.1538814, pp. 107–115 (cited on page 49).

[LAD20] Tamara Rezk Lesly-Ann Daniel Sébastien Bardin. "Binsec/Rel: Efficient Relational SymbolicExecution for Constant-Time at Binary-Level." In: *2020 IEEE Symposium on Security and Privacy*. https://people.csail.mit.edu/jgross/personal-website/papers/2019-fiat-crypto-ieee-sp.pdf. 2020, pp. 1021–1038 (cited on page 183).

[LIS12] Ji Li, Takanori Isobe, and Kyoji Shibutani. "Converting Meet-In-The-Middle Preimage Attack into Pseudo Collision Attack: Application to SHA-2." In: *Fast Software Encryption (FSE)*. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-642-34047-5_16. Springer, 2012, pp. 264–286 (cited on page 239).

[LIM19] Fukang Liu, Takanori Isobe, and Willi Meier. "Preimages and Collisions for Up to 5-Round Gimli-Hash Using Divide-and-Conquer Methods." In: *IACR Cryptology ePrint Archive* 2019 (2019). https://eprint.iacr.org/2019/1080, p. 1080 (cited on page 99).

[LIM20a] Fukang Liu, Takanori Isobe, and Willi Meier. "Automatic Verification of Differential Characteristics: Application to Reduced Gimli." In: *Advances in Cryptology – CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12172. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-030-56877-1_8. Springer, 2020, pp. 219–248 (cited on page 99).

[LIM20b] Fukang Liu, Takanori Isobe, and Willi Meier. "Exploiting Weak Diffusion of Gimli: A Full-Round Distinguisher and Reduced-Round Preimage Attacks." In: *IACR Cryptology ePrint Archive* 2020 (2020). https://eprint.iacr.org/2020/561, p. 561 (cited on page 99).

[Liu19] S. Liu. *IoT connected devices worldwide 2030*. https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/. 2019 (cited on page 6).

[MT21] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, Jan. 2021 (cited on page 210).

[MS01] Itsik Mantin and Adi Shamir. "A Practical Attack on Broadcast RC4." In: *Fast Software Encryption – FSE 2001*. Vol. 2355. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/3-540-45473-X_13. Springer, 2001, pp. 152–164 (cited on page 162).

[Mat93] Mitsuru Matsui. "Linear Cryptanalysis Method for DES Cipher." In: *Advances in Cryptology – EUROCRYPT 1993*. Ed. by Tor Helleseth. Vol. 765. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/3-540-48285-7_33. Springer, 1993, pp. 386–397 (cited on pages 144, 148, 149, 162).

[MY92] Mitsuru Matsui and Atsuhiro Yamagishi. "A New Method for Known Plaintext Attack of FEAL Cipher." In: *Advances in Cryptology – EUROCRYPT 1992*. Ed. by Rainer A. Rueppel. Vol. 658. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/3-540-47555-9_7. Springer, 1992, pp. 81–91 (cited on page 144).

[MRH04] Ueli Maurer, Renato Renner, and Clemens Holenstein. "Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology." In: *Theory of Cryptography - TCC 2004*. Ed. by M. Naor. Lecture Notes in Computer Science (LNCS) 2951. Springer, 2004, pp. 21–39 (cited on page 235).

[Meh14] Neel Mehta. *Heartbleed – CVE-2014-0160*. https://heartbleed.com/. Apr. 2014 (cited on page 7).

[MNS13] Florian Mendel, Tomislav Nad, and Martin Schläffer. "Improving Local Collisions: New Attacks on Reduced SHA-256." In: *Advances in Cryptology - EUROCRYPT*. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 262–278 (cited on page 239).

[MRV15] Bart Mennink, Reza Reyhanitabar, and Damian Vizár. "Security of Full-State Keyed Sponge and Duplex: Applications to Authenticated Encryption." In: *Advances in Cryptology – ASIACRYPT 2015*. Ed. by Tetsu Iwata and Jung Hee Cheon. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-662-48800-3_19. Berlin, Heidelberg: Springer, 2015, pp. 465–489 (cited on page 32).

[MW18] D. Micciancio and M. Walter. "On the Bit Security of Cryptographic Primitives." In: *Advances in Cryptology – EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-319-78381-9_1. Springer, Mar. 2018, pp. 3–28 (cited on page 235).

[MDV15] Aleksandra Mileva, Vesna Dimitrova, and Vesselin Velichkov. "Analysis of the Authenticated Cipher MORUS (v1)." In: *Cryptography and Information Security in the Balkans – BalkanCryptSec 2015*. Ed. by Enes Pasalic and Lars R. Knudsen. Vol. 9540. Lecture Notes in Computer Science (LNCS). https://10.1007/978-3-319-29172-7_4. Springer, 2015, pp. 45–59 (cited on page 144).

[Mil86] Victor S. Miller. "Use of Elliptic Curves in Cryptography." In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Ed. by Hugh C. Williams. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/3-540-39799-X_31. Berlin, Heidelberg: Springer, 1986, pp. 417–426 (cited on page 5).

[Min14] Brice Minaud. "Linear Biases in AEGIS Keystream." In: *Selected Areas in Cryptography – SAC 2014*. Ed. by Antoine Joux and Amr M. Youssef. Vol. 8781. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-13051-4_18, see also: https://eprint.iacr.org/2018/292. Springer, 2014, pp. 290–305 (cited on pages 144, 162).

[Mon87] Peter L. Montgomery. "Speeding the Pollard and Elliptic Curve Methods of Factorization." In: *Mathematics of Computation* 48.177 (1987). http://links.jstor.org/sici?sici=0025-5718(198701)48:177<243:STPAEC>2.0.CO;2-3, pp. 243–243 (cited on pages 186, 204).

