Angela Promitzer, BSc

Efficient Picnic: Optimizing a Post-Quantum Signature Scheme

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Christian Rechberger

Institute of Applied Information Processing and Communications

Advisor
Sebastian Ramacher

Graz, March 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

Digital signatures play a crucial role in our digital world. They have for example a major impact on the security of our online communication, e-government applications, and online banking. It is known since the 1980s that most of the digital signature schemes which are in use today can be broken by a sufficiently large quantum computer. Although such a powerful quantum computer does not exist yet, it is necessary to start the transition to so-called post-quantum signature schemes already now. Thus, the National Institute of Standards and Technology hosts the Post-Quantum Cryptography project which aims to find suitable post-quantum signature schemes. Picnic, a post-quantum signature scheme which was developed at the Institute of Applied Information Processing and Communications, was submitted to this project.

In this thesis we optimize LowMC which is one of the main building blocks of Picnic. We propose two structural and one implementational optimization. First, we improve the computation of the round key which yields a performance gain of almost 50%. Second, we propose a Feistel network which replaces a matrix multiplication. This optimization can reduce the memory requirements of a matrix multiplication by up to 97%. The third optimization targets the implementation of Picnic on devices which feature an ARM CPU. We exploit the single instruction, multiple data instruction set extension NEON for implementing matrix operations. This optimization creates a performance gain of 5 to 10%.

Kurzfassung

Digitale Signaturen sind ein wichtiger Bestandteil unserer digitalen Welt. Sie haben beispielsweise großen Einfluss auf die Sicherheit von Online-Kommunikation, E-Government Anwendungen, und Onlinebanking. Es ist seit den 1980er Jahren bekannt, dass die meisten digitalen Signaturschemata, die heutzutage verwendet werden, durch einen ausreichend großen Quantencomputer gebrochen werden können. Obwohl ein solcher leistungsfähiger Quantencomputer noch nicht existiert, ist es bereits jetzt notwendig, dass wir mit dem Übergang zu sogenannten Post-Quanten-Signaturschemata zu beginnen. Deshalb hat das *National Institute for Standards and Technologie* das *Post-Quantum Cryptography* Projekt ausgeschrieben, das zum Ziel hat, geeignete Post-Quanten-Signaturschemata zu finden. Picnic, ein Post-Quanten-Signaturschema das am Institut für angewandte Informationsverarbeitung und Kommunikationstechnologie entwickelt wurde, wurde zu diesem Projekt eingereicht.

In dieser Masterarbeit optimieren wir LowMC, einen der Hauptbestandteile von Picnic. Wir präsentieren drei Optimierungen, wovon zwei die Struktur und eine die Implementierung von LowMC betreffen. Zuerst verbessern wir die Berechnung des Rundenschlüssels, was zu einem Leistungsgewinn von fast 50% führt. Zweitens schlagen wir ein Feistel-Netzwerk vor, das eine Matrixmultiplikation ersetzt. Diese Optimierung kann den Speicherbedarf für eine Matrixmultiplikation um bis zu 97 % reduzieren. Die dritte Optimierung zielt auf die Implementierung von Picnic auf Geräten mit einer ARM-CPU ab. Wir verwenden die *Single instruction, multiple data* Erweiterung NEON zum Implementieren von Matrixoperationen. Diese Optimierung erzeugt einen Leistungszuwachs von 5 bis 10%.

Acknowledgements

First of all, I would like to express my gratitude to Sebastian Ramacher for providing his support during the implementation of this master's thesis. His door was always open for my questions no matter how ridiculous they were and we spent countless hours discussing more or less serious issues we came across. I am also thankful for his reviews of the written drafts of this thesis.

Second, I want to thank my parents Christine and Manfred Promitzer. They always gave me the support I needed during my studies and encouraged me to fight for my goals.

I am very thankful for many discussions with Michael Schwarz. He was a big support during the creation of this thesis and provided valuable insights concerning code generation of compilers and code optimization.

Furthermore, I owe my thanks to Christian Rechberger who introduced me to the topic in the first place and supervised the thesis, Tyge Tiessen for ideas for the optimization of LowMC's linear operations, and Léo Perrin for his input to the Fibonacci Feistel network.

Finally, I thank Andrea Pferscher for reviewing my thesis, and David Derler for providing input to my discussions with Sebastian and feedback for my presentation of the intermediate results of the thesis.