[Mou15] Nicky Mouha. "The Design Space of Lightweight Cryptography." In: *NIST Lightweight Cryptography Workshop 2015*. https://hal.inria.fr/hal-01241013/file/session5-mouha-paper.pdf. Gaithersburg, United States, July 2015 (cited on page 115).

[Mou+14] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. "Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers." In: *Selected Areas in Cryptography – SAC 2014*. Ed. by Antoine Joux and Amr Youssef. Vol. 8781. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-13051-4_19. Springer, 2014, pp. 306–323 (cited on pages 67, 91).

[NIS95] NIST. *Federal Information Processing Standard 180-1, Secure Hash Standard*. Apr. 1995 (cited on page 244).

[NIS01] NIST. *Federal Information Processing Standard 197, Advanced Encryption Standard (AES)*. https://doi.org/10.6028/NIST.FIPS.197. Gaithersburg, MD, USA, Nov. 2001 (cited on page 5).

[NIS02] NIST. *Federal Information Processing Standard 180-2, Secure Hash Standard*. Aug. 2002 (cited on page 244).

[NIS07] NIST. *Special Publication 800-38D, Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. https://doi.org/10.6028/NIST.SP.800-38D. Gaithersburg, MD, USA, Nov. 2007 (cited on page 25).

[NIS15] NIST. *Federal Information Processing Standard 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. http://dx.doi.org/10.6028/NIST.FIPS.202. Gaithersburg, MD, USA, Aug. 2015 (cited on pages 32, 231, 233, 240, 245).

[NIS16] NIST. *NIST Special Publication 800-185, SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash*. https://doi.org/10.6028/NIST.SP.800-185. Dec. 2016 (cited on pages 232, 240, 245).

[Nev] Samuel Neves. *BLAKE2 AVX2 implementations*. https://github.com/sneves/blake2-avx2 (cited on page 245).

[Nis+19] Gorkem Nisanci, Remzi Atay, Meltem Kurt Pehlivanoglu, Elif Bilge Kavun, and Tolga Yalcin. *Will the Future Lightweight Standard be RISC-V Friendly?* https://csrc.nist.gov/CSRC/media/Presentations/will-the-future-lightweight-standard-be-risc-v-fri/images-media/session4-yalcin-will-future-lw-standard-be-risc-v-friendly.pdf. 2019 (cited on page 116).

[Opea] OpenSSL community. *OpenSSL – Cryptography and SSL/TLS Toolkit*. https://github.com/openssl/openssl (cited on pages 7, 245).

[Opeb] OpenSSL/Vulnerabilities. *OpenSSL – Cryptography and SSL/TLS Toolkit*. https://www.openssl.org/news/vulnerabilities.html (cited on page 7).

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES." In: *Topics in Cryptology – CT-RSA 2006*. Ed. by David Pointcheval. Lecture Notes in Computer Science (LNCS). https://www.cs.tau.ac.il/~tromer/papers/cache.pdf. Berlin, Heidelberg: Springer, 2006, pp. 1–20 (cited on page 132).

[Per14] Ray A. Perlner. *Extendable-Output Functions (XOFs)*. SHA 3 workshop 2014. https://csrc.nist.gov/events/2014/sha-3-2014-workshop. Aug. 2014 (cited on pages 30, 232).

[Phi18] Jade Philipoom. "Correct-by-construction finite field arithmetic in Coq." http://adam.chlipala.net/theses/jadep_meng.pdf. MA thesis. Massachusetts Institute of Technology, 2018 (cited on page 184).

[Pie+18a] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Version 5.5. http://www.

cis.upenn.edu/~bcpierce/sf. Electronic textbook, May 2018 (cited on page 43).

[Pie+18b] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf. Electronic textbook, May 2018 (cited on page 43).

[Poe03] B. Poettering. *AVRAES: The AES block cipher on AVR controllers*. http://point-at-infinity.org/avraes/. 2003 (cited on page 91).

[Pro+19] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cedric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*. Cryptology ePrint Archive, Report 2019/757. https://eprint.iacr.org/2019/757. 2019 (cited on page 183).

[Pro+17] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. "Verified Low-Level Programming Embedded in F*." In: *Proceedings of the ACM on Programming Languages*. ICFP 1. http://arxiv.org/abs/1703.00053. Association for Computing Machinery, 2017, p. 17 (cited on pages 8, 183, 190).

[Rfca] *RFC 2360 – Guide for Internet Standards Writers*. https://datatracker.ietf.org/doc/html/rfc2360. June 1998 (cited on page 253).

[Rfcb] *RFC 2418 – IETF Working Group – Guidelines and Procedures*. https://datatracker.ietf.org/doc/html/rfc2418. Sept. 1998 (cited on page 253).

[Rfcc] *RFC 8729 – The RFC Series and RFC Editor*. https://datatracker.ietf.org/doc/html/rfc8729. Feb. 2020 (cited on page 253).

[RSD06] Chester Rebeiro, David Selvakumar, and A. S. L. Devi. "Bitslice Implementation of AES." In: *CANS 2006 – Cryptology and Network Security*. Ed. by David Pointcheval, Yi Mu, and Kefei Chen. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.

1007/11935070_14. Berlin, Heidelberg: Springer, 2006, pp. 203–212 (cited on page 133).

[Res14] Paul Resnick. *RFC 7282 – On Consensus and Humming in the IETF*. https://datatracker.ietf.org/doc/html/rfc7282. June 2014 (cited on page 253).

[RF16] Paul Resnick and Adrian Farrel. *RFC 7776 – IETF Anti-Harassment Procedures*. https://datatracker.ietf.org/doc/html/rfc7776. Mar. 2016 (cited on page 253).

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. Vol. 17. http://www.cs.cmu.edu/~jcr/seplogic.pdf. IEEE, 2002, pp. 55–74 (cited on pages 48, 181).

[RSS11] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. "Careful with Composition: Limitations of the Indifferentiability Framework." In: *Eurocrypt 2011*. Lecture Notes in Computer Science (LNCS). http://dx.doi.org/10.1007/978-3-642-20465-4_27. Springer, 2011, pp. 487–506 (cited on page 235).

[Riv92] Ronald L. Rivest. *The MD5 message-digest algorithm*. RFC 1321 – Internet Request for Comments. Apr. 1992 (cited on page 244).

[Riv02] Ronald L. Rivest. *Foreword in The Design of Rijndael: AES - The Advanced Encryption Standard by Joan Daemen and Vincent Rijmen*. Springer, 2002 (cited on page 5).

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public Key Cryptosystems." In: *Association for Computing Machinery* 21.2 (Feb. 1978). https://doi.org/10.1145/359340.359342, pp. 120–126 (cited on page 5).

[SKC17] Dhiman Saha, Sukhendu Kuila, and Dipanwita Roy Chowdhury. "SymSum: Symmetric-Sum Distinguishers Against Round Reduced SHA3." In: *IACR Transactions on Symmetric Cryptology* 2017.1 (2017). https://doi.org/10.13154/tosc.v2017.i1.240-258, pp. 240–258 (cited on page 239).

[Sal+17] Md. Iftekhar Salam, Leonie Simpson, Harry Bartlett, Ed Dawson, Josef Pieprzyk, and Kenneth Koon-Ho Wong. "Investigating Cube Attacks on the Authenticated Encryption Stream Cipher MORUS." In: *IEEE Trustcom/BigDataSE/ICESS 2017*. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.337. IEEE, 2017, pp. 961–966 (cited on page 144).

[SN16] Niels Samwel and Moritz Neikes. *arm-chacha20*. https://gitlab.science.ru.nl/mneikes/arm-chacha20/tree/master. 2016 (cited on page 91).

[SY15] Yu Sasaki and Kan Yasuda. "How to incorporate associated data in sponge-based authenticated encryption." In: *Topics in Cryptology — CT-RSA 2015*. Ed. by Kaisa Nyberg. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-319-16715-2_19. Springer, 2015, pp. 353–370 (cited on page 32).

[SG15] Erik Schneider and Wouter de Groot. *spongent-avr*. https://github.com/weedegee/spongent-avr. 2015 (cited on page 91).

[SS a] Peter Schwabe and Ko Stoffelen. "All the AES you need on Cortex-M3 and M4." In: *Selected Areas in Cryptology – SAC 2016*. Ed. by Roberto Avanzi and Howard Heys. Lecture Notes in Computer Science (LNCS). https://cryptojedi.org/papers/#aesarm. Springer, to appear (cited on page 91).

[SYY12] Peter Schwabe, Bo-Yin Yang, and Shang-Yi Yang. "SHA-3 on ARM11 processors." In: *Progress in Cryptology – AFRICACRYPT 2012*. Ed. by Aikaterini Mitrokotsa and Serge Vaudenay. Vol. 7374. Lecture Notes in Computer Science (LNCS). https://cryptojedi.org/papers/#sha3arm. Springer, 2012, pp. 324–341 (cited on page 97).

[Sha45] Claude E. Shannon. *A Mathematical Theory of Cryptography*. Classified report. https://www.iacr.org/museum/shannon/shannon45.pdf. Murray Hill, NJ, USA: Bell Laboratories, Sept. 1945 (cited on page 22).

[Shi+16] Tairong Shi, Jie Guan, Junzhi Li, and Pei Zhang. "Improved Collision Cryptanalysis of Authenticated Cipher MORUS." In: *Artificial Intelligence and Industrial Engineering – AIIE 2016*. Vol. 133. Advances in Intelligent Systems Research. Atlantis Press, 2016, pp. 429–432 (cited on page 144).

[SM87] Akihiro Shimizu and Shoji Miyaguchi. "Fast data encipherment algorithm FEAL." In: *Advances in Cryptology – EUROCRYPT' 87*. Ed. by David Chaum and Wyn L. Price. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/3-540-39118-5_24. Springer. Cham: Springer, Apr. 1987, pp. 267–278 (cited on page 21).

[Sho94] Peter W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring." In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. https://doi.org/10.1109/SFCS.1994.365700. 1994, pp. 124–134 (cited on page 6).

[Soc] American Mathematical Society. *The Culture of Research and Scholarship in Mathematics: Joint Research and Its Publication*. https://www.ams.org/profession/leaders/CultureStatement04.pdf (cited on page 9).

[SLG17a] Ling Song, Guohong Liao, and Jian Guo. "Non-full Sbox Linearization: Applications to Collision Attacks on Round-Reduced Keccak." In: *Advances in Cryptology - CRYPTO 2017*. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-63715-0_15. Springer, 2017, pp. 428–451 (cited on pages 231, 238).

[SLG17b] Ling Song, Guohong Liao, and Jian Guo. *Solution to the 6-round collision challenge*. https://keccak.team/crunchy_contest.html. 2017 (cited on page 231).

[Spia] *SpinalHDL language*. https://github.com/SpinalHDL/SpinalHDL (cited on page 118).

[Sta77] National Bureau of Standards. *Federal Information Processing Standard 46: Data Encryption Standard*. https://csrc.nist.gov/CSRC/media/Publications/fips/46/archive/1977-01-15/documents/NBS.FIPS.46.pdf. Gaithersburg, MD, USA, Jan. 1977 (cited on pages 4, 21).

[Sta81] National Bureau of Standards. *Federal Information Processing Standard 74, Guidelines for Implementing and Using the NBS Data Encryption Standard*. https://doi.org/10.6028/NBS.FIPS.74. Gaithersburg, MD, USA, Apr. 1981 (cited on page 4).

[Sta85] National Bureau of Standards. *Federal Information Processing Standard 113, Computer Data Authentication*. https://csrc.nist.gov/publications/detail/fips/113/archive/1985-05-30, see also https://csrc.nist.rip/publications/fips/fips113/fips113.html. Gaithersburg, MD, USA, Apr. 1985 (cited on page 25).