Contents

List of Figures

List of Tables

1. Introduction

In our modern world of digital communication it is indispensable that we make use of digital signatures. For example, we want to send a message and make sure that the receiver knows that the received message is really the message we sent. Furthermore, we also need to identify our communication partner correctly. It is not enough to know that the received message has not been modified if we do not know who sent the message. When we connect to our online banking we want to be sure we are truly connecting to the website of our bank and not to some adversary's website. In other words, we have to ensure that we can confirm the identity of our communication party with an error probability that is negligible, *i.e.,* nobody can trick us into believing that he or she is somebody else. Digital signatures provide the possibility for our communication partner to ensure that the received message is truly from us and that the content of the message has not been changed during the transmission. The security of our digital communication is ensured in all modern browsers using TLS [DR08]. TLS makes use of digital signatures for example to authenticate the key exchange between two parties which is required for the further encryption of the communication. Another application of digital signatures in TLS is the identification of the communicating parties. Furthermore, digital signatures are used in the Domain Name System Security Extensions (DNSSEC [Are+05]) which aims to secure the Domain Name System (DNS [Moc87]). DNS is mostly known for translating the URL that is entered into the browser into an IP-Address that can be used to locate the requested website on the internet. Another application of digital signatures is code signing in an app store[1]. There, all applications that are uploaded to an app store are signed to prevent adversaries from infiltrating a malicious app into the store. If a user downloads a signed app he can make sure that the version he installs

[1] `https://developer.apple.com/support/code-signing/`

is truly the one that the owner of the store approved. Similarly hardware manufacturers provide signed device drivers[2].

Digital signatures can also be used in the context of e-government applications. In Austria, it is possible to create a legally binding digital signature using a smartphone or a smartcard[3]. These signatures can be used for administrative procedures including contract and petition signing, online banking, and tax declarations. They can replace handwritten signatures which are an important part of our daily life. When we sign a contract we agree to the content of the contract and all terms and conditions associated with it. Our signature, no matter if handwritten or digital, can be used by the second party of the contract to prove that it was truly us who signed the contract.

In general, digital signatures have to fulfill three main security notions to be considered secure [MOV96]:

- **Authenticity**: The sender of the message is truly who he or she claims to be.
- **Integrity**: The sent message has not been altered on the way from sender to receiver.
- **Non-repudiation**: The creator of the signature cannot deny that the signature was created by him- or herself.

In recent decades there was a notable advancement in the development of quantum computers [BBD09]. At least since the discovery of Shor's algorithm in 1994, we know that a sufficiently large and powerful quantum computer can break a considerable amount of our modern-day public-key encryption and signature schemes [Sho94]. Hence, once a sufficiently powerful quantum computer exists, all of our digital communication that relies on broken primitives becomes vulnerable and insecure. However, there are also approaches for public-key encryption, key exchange algorithms, and signatures schemes which are secure in this so-called post-quantum setting. It takes a development process of several years until a post-quantum scheme is usable in practice. Therefore, it is necessary to start the transition to post-quantum schemes as early

[2] `https://docs.microsoft.com/en-us/windows-hardware/drivers/dashboard/driver-signing`

[3] `https://www.digitales.oesterreich.gv.at/elektronische-signaturen#Elektronische_Signaturarten`

as possible. Up to now, there have been many approaches to construct post-quantum signature schemes which are built on different primitives. Examples of post-quantum signature schemes include lattice-based schemes like TESLA [Alk+15] and BLISS [Duc+13], hash-based schemes like SPHINCS [Ber+15], multivariate schemes like MQDSS [Che+16], and code-based schemes like the Niederreiter scheme [Nie86]. Furthermore, all signature schemes which are based on symmetric-key primitives are, to the best of our knowledge, secure in the post-quantum setting although their security level is reduced by one half. This reduction of security is due to the discovery of Grover's algorithm in 1996 [Gro96]. Grover's algorithm reduces the complexity of exhaustive key search attacks by a factor of $\frac{1}{2}$. Therefore, we have to double the block size of a symmetric-key algorithm to achieve the same security.

One feature of a signature scheme which is particularly important when it comes to the real world application is its performance. A perfectly secure post-quantum signature scheme is only of theoretical interest if its performance is too poor or its memory requirements are too high for practical usage. Especially on less powerful devices like embedded devices or smartphones, it is essential that the performance of the scheme not hinder the regular workflow. ECDSA achieves a performance in the order of microseconds which is not noticeable by a human being. Many post-quantum signature schemes suffer either from large keys, large signatures or long execution time [Cha+17a]. There is still a vast optimization potential until these schemes are as optimized as ECDSA is nowadays.