[SN97] National Institute of Standards and Technology (NIST). *Announcing Development of a Federal Information Processing Standard for Advanced Encryption Standard*. https://csrc.nist.gov/news/1997/announcing-development-of-fips-for-advanced-encryp. 1997 (cited on page 4).

[SN15] National Institute of Standards and Technology (NIST). *Lightweight Cryptography*. https://csrc.nist.gov/Projects/lightweight-cryptography. 2015 (cited on pages 6, 68, 115).

[SN17] National Institute of Standards and Technology (NIST). *Post-Quantum Cryptography Standardization*. https://csrc.nist.gov/projects/post-quantum-cryptography. 2017 (cited on page 6).

[Sto19] Ko Stoffelen. "Efficient Cryptography on the RISC-V Architecture." In: *Progress in Cryptology – LATINCRYPT 2019*. Ed. by Peter Schwabe and Nicolas Thériault. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-030-30530-7_16. Cham: Springer, 2019, pp. 323–340 (cited on pages 11, 115, 130, 132, 133, 136).

[Sul15] Nick Sullivan. *Do the ChaCha: better mobile performance with cryptography*. https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/. 2015 (cited on page 67).

[TW15] Biaoshuai Tao and Hongjun Wu. "Improving the Biclique Cryptanalysis of AES." In: *ACISP 2015: Information Security and Privacy*. Ed. by Ernest Foo and Douglas Stebila. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-319-19962-7_3. Cham: Springer, 2015, pp. 39–56 (cited on page 26).

[Coq] *The Coq Proof Assistant – Frequently Asked Questions*. https://coq.inria.fr/faq (cited on pages 45, 181, 190, 213).

[Thi] *Things that use Curve25519*. https://ianix.com/pub/curve25519-deployment.html. 2019 (cited on pages 5, 181).

[TS20] Martin Thomson and Barbara Stark. *RFC 8874 – Working Group GitHub Usage Guidance*. https://datatracker.ietf.org/doc/html/rfc8874. Aug. 2020 (cited on page 253).

[Tod15] Yosuke Todo. "Structural Evaluation by Generalized Integral Property." In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science (LNCS). https://eprint.iacr.org/2015/090.pdf. Springer, 2015, pp. 287–314 (cited on page 89).

[TM16] Yosuke Todo and Masakatu Morii. "Bit-Based Division Property and Application to Simon Family." In: *Fast Software Encryption - 23rd International Conference, FSE 2016*. Ed. by Thomas Peyrin. Vol. 9783. Lecture Notes in Computer Science (LNCS). https://eprint.iacr.org/2016/285.pdf. Springer, 2016, pp. 357–377 (cited on page 89).

[Tur+21] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Çağdaş Çalık, Lawrence Bassham, Jinkeon Kang, and John Kelsey. *Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process*. Tech. rep. https://doi.org/10.6028/NIST.IR.8369. Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST), July 2021 (cited on page 99).

[VV17] Serge Vaudenay and Damian Vizár. *Under Pressure: Security of CAESAR Candidates beyond their Guarantees*. Cryptology ePrint Archive, Report 2017/1147. https://eprint.iacr.org/2017/1147. 2017 (cited on page 144).

[Ver26] Gilbert Sandford Vernam. "Cipher Printing Telegraph Systems for Secret Wire and Radio Telegraphic Communications." In: *Journal American Institute of Electrical Engineers* XLV (1926), pp. 109–115 (cited on page 4).

[Spib] *VexRiscv simulator*. https://github.com/SpinalHDL/VexRiscv (cited on page 118).

[Vig18] Benoît Viguier. *KangarooTwelve*. Internet Research Task Force draft. https://datatracker.ietf.org/doc/draft-viguier-kangarootwelve/. Mar. 2018 (cited on page 241).

[Wan+20] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. "XMSS and Embedded Systems." In: *Selected Areas in Cryptography – SAC 2019*. Ed. by Kenneth G. Paterson and Douglas Stebila. Lecture Notes in Computer Science (LNCS). https://link.springer.com/chapter/10.1007/978-3-030-38471-5_21. Cham: Springer, 2020, pp. 523–550 (cited on page 115).

[Wat+17] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.2*. 2017 (cited on page 115).

[Wea16] Rhys Weatherley. *ArduinoLibs*. https://rweather.github.io/arduinolibs/crypto.html. 2016 (cited on page 91).

[Wea20] Rhys Weatherley. *Lightweight Cryptography Primitives*. https://rweather.github.io/lightweight-crypto/index.html. 2020 (cited on pages 11, 134, 137).

[WP96] Abel Weinrib and Jon Postel. *RFC 2014 – IRTF Research Group Guidelines and Procedures*. https://datatracker.ietf.org/doc/html/rfc2014. Oct. 1996 (cited on page 252).

[Wu11] Hongjun Wu. *The Hash Function JH*. Submission to NIST (round 3). http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf. 2011 (cited on page 244).

[WH16] Hongjun Wu and Tao Huang. *The Authenticated Cipher MORUS (v2)*. Submission to CAESAR: Competition for Authenticated Encryption. Security, Applicability, and Robustness (Round 3 and Finalist). http://competitions.cr.yp.to/round3/morusv2.pdf. Sept. 2016 (cited on pages 143–145).

[WP13] Hongjun Wu and Bart Preneel. "AEGIS: A Fast Authenticated Encryption Algorithm." In: *Selected Areas in Cryptography – SAC 2013*. Ed. by Tanja Lange, Kristin E. Lauter, and Petr Lisonek. Vol. 8282. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-662-43414-7_10, see also: https://eprint.iacr.org/2013/695. Springer, 2013, pp. 185–201 (cited on pages 143, 144).

[WP16] Hongjun Wu and Bart Preneel. *AEGIS: A Fast Authenticated Encryption Algorithm (v1.1)*. Submission to CAESAR: Competition for Authenticated Encryption. Security, Applicability, and Robustness (Round 3 and Finalist). http://competitions.cr.yp.to/round3/aegisv11.pdf. Sept. 2016 (cited on pages 143, 144).

[Xia+16] Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. "Applying MILP Method to Searching Integral Distinguishers Based on Division Property for 6 Lightweight Block Ciphers." In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science (LNCS). https://eprint.iacr.org/2016/857. Springer, 2016, pp. 648–678 (cited on page 89).

[Ye+17] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. "Verified Correctness and Security of MbedTLS HMAC-DRBG." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS 17. https://doi.org/10.1145/3133956.3133974. New York, NY, USA: Association for Computing Machinery, 2017, pp. 2007–2020 (cited on page 184).

[Yuv79] Gideon Yuval. "How to swindle Rabin." In: *Cryptologia* 3.3 (1979). https://www.tandfonline.com/doi/pdf/10.1080/0161-117991854025, pp. 187–191 (cited on page 27).

[ZBB16] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. "A Verified Extensible Library of Elliptic Curves." In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7536383. IEEE, 2016, pp. 296–309 (cited on pages 8, 183).

[Zin+17] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A verified modern cryptographic library." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. https://eprint.iacr.org/2017/536.pdf. Association for Computing Machinery, 2017, pp. 1789–1806 (cited on pages 8, 183, 214).

[ZDW19] Rui Zong, Xiaoyang Dong, and Xiaoyun Wang. "Collision Attacks on Round-Reduced Gimli-Hash/Ascon-Xof/Ascon-Hash." In: *IACR Cryptology ePrint Archive* 2019 (2019). https://eprint.iacr.org/2019/1115, p. 1115 (cited on pages 99, 100).

ACRONYMS

ACNS	Applied Cryptography and Network Security
AD	Associated Data
AEAD	Authenticated Encryption scheme with Associated Data
AES	Advanced Encryption Standard
AMS	American Mathematical Society
API	Application Programming Interface
ARX	Add Rotate and Xor
AST	Abstract Syntax Tree
AVX	Advanced Vector Extensions
CBC	Cipher Block Chaining
CFRG	Crypto Forum Research Group
CI	Continuous Integration
CICO	constrained-input constrained-output
CPU	Central Processing Units
CTR	Counter
CVE	Common Vulnerabilities and Exposures
DDT	differential distribution table
DES	Data Encryption Standard
DP	Differential Probability
DSL	domain-specific language
ECC	Elliptic-Curve Cryptography
EDP	Expected Differential Probability
FEAL	Fast data Encipherment ALgorihtm
FIPS	Federal Information Processing Standard
GPG	GNU Privacy Guard
IACR	International Association for Cryptologic Research
IBM	International Business Machines Corporation
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
IOT	"Internet of Things"
IRSG	Internet Research Steering Group
IRTF	Internet Research Task Force

ISO	International Organization for Standardization
IP	Internet Protocol
IV	Initial Value
KCP	Keccak code package
LWC	Lightweight Cryptography
MAC	Message Authentication Code
MITM	Meet-In-The-Middle
NBS	National Bureau of Standards
NIST	National Institute of Standards and Technology
NSA	National Security Agency
NSS	Network Security Services
OTP	One-Time Pad
P-box	Permutation box
PRP	pseudo-random permutation
RG	Research Group
RFC	Request for Comments
S-box	Substitution box
SHA	Secure Hash Algorithm
SIMD	single-instruction-multiple-data
SoK	Systematization-of-Knowledge
SPN	Substitution Permutation Network
SSD	solid-state drives
SSE	Streaming SIMD Extensions
SSH	Secure Shell
TCP	Transmission Control Protocol
TLS	Transport Layer Security
VST	Verifiable Software Toolchain
WG	Working Group
XOF	eXtendable Output Function
XOP	eXtended Operations

RESEARCH DATA MANAGEMENT

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands.[5]

The following research datasets have been produced during this PhD research:

- Chapter 4: Bernstein, D.J., Kölbl, S., Lucks, S., Massolino, P.M.C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F.-X., Todo, Y. & Viguier., B.G.P. (2017). Gimli: a cross-platform permutation (2017).
- Chapter 5: Campos, F., Jellema, L., Lemmen, M., Müller, L., Sprenkels, D. & Viguier, B.G.P. (2020). Assembly or Optimized C for Lightweight Cryptography on RISC-V?
- Chapter 6: Ashur, T., Eichlseder, M., Lauridsen, M.M., Leurent, G., Minaud, B., Rotella, Y., Sasaki, Y. & Viguier, B.G.P. (2018). Cryptanalysis of MORUS.
- Chapter 7: Schwabe, P., Viguier, B.G.P., Weerwag, T. & Wiedijk, F. (2020). A Coq proof of the correctness of X25519 in TweetNaCl.
- Chapter 8: Bertoni, G., Daemen, J., Peeters, M., Assche, G. Van, Keer, R. Van & Viguier, B.G.P. (2018) KangarooTwelve: Fast Hashing Based on KECCAK-p.

An archive *A Panorama on Classical Cryptography - Files* with all the source files, with versions corresponding to the ones presented in this thesis, is available at: `https://doi.org/10.5281/zenodo.4534692`.

5 ru.nl/icis/research-data-management/, last accessed January 20th, 2020.

SUMMARY

When taking a picture, the landscape photographer has to decide between using a telephoto lens, focusing on details at distance; or using a wide-angle one to capture a broad section of the scene. Similarly, in this thesis we chose the second approach to cover a large part of the classical cryptography world: we examine the design of new symmetric primitive; we explore implementation strategies of lightweight schemes; we analyze a new high performance algorithm; we use formal verification to prove the correctness of Elliptic-Curve Cryptography (ECC) implementations; and finally we describe one of the way algorithms are standardized.