The National Institute of Standards and Technologies (NIST) launched a project[4] which aims to find secure key exchange, public-key encryption and signature algorithms. The deadline for the submissions was in November 2017 and in total 69 algorithms where submitted. The Institute of Applied Information Processing and Communications (IAIK) at Graz University of Technology submitted the signature scheme Picnic [Cha+17a].

Picnic is a symmetric-key based, post-quantum signature scheme which is based on the symmetric-key primitive LowMC [Alb+16b]. This thesis focuses on the optimization of LowMC in the context of Picnic so that Picnic becomes usable in practice. We propose two structural optimizations and one

[4] http://csrc.nist.gov/groups/ST/post-quantum-crypto/

implementational optimization. The structural optimizations focus on two different bottlenecks of LowMC. The first optimization targets the computation of the round key layer, which is a costly part of LowMC. We propose to split the round key computation such that a part of it can be precomputed for multiple encryptions without decreasing the security of Picnic. The second optimization targets the matrix multiplication in the linear layer of LowMC. Our proposal substitutes the multiplication with a Feistel network which implements the multiplication through its round functions. Furthermore, we propose to implement parts of the Picnic code using the NEON instruction set extension of ARM CPUs. This instruction set extension implements single instruction, multiple data intrinsics which allow performing the same operation on multiple data instances at the same time. All of these optimizations pursue the same target, which is to make Picnic usable in practice. First, they aim to optimize the performance of Picnic so that it is no handicap in a regular workflow. Second, they try to minimize the memory consumption of the signature scheme. The reduction of the memory requirements should make Picnic usable on platforms with limited available memory, *e.g.,* embedded devices.

1.1. Structure of this Document

Chapter 2 includes background information on signature schemes, post-quantum cryptography, LowMC, Picnic, Feistl networks and single instruction, multiple data instruction set extensions. In Chapter 3, we describe the optimizations that we propose for LowMC. The performance and the memory requirements of these optimizations are evaluated in Chapter 4. Chapter 5 briefly discusses the results and implications of Chapter 4. Finally, Chapter 6 concludes the works.

2. Preliminaries

This chapter describes all necessary background information required in the remainder of this thesis.

2.1. Signature Schemes

Whenever we want to confirm our identity for whatever reason, *e.g.,* a contract, we provide a hand-written signature. This signature binds us to the content of the document we signed and we cannot deny that the signature is truly ours. Nowadays, a huge part of our daily life includes digital communication. To be able to identify ourselves digitally to our communication partner we need a digital equivalent to the hand-written signature. This is where digital signature schemes come into play. A digital signature scheme consists of three algorithms: Keygen, Sign, and Verify [Kat10]. The Keygen algorithm is used to create the secret and the public key for creating and verifying signatures with respect to a certain security parameter 1^κ, *i.e.,* $(sk, pk) \leftarrow \mathsf{Keygen}(1^\kappa)$. The person who signs a message uses the secret key sk to create a signature σ for the message m, *i.e.,* $\sigma \leftarrow \mathsf{Sign}(m, sk)$. Whoever wants to verify the signature can do this by using the public key pk. The verification is either successful ($\top$) or not ($\bot$), *i.e.,* $\{\top, \bot\} \leftarrow \mathsf{Verify}(\sigma, m, pk)$. Commonly used signature schemes include RSA [RSA78], the ElGamal signature scheme [Gam84], the (elliptic curve) digital signature algorithm (DSA [ST93] and ECDSA [ST13]), and Edwards-curve digital signature algorithm (EdDSA [JL17; Ber+12; Ham15]).

Signature schemes need to fulfill EUF-CMA security to be considered secure [Kat10]. EUF-CMA describes existential unforgeability under adaptively chosen message attack. In this setting a polynomially bounded adversary A is

allowed to query the signatures from a signing oracle O^S for any desired messages. If the scheme is EUF-CMA secure, the adversary cannot produce a valid signature for a message that was not queried except for a negligible probability. A function $\epsilon(x)$ is called negligible if for every polynomial $p(x)$ there exists an n such that $\epsilon(n) < \frac{1}{p(n)}$. EUF-CMA secure can more formally be described as

$$\Pr\left[\begin{matrix}(pk, sk) \leftarrow \mathsf{Keygen}(1^\kappa), (m^*, \sigma^*) \leftarrow A^{O^S(\cdot, sk)}(pk) : \\ m^* \notin Q \wedge \mathsf{Verify}(m^*, \sigma^*, pk) = \top\end{matrix}\right] \leq \epsilon(\kappa).$$