PART I. This aims to provide the necessary background information for a broad understanding of the concepts presented later in the text. Chapter 2 focuses on the basis of symmetric cryptography and cryptanalysis, and chapter 3 gives the reader a brief introduction to formal verification.

The rest of this thesis (PART II, III, & IV) is organized a sequence of paper, whose contents are succinctly described here.

CHAPTER 4. This chapter introduces GIMLI, a 384-bit permutation designed to achieve high security with high performance across a broad range of platforms, covering 64-bit Intel/AMD server CPUs, ARM smartphone CPUs, ARM & AVR microcontrolers, and other specialized devices. Additionally, we introduce GIMLI-CIPHER and GIMLI-HASH, our submissions at the NIST lightweight cryptography competition.

CHAPTER 5. This chapter presents different optimization strategies applied to several candidates of the NIST lightweight cryptography standardization efforts. We focus on the RISC-V 32 bit architecture and study the general impact of optimizations in assembly and in plain C. We present optimized implementations with speed-up of up to 81% over available implementation at that time. Furthermore, we highlight the necessity to benchmark implementations on physical devices as opposed to only using RISC-V simulators.

CHAPTER 6. This chapter focuses on MORUS, a high-performance authenticated encryption algorithm submission and finalist at the CAESAR competition. We analyze the candidate's components (initialization, state update and tag generation) and report several results.

Our main result is the linear correlation in the keystream of full MORUS, which can be used to distinguish its output from random and

to recover plaintext bits in the broadcast setting. We exhibit correlation which can be exploited after 2^{152} encryptions, breaking the 256-bit security claim.

CHAPTER 7. This chapter presents the mechanized formal proof of correctness of the X25519 key-exchange protocol in the TweetNaCl cryptographic library. With the theorem prover Coq, we provide three computer-verified proofs: (1) the implementation matches the RFC 7748 standard without undefined behaviour, overflows etc. (2) the RFC matches the definition from Bernstein's 2006 paper, (3) the X25519 definition in Bernstein's paper correctly computes a scalar multiplication on the elliptic curve Curve25519.

This proof makes use of the Verifiable Software Toolchain to prove the C implementation correctness and the elliptic-curve library from Bartzia and Strub to verify X25519.

PART IV. Chapter 8 proposes KangarooTwelve, a fast and secure XOF —a hash function with arbitrary output length— aiming at higher speed than the FIPS-202's SHA-3 functions. It provides the same security strength as the SHAKE128 but with two major improvements. First it comes with a built-in parallel mode which efficiently exploits SIMD instructions. Second it makes use of the ten years of cryptanalysis of Keccak, halving the number of rounds. With these two changes KangarooTwelve achieves significant speed improvement and consumes less than 0.55 cycles/bytes for long messages on the Intel's SkylakeX architecture.

Chapter 9 focuses our efforts on the standardization of the KangarooTwelve function at the IRTF. More generally we provide insights of how the IETF standardization process works with an emphasis of the procedures within the CFRG working group.

SAMENVATTING

Bij het maken van een foto moet een landschapsfotograaf kiezen tussen een telelens, met een focus op details van een afstand; of een groothoeklens om het hele landschap in een beeld vast te leggen. Op eenzelfde manier kiezen we in deze proefschrift voor de tweede aanpak om een groot deel van de klassieke cryptografische wereld te omvatten: we onderzoeken het design van nieuwe symmetrische primitieven; we verkennen verschillende strategieën voor implementatie van lichtgewicht cryptosystemen; we analyseren een nieuw hoogpresterend encryptie-algoritme; we gebruiken formele verificatie om de correctheid van implementaties van cryptografie met elliptische krommen (ECC) te bewijzen; en tenslotte beschrijven we een van de manieren waarop algoritmes gestandaardiseerd worden.

DEEL I. In dit deel beschrijven we de achtergrondinformatie die nodig is om concepten die later aan bod komen goed te begrijpen. Hoofdstuk 2 focust op de basis van symmetrische cryptografie en cryptanalyse, en hoofdstuk 3 geeft de lezer een korte introductie tot formele verificatie.

De rest van deze proefschrift (DEEL II, III & IV) is georganiseerd als een serie publicaties, waarvan we de inhoud hier beknopt beschrijven.

HOOFDSTUK 4. Dit hoofdstuk introduceert GIMLI, een 384-bits permutatie ontworpen voor sterke veiligheid met hoge prestaties over een breed spectrum van platformen, die onder andere 64-bits Intel/AMD server CPUs, ARM smartphone CPUs, ARM & AVR microcontrollers en andere gespecialiseerde apparatuur omvat. Daarnaast introduceren we GIMLI-CIPHER en GIMLI-HASH, onze inzendingen voor de NIST lightweight cryptography competitie.

HOOFDSTUK 5. Dit hoofdstuk presenteert verschillende optimalisatie-strategieën toegepast op verschillende kandidaten voor het NIST lightweight cryptography standaardisatieproces. We focussen specifiek op de RISC-V 32-bits architectuur en bestuderen de algemene impact van optimalisaties in assembly en in standaard C. We presenteren geoptimaliseerde implementaties die een versnelling van meer dan 81% geven ten opzichte van beschikbare implementaties van die tijd. Verder belichten we de noodzaak om implementaties te benchmarken op fysieke apparaten, in tegenstelling tot RISC-V simulators.

HOOFDSTUK 6. Dit hoofdstuk focust op MORUS, een hoogpresterend algoritme voor geauthenticeerde encryptie en een finalist bij de CAESAR competitie. We analyseren verschillende componenten van

dit algoritme (initialisatie, toestandsupdate, en tag-generatie). Het belangrijkste resultaat is de lineaire correlatie in de keystream van Morus, die gebruikt kan worden om zijn output van willekeurige output te onderscheiden, waarmee de originele tekst achterhaald kan worden in een broadcast situatie. De correlatie in de keystream kan worden uitgebuit na 2^{152} vercijferingen, en breekt daarmee de 256-bits veiligheid.