2.2. Post-quantum Cryptography

Many present-day cryptographic signature and public-key encryption algorithms rely on specific hard mathematical problems like the integer factorization problem, the discrete logarithm problem and the elliptic curve discrete logarithm problem. A hard problem is defined as a problem to which no polynomial algorithm is known to solve it. Examples for algorithms that are based on these problems include RSA, DSA, ECDSA, the ElGamal signature scheme [ST13], and the (elliptic curve) Diffie-Hellman key exchange (DH [DH76] and ECDH [Mil85]). In 1994, Shor proposed an algorithm which is capable of efficiently solving the discrete logarithm and the factorization problem on a powerful enough quantum computer [Sho94]. To be more specific, there is no known algorithm for classical computers that can factor a number in polynomial time, whereas Shor's algorithm can factor a b-bit number in approximately $O(b^3)$. The algorithm can also solve the discrete logarithm problem in polynomial time by building on the solution of the factorization problem. Therefore, once a quantum computer exists that solves the mentioned problems in a reasonable amount of time, all cryptographic primitives that build on these problems are insecure. All applications where the insecure algorithms are used become vulnerable. To overcome the threat of quantum computers there exist new approaches for building public-key primitives which are secure against attacks from quantum computers. Post-quantum cryptography refers to cryptographic algorithms which are considered secure against attacks using quantum computers [BBD09]. These approaches include, among others, symmetric-key-based, hash-based, lattice-based, multivariate, and code-based signature schemes.

The development of quantum computers is still in its infancy, and it will be years or even decades until a quantum computer can be used to break modern cryptography standards [BBD09]. However, once such a quantum computer exists, all cryptographic algorithms that are based on the problems presented above can be broken. NIST already launched a post-quantum cryptography project to find suitable key exchange, public-key encryption and signature algorithms for the post-quantum era. According to the proposed timetable, the selection of one or multiple suitable post-quantum schemes will take at least five years.

Symmetric-key-based Signatures. To current knowledge, symmetric key primitives cannot be broken by quantum computers because they do not rely on integer factorization or the discrete logarithm problem [BBD09]. However, in 1996 Grover proposed an algorithm for quantum computers which allows to search in a unsorted database with N entries in $O(\sqrt{N})$. Compared to classical computers, which can search in unsorted databases in $O(N)$, this is a quadratic speedup. This means that Grover's algorithm can recover a 256-bit key for a symmetric cipher using brute-force with 2^{128} trials instead of 2^{256}. To make symmetric key primitives secure against attacks from a quantum computer, we can double the security parameter for symmetric key primitive. In other words, we can double the key size for symmetric key primitives to achieve the same security of the primitives on quantum computers as we have today on classical computers. How symmetric key primitives can be used to create a signature scheme will be covered in detail in Section 2.4.

Hash-based Signatures. Hashes are believed to be post-quantum secure if the hash size is doubled to avoid attacks using Grover's algorithm. Hash-based signature schemes use a one-way function like a hash as a primary building block because hashes provide collision resistance [BBD09]. Collision resistance means that no two inputs to a hash function lead to the same output except for a small probability. Merkle was the first to propose a practically usable hash-based signature scheme [Mer89]. It uses a one-time signature scheme, like the Lamport-Diffie scheme [Lam79], as a basis. With a one-time signature scheme a pair of a private signing and a public verifying key can only be used to sign one single document because the computed signature reveals part of the signing key. Merkle uses the Lamport-Diffie scheme as a building block for his signature scheme and builds a hash tree to extend the validity of the signature key. In 2015 Bernstein et al. published the hash-based signature

scheme SPHINCS [Ber+15] which is the state-of-the-art hash-based signature scheme.

Lattice-based Signatures. Lattice-based signatures use lattices as their basis, which can informally be described as a set of points in an n-dimensional space, that follows a periodic structure [BBD09]. The security of lattice-based cryptography is based on the hardness of particular problems in the lattice space, *e.g.,* short integer solution problem (SIS) and learning with errors problem (LWE). The SIS problem is defined as follows. Let $A \in \mathbb{Z}_q^{n \times m}$ be a random matrix. The goal in the SIS problem is to find a non-zero vector $x \in \mathbb{Z}_q^m$ such that $A \cdot x = 0$. The definition of the LWE problem is as follows. Let $A \in \mathbb{Z}_q^{n \times m}$ be a random matrix and $s \in \mathbb{Z}_q^n$ be a secret vector. Furthermore, we require $e \in \mathcal{E}_q^n$ and the calculation $g = A \cdot s + e \bmod q$. Given A and g, the goal is to find the secret vector s. The SIS and the LWE problem are believed to be hard to solve efficiently on a classical and a quantum computer. Therefore, there exist several signature schemes which build on these problems, and the Fiat-Shamir transform (cf. Section 2.4) [Alk+15; Lyu09; Lyu12]. Ajtai was the first to propose a lattice-based cryptographic scheme together with the hardness analysis of the SIS problem [Ajt96]. However, lattice-based signature schemes often suffer from immense key sizes up to several megabytes [Alk+15; Lyu12; Dag+14; GPV08]. There exists a variety of efficient schemes which do not rely on classical lattice problems, but on their ring analogues [Akl+16; BB13; Bar+16; GLP12]. One major advantage is that these constructions reduce the key size from several megabytes to kilobytes. Ducas et al. proposed such a signature scheme, which is highly efficient, under the name BLISS in 2013 [Duc+13].

Multivariate Signatures. Multivariate cryptography builds on the problem of solving multivariate, non-linear equation systems over a finite field $\mathbb{F}$ [BBD09]. This problem is proven to be NP-complete, which is the basis of the security of these schemes. The public key $\mathcal{P}$ consists of a set of non-linear (usually quadratic), multivariate equations. To construct the private key, we require two affine transformations $\mathcal{S}$ and $\mathcal{T}$ and an easily invertible map Q. The relation between the public and the private key is $\mathcal{P} = \mathcal{S} \circ \mathcal{Q} \circ \mathcal{T}$. The signature s for a block x is calculated as $s = S^{-1}(Q^{-1}(T^{-1}(x)))$. The verification can be done by computing $x = P^{-1}(s)$. In 2016, Chen et al. proposed the signature scheme MQDSS, which is based on solving quadratic multivariate equations and the Fiat-Shamir transform [Che+16].

Code-based Signatures. Code-based cryptography builds upon error correcting codes (ECC). The first approach for a public-key encryption scheme based on ECC was made by McEliece in 1978 [McE78]. In 1986, Niederreiter published a signature scheme based on McEliece's encryption scheme [Nie86]. Many code-based signatures are based on the syndrome decoding problem. Informally, the syndrome decoding problem is defined as follows. Let $H \in \mathbb{F}_q^{x \times y}$ be a parity check matrix, i a vector of length x, and t an integer. The syndrome decoding problem describes the question if there is a vector e of length y such that $He = i$ and the Hamming weight of $e < p$. Véron proposed an identification system based on this problem [Vér96]. This identification scheme can be transformed into a signature scheme by applying the Fiat-Shamir transform.

2.3. LowMC

LowMC is a symmetric encryption scheme which is highly parameterizable and was proposed by Albrecht et al. [Alb+15; Alb+16b]. It is a block cipher which is based on a substitution-permutation network [KD79; MOV96]. The parameters of LowMC include the block size n, the key size k, the number of S-boxes m, and the number of rounds r.
One round of the LowMC encryption consists of a non-linear *S-box layer*, a *linear layer*, and the addition of a *round constant* and a *round key*. Each of these layers operates on a n-bit state $s \in \mathbb{F}_2^n$. This state is initialized with the plaintext and an inital key. LowMC requires several matrices in each round. Those matrices are described below in the corresponding section. We refer to a set of fixed parameters and matrices as a LowMC instance.

S-box Layer. LowMC's S-box layer consists of multiple 3-bit S-boxes and does not necessarily operate on the whole state. Although the number of S-boxes m can be chosen freely, $3m$ bits must not exceed the block size n. The m S-boxes operate on the lower $3m$ bits of the state s. The upper $n-3m$ bits, which are not influenced by the S-box, use an identity mapping, *i.e.,* they do not change. We call the lower $3m$ bits the *non-linear* part of the state and the remaining $n-3m$ bits the *linear* part. The S-box for the three input bits (a, b, c) is defined as $S(a, b, c) = (a \oplus bc, a \oplus b \oplus ac, a \oplus b \oplus c \oplus ab)$ [Alb+15].