HOOFDSTUK 7. Dit hoofdstuk presenteert het gemechaniseerde formele bewijs van correctheid van het X25519-sleuteluitwisselingsprotocol in de TweetNaCl cryptografische bibliotheek. Met de stellingbewijzer Coq geven we drie bewijzen: (1) de implementatie komt overeen met de RFC 7748-standaard zonder ongedefinieerd gedrag, overflows etc. (2) de RFC komt overeen met de definitie van de publicatie van Bernstein uit 2006, (3) de X25519-definitie in Bernstein's publicatie berekent het scalaire veelvoud op de elliptische kromme Curve25519. Dit bewijs maakt gebruik van de Verifiable Software Toolchain om de correctheid van de C-implementatie te bewijzen, en van de elliptische-krommenbibliotheek van Bartzia en Strub om X25519 te verifiëren.

DEEL IV. Hoofdstuk 8 stelt KangarooTwelve voor: een snelle en veilige XOF –een hash functie met willekeurige outputlengte– die mikt op een hogere snelheid dan de SHA-3 functies uit FIPS-202. Het biedt dezelfde veiligheid als SHAKE128, maar bevat twee grote verbeteringen. Allereerst beschikt het over een ingebouwde parallelmodus die efficiënt gebruik maakt van SIMD instructies. Bovendien maakt het gebruik van tien jaar ervaring in cryptoanalyse van Keccak, waardoor het aantal rondes halveert. Met deze twee veranderingen bereikt KangarooTwelve een significante versnelling en gebruikt het minder dan 0.55 cycles per byte voor langere berichten op Intel's SkylakeX architectuur.

Hoofdstuk 9 focust op de standaardisatie van de KangarooTwelve functie bij het IRTF. Algemener gesproken geven we inzicht in hoe het IETF-standaardisatieproces werkt, met een nadruk op de procedures in de CFRG werkgroep.

SOMMAIRE

Afin de prendre une photo, le photographe a le choix entre utiliser un téléobjectif, pour simplifier la composition et de se focaliser sur un détail lointain, et utiliser un objectif grand angle pour réaliser un panorama. Nous avons choisi la seconde approche dans ce manuscrit : nous esquissons le large sujet qu'est la cryptographie. Pour cela, nous examinons le design d'une nouvelle primitive cryptographique, nous explorons les multiples stratégies d'implémentations pour des algorithmes de chiffrement léger, nous analysons un algorithme de chiffrement à haute performance, nous nous penchons sur l'utilisation des méthodes formelles pour prouver la bonne d'implémentation de la cryptographie à courbes elliptiques (ECC), et enfin, nous détaillons le processus de standardisation d'une nouvelle fonction de hachage.

PARTIE I. Dans cette partie, nous posons, dans le premier chapitre, les bases nécessaires à la compréhension des idées ensuite développées dans cette thèse. Le second chapitre présente plus en détails les notions de la cryptographie et de la cryptanalyse symétrique. Puis, le troisième chapitre fourni une brève introduction aux méthodes de vérification formelle.

Dans la suite de cette thèse ; c'est-à-dire les parties II, III et IV ; une suite de publications est présentée dont nous décrivons le contenu ci-dessous.

CHAPITRE 4. Ce chapitre présente GIMLI, une permutation de 384 bits prévue pour un haut niveau de sécurité, sans compromis de performance, pour un large éventail de plateformes couvrant : les serveurs avec processeurs 64 bits Intel et AMD, les téléphones avec processeur ARM, les microcontrôleurs à processeur ARM et AVR, ainsi qu'une variété d'outils specialisés. De plus, nous décrivons nos candidats à la compétition de la cryptographie légère du NIST : GIMLI-CIPHER et GIMLI-HASH à.

CHAPITRE 5. Ce chapitre présente différentes stratégies d'optimisations possible qui sont appliquées à plusieurs candidats de la compétition pour la cryptographie légère du NIST. Nous nous focalisons sur l'architecture 32 bit RISC-V, et nous étudions l'impact des optimisations en assembleur et dans le langage C.

En guise de résultat, nous obtenons des améliorations de vitesse allant jusqu'à 81% par rapport à d'autres implémentations dès à présent disponibles. De plus, nous soulignons la nécessité de tester les implémentations sur des cœurs physiques, et non pas seulement sur des simulateurs RISC-V.

CHAPITRE 6. Ce chapitre se concentre sur MORUS, un algorithme de chiffrement à haute performance avec authentification. Nous analysons les composants (l'initialisation, la mise-à-jour de l'état et la génération du tag d'authentification) de ce finaliste de la compétition CAESAR.

À la suite de cela, nous avons trouvé une corrélation linéaire dans le flot de chiffrement de MORUS. Ce biais peut être utilisé pour identifier l'algorithme utilisé par rapport à un flot complètement aléatoire. Il peut aussi être mis à profit pour extraire des bits du message en clair dans un système de diffusion. La corrélation pouvant être exploitée après 2^{152} chiffrements, elle casse le niveau de sécurité revendiqué à 256 bits.

CHAPITRE 7. Ce chapitre présente une preuve formelle de l'exactitude du protocole d'échange de clé X25519 dans son implémentation dans la bibliothèque cryptographique TweetNaCl. Nous utilisons l'assistant de preuve Coq pour prouver : (1) que l'implémentation répond bien au standard RFC 7748, et ce sans comportement non-défini, (2) que le standard RFC correspond bien aux définitions du papier de Bernstein publié en 2006, (3) que la définition de X25519 dans le papier de Bernstein calcule correctement une multiplication scalaire sur la courbe elliptique Curve25519.

Cette preuve utilise le VST afin de prouver la justesse de l'implémentation en C du protocole X25519. La preuve de la définition de X25519 s'appuie également sur la bibliothèque pour les courbes elliptiques de Bartzia et Strub.