Linear Layer. In the linear layer of round i the state s is multiplied by the linear layer matrix $L_i \in \mathbb{F}_2^{n \times n}$. The matrices L_i are chosen uniformly at random for all rounds i. However, all matrices L_i have to be invertible.

Round Constant. In each round i the round constant $C_i \in \mathbb{F}_2^n$ is added to the state. The round constants C_i are chosen uniformly at random and need not fulfill any special constraints.

Round Key. Each round contains an addition of the round key to the state. The round key for round i is calculated by multiplying the round key matrix $K_i \in \mathbb{F}_2^{n \times k}$ by the secret key $y \in \mathbb{F}_2^k$. The matrices K_i are chosen uniformly at random for all rounds i. All matrices K_i require rank $min(n, k)$. Additionally, we need one more key matrix K_0 for the initial key, which is also computed by multiplying the key matrix by the secret key. This initial key matrix has to fulfill the same constraints as all other round key matrices.

All matrices L_i, C_i and K_i are generated once, fixed for one instance of LowMC and do not necessarily have to be kept secret. In other words, a LowMC instance is a set of fixed, publicly known parameters and matrices which is used for multiple encryptions. Additionally to the LowMC matrices presented above, we require a secret key. The secret key y is chosen uniformly at random and has to be kept secret. However, in contrast to all other matrices, it is not fixed for one LowMC instance. Therefore, it is possible to execute LowMC encryptions with the same LowMC instance but with different secret keys, *e.g.,* secret keys from different people.

Algorithm 1 LowMC encryption for key matrices $K_i \in \mathbb{F}_2^{n \times k}$ for $i \in [0, r]$, linear layer matrices $L_i \in \mathbb{F}_2^{n \times n}$ and round constants $C_i \in \mathbb{F}_2^n$ for $i \in [1, r]$.

Require: plaintext $p \in \mathbb{F}_2^n$ and key $y \in \mathbb{F}_2^k$
1: $s \leftarrow K_0 \cdot y + p$
2: **for** $i \in [1, r]$ **do**
3: $\quad s \leftarrow \text{SBOX}(s)$
4: $\quad s \leftarrow L_i \cdot s$
5: $\quad s \leftarrow C_i + s$
6: $\quad s \leftarrow K_i \cdot y + s$
7: **end for**
8: **return** s

Algorithm 1 shows the full algorithm for the LowMC encryption. Line 1 shows the initialization of the state. Lines 3 to 6 show the non-linear S-box layer, the linear layer, the round constant addition and the round key addition, respectively.

As Albrecht et al. describe, LowMC aims to optimize two characteristics of a cipher: the multiplicative complexity and the ANDdepth [Alb+16b]. Both terms refer to the representation of the cipher as a boolean circuit. The multiplicative complexity describes the number of ANDs which is necessary to implement a given boolean circuit, *i.e.,* the total number of AND gates which will be used for building the circuit [BPP00]. The ANDdepth is the depth of the circuit. In other words, the depth of a circuit describes the length of the longest path between an input and an output gate. In LowMC the ANDdepth corresponds to the number of rounds. Doröz et al. already suggest that a low ANDdepth is necessary for an efficient implementation of an encryption scheme and demonstrate it on the example of PRINCE [Dor+14]. The goal of LowMC is to minimize the multiplicative complexity and the ANDdepth at the same time. This should yield a good tradeoff between a fast and a small hardware implementation. A low number of AND gates is desirable in a hardware implementation where the circuit should be implemented as small as possible, *e.g.,* on a microchip. The low ANDdepth favors a fast execution of the algorithm because the input has to pass fewer gates to reach the output. However, minimizing the multiplicative complexity and the ANDdepth at the same time is not a trivial problem [Alb+16b]. If the ANDdepth is optimized and decreased, the total number of gates increases, which makes the circuit bigger and requires more space on a chip. If the number of AND gates is optimized and decreased, the depth of the circuit increases, which makes the path between input and output gates longer and therefore the execution slower. Hence, it is necessary to find a good tradeofff between the characteristics. The number of rounds r of the encryption is an adaptable parameter of LowMC. It can be adapted according to the security level that should be reached. The number of S-boxes m, the block size n, and the key size k also have an impact on r for a certain security level. If the number of S-boxes m decreases, the number of rounds r has to be increased to reach the same security level. If the block size n or the key size k or both are increased, r also has to be increased. Increasing the number of S-boxes without changing the number of rounds, the block size, or the key size increases the security level.

Block size n	128				256				512			
S-boxes m	1	10	30	42	1	10	30	84	1	10	30	170
Rounds r	191	20	8	6	380	38	13	6	757	78	26	6
Data complexity d	1	1	1	1	1	1	1	1	1	1	1	1
ANDs/bit	4.48	4.69	5.63	5.91	4.45	4.45	4.57	5.91	4.44	4.57	4.57	5.98
S-boxes m	1	10	30	42	1	10	30	84	1	10	30	170
Rounds r	287	32	20	19	537	58	27	22	809	83	31	17
Data complexity d	128	128	128	128	256	256	256	256	256	256	256	256
ANDs/bit	6.73	7.5	14.06	18.7	6.29	6.8	9.49	21.66	4.74	4.92	5.45	16.93

Table 2.1.: LowMC parameters for different block sizes, number of S-boxes and data complexities.

Also increasing the number of rounds without changing the block or key size, or the number of S-boxes has a positive impact on the security level. However, increasing the block size without adapting the number of S-boxes or rounds decreases the security level.

The data complexity d is a measure of the security of a cipher [MOV96]. 2^d describes the number of plaintext-ciphertext pairs, which are generated with the same key, the attacker is allowed to use for mounting an intended attack. The higher the data complexity, the more the attacker can learn from the given plaintext-ciphertext pairs. For example, if the relation between the plaintext and the ciphertext can be described as an equation, more equations may help to solve the equation system.

Table 2.1 shows the calculated number of rounds for data complexity 1 and 256 according to Albrecht et al. for different block sizes and numbers of S-boxes [Alb+16b]. The instances with one S-box have the lowest number of AND gates in the implemented circuit. Using as many S-boxes as possible decreases the number of necessary rounds to 6 for all proposed block sizes when a data complexity of 1 is used. Furthermore, Table 2.1 shows ANDs gates that are required per bit of the block size. The ANDs/bit can be calculated by dividing the total number of AND gates in the circuit by the block size, *i.e.*, $\frac{3 \cdot m \cdot r}{n}$.

2.4. Picnic

Picnic is a signature scheme, which aims to achieve security in a post-quantum era [Cha+17a]. It can be classified as a symmetric-key based signature algorithm. Picnic is based on the symmetric cipher LowMC, which is believed to be post-quantum secure, and uses it as a one-way function. The core building blocks of Picnic are described in the following sections.

2.4.1. Zero-knowledge Proofs

A zero-knowledge proof is a special form of proof system which tries to minimize the amount of knowledge that is exchanged during the proof [GMR85]. The prover P can convince a verifier V that he or she "knows something" but does not reveal this actual knowledge [Qui+89]. V has to be convinced of P's knowledge by using the zero-knowledge protocol without learning the explicit knowledge of P. We use the term "perfect zero-knowledge" to refer to a zero-knowledge proof which truly does not leak any additional, unintended information. Cramer, Damgård, and MacKenzie investigate on efficient methods for such perfect zero-knowledge proofs [CDM00].

A zero-knowledge proof has to fulfill the three properties completeness, soundness and zero-knowledge [GMR85]. If one of these three properties is not fulfilled, the proof cannot be considered a valid proof system.

- **Completeness.** If an honest prover P indeed possesses the knowledge that P wants to prove to the honest verifier V, P can always convince V of this fact. An honest prover and an honest verifier do not cheat and follow the protocol as intended.

- **Soundness.** If P is cheating, an honest V cannot be convinced that P possesses the claimed knowledge, except for a negligible probability.

- **Zero-knowledge.** After the execution of the protocol, an honest verifier V learns nothing apart from the fact that P truly possesses the proven knowledge.

Σ-Protocol. A Sigma protocol (Σ-protocol) is a form of zero-knowledge proof which is interactive. Interactive zero-knowledge proofs use messages (interactions) between a prover P and a verifier V. A Σ-protocol is a three-move protocol between P and V, where P proves "some knowledge" to V. After executing the protocol, P has proven "some knowledge" to V, and V is convinced that P truly "possesses this knowledge". This standard definition of a "proof of knowledge" was proposed by Bellare and Goldreich [BG92].

Schnorr proposed the first efficient Σ-protocol in [Sch89]. To prove the knowledge, P sends a random commitment a to V. The commitment should prevent P from cheating by binding him to the fresh execution of the protocol. V responds with a challenge e which has to be unpredictable. The challenge should be constructed in such a way that P can only solve it if he truly has the knowledge that he claims to have. P then computes the response z, which should convince V of P's knowledge. The transcript of this protocol, which can be seen by an observer, is (a, e, z).