PARTIE IV. Le chapitre 8 décrit KANGAROOTWELVE, une fonction de hachage à sortie variable rapide et sûre. Cette fonction vise des vitesses supérieures à celles du standard SHA-3 défini dans FIPS-202. Le niveau de sécurité revendiqué est égal à celui de SHAKE128, mais apporte deux améliorations majeures. Tout d'abord, notre fonction utilise un parallélisme interne qui exploite les instructions SIMD, et elle met à profit les dix ans de cryptanalyse de KECCAK pour diviser par deux le nombre de rondes requises. Avec ces deux changements, KANGAROOTWELVE fourni des améliorations de vitesse conséquent, utilisant moins de 0,55 cycles par octet, pour des messages longs, testée sur la dernière architecture serveur Intel SkylakeX.

Enfin, le dernier chapitre de cette thèse se tourne sur les efforts de standardisation de KANGAROOTWELVE au sein de l'IRTF. Nous décrivons plus en détail le processus au sein du groupe de travail CFRG.

ABOUT THE AUTHOR

Benoît Viguier was born in Angers, France, on March 29th, 1988. From 2006, he attended the Higher School Preparatory Classes (CPGE) at the Lycée Chateaubriand in Rennes. Two years later he joined the 3rd year Bachelor of Mathematics at Université Rennes 1 and subsequently obtained in 2011 his Master's degree of Mathematics, with a major in teaching. One year later he received his CAPES[1] of Mathematics and continued as a Mathematics teacher in high school and junior high school throughout the school years of 2013 and 2014.

In September 2014, he attended the National Institute of Applied Sciences (INSA) in Rennes, and joined the double-degree program Master Research in Computer Science in 2015. He received his Engineer Diploma and Master of Research degree in 2016 before starting his PhD at Radboud University, in Nijmegen, The Netherlands.

MENS SANA IN CORPORE SANO. While working on his PhD, Benoît started dance sport, and is now practicing at a high level. In 2018, he joined the Sway of Life formation team. Together they competed thrice at the Dutch championship (2018, 2019, & 2021), ranking first each time. They also got 6th place in the finals of the 2019 World championship in Moscow, Russia. In couple dancing with his dance partner, they reached 4th place in C-class on their first competition at the 2020 Dutch championship.

In addition to dancing, Benoît is heavily involved in photography with interest ranging from landscapes (sunrises & sunsets), dance sport, and portraiture in natural light.

1 *Certificate of aptitude for secondary school teachers*

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[Ash+18] Tomer Ashur, Maria Eichlseder, Martin M. Lauridsen, Gaëtan Leurent, Brice Minaud, Yann Rotella, Yu Sasaki, and Benoît Viguier. "Cryptanalysis of MORUS." In: *Advances in Cryptology – ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*. Ed. by Thomas Peyrin and Steven D. Galbraith. Vol. 11273. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-030-03329-3_2, see also https://eprint.iacr.org/2018/464. Springer, 2018, pp. 35–64.

[Ber+17a] 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*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-66787-4_15, see also https://eprint.iacr.org/2017/630. Springer, 2017, pp. 299–320.

[Ber+18b] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. "KangarooTwelve: Fast Hashing Based on Keccak-p." In: *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*. Ed. by Bart Preneel and Frederik Vercauteren. Vol. 10892. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-319-93387-0_21, see also https://eprint.iacr.org/2016/770. Springer, 2018, pp. 400–418.

[Cam+20] Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoît Viguier. "Assembly or Optimized C for Lightweight Cryptography on RISC-V?" In: *CANS 2020: Cryptology and Network Security*. Ed. by Stephan Krenn, Haya Shulman, and Serge Vaudenay. Vol. 12579. Lecture Notes in Computer Science (LNCS). https://doi.org/10.1007/978-3-030-65411-5_26. Cham: Springer, 2020, pp. 526–545.

[Sch+21] Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. "A Coq proof of the correctness of X25519 in TweetNaCl." In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. https://doi.ieeecomputersociety.org/10.1109/CSF51468.2021.00023. Los Alamitos, CA, USA: IEEE Computer Society, June 2021, pp. 265–280.

[Vig18] Benoît Viguier. *KangarooTwelve*. Internet Research Task Force draft. https://datatracker.ietf.org/doc/draft-viguier-kangarootwelve/. Mar. 2018.

COLOPHON

PROOFREADERS. This thesis would not have reached such quality without the feedback and patience of the following amazing people who took the time to go through the text in its multiple iterations:
Denisa Greconici (*Chapters 1, 8, & 9*),
Peter Schwabe (*Chapter 1, 4, 5 & 9*),
Joan Daemen (*Chapter 2, 6, & 8*),
Krijn Reijnders (*Chapter 2, & Samenvatting*),
Freek Wiedijk (*Chapter 3, 7, & Samenvatting*),
Gilles Van Assche (*Chapter 2*),
Alyssa Byrnes (*Chapter 3*),
Łukasz Chmielewski (*Chapter 3*),
Marcel Fourné (*Chapter 3*),
Herman Geuvers (*Chapter 3*),
Daan Sprenkels (*Chapter 5*),
Robert Moskowitz (*Chapter 9*),
Thom Wiggers (*Samenvatting*),
Anna Guinet (*Sommaire*)

XKCD. Figures 1.1 and 7.5 were originally published on xkcd.com. They are licensed under under a Creative Commons Attribution-NonCommercial 2.5 License; see `https://xkcd.com/license.html`, *last accessed March 18th, 2021*.

COVER DESIGN. With Loes Kema, thank you for your patience with the multiple iterations. Photographs taken by the author.

STYLE. This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".

PRINTED. By GVO Drukkers & Vormgevers – `proefschriften.nl`

ISBN. 978–94–6332–806–7

FINAL VERSION. As of October 27, 2021 (`58aa764e`).