Example 2.1. Schnorr proposes a Σ-protocol that operates on a cyclic group G with the generator g in which the discrete logarithm problem is hard [Sch89]. The protocol allows the prover P to prove to the verifier V that he knows x such that $y = g^x$, *i.e.,* P proves the knowledge of the discrete logarithm x. Finding the discrete logarithm is considered a hard problem and no known, classical (not quantum) algorithm exists which is able to compute the discrete logarithm efficiently. y, the generator g, and the prime p are public. The protocol works as follows

1. P chooses a random commitment r and sends $a = g^r$ to V.
2. V chooses a random challenge e to P.
3. P sends $z = r + e \cdot x$ to V.

V accepts the proof of knowledge if $g^z = a \cdot y^e$, since

$$g^z = g^r \cdot g^{e \cdot x} = g^r \cdot (g^x)^e = a \cdot y^e.$$

Fiat-Shamir Transform. Σ-protocols are interactive proofs of knowledge. However, to create a signature scheme, we require a non-interactive zero-knowledge proof (NIZK). This non-interactive proof can be achieved by applying the Fiat-Shamir transform [FS86] on the interactive Σ-protocol. It is

important to consider which properties the Σ-protocol has to fulfill to create a NIZK proof [Fau+12]. These properties include an exponentially large challenge space and a minimal entropy α of the commitment r such that $2^{-\alpha}$ is negligible with respect to the chosen security parameter. In the transform, the same messages a, e, and z as in the Σ-protocol are used. To make the proof non-interactive, the transform needs a random oracle R [BR93]. The random oracle returns a uniformly distributed value for each input that is unique, *i.e.,* the same input yields the same output, but two different inputs should not yield the same output. However, a random oracle can collide with the probability 1 : *Size of output space.* If the input space is larger than the output space, a collision is inevitable once all possible outputs have been generated. In the non-interactive proof version the challenge e is not computed by the verifier V but by the prover himself using $e \leftarrow R(a)$. Hence, the transcript of the protocol for an observer is $(a, R(a), z)$.

Example 2.2. If the Fiat-Shamir transform is applied to Schnorr's protocol, which was presented in Example 2.1, the challenge e is computed as $e \leftarrow R(a) = R(g^r)$. The response z can further be computed as

$$z \leftarrow r + e \cdot x = r + R(a) \cdot x = r + R(g^r) \cdot x.$$

To build a signature scheme using the Fiat-Shamir transform, it is necessary to include the message, which should be signed by the signature scheme, into the protocol. This can be done by including the message m into the calculation of the challenge such that $e \leftarrow R(a, m)$ [Der+16]. The signature consists of the challenge and the response. The Fiat-Shamir transform cannot only be used to transform a Σ-protocol into a signature scheme, but it can also be applied to an identification scheme [Abd+02; Abd+12; BPS16; Dag+16; KMP16; OO98; PS96]. For the verification of the signature, the verifier recalculates the challenge and the response. If the recalculated response of the verifier and the response in the signature are the same, the signature is valid.

Example 2.3. As a continuation of Example 2.2 we now construct a signature based on the Schnorr protocol. The private key in our signature scheme is the discrete logarithm x and the public key is $y = g^x$. All values that are computed by the prover who creates the signature are subscripted with s, whereas all values computed by the verifier are subscripted with v. To sign the message

m we calculate the challenge e_s as follows ($||$ denotes concatenation)

$$e_s \leftarrow R(a_s||m) = R(g^{r_s}||m)$$

As the next step we compute the response z_s.

$$z_s \leftarrow r_s - e_s \cdot x = r_s - R(g^{r_s}||m) \cdot x$$

The signature is the pair (e_s, z_s). To verify a signature the verifier has to compute r_v and e_v.

$$r_v \leftarrow g^{z_s} \cdot y^{e_s}, e_v \leftarrow R(r_v||m)$$

The verifier accepts the signature if $e_v = e_s$.

$$\begin{aligned} e_v &= R(g^{z_s} \cdot y^{e_s}||m) = R(g^{r_s - e_s \cdot x} \cdot g^{x e_s}||m) = R\left(\frac{g^{r_s}}{g^{e_s \cdot x}} \cdot g^{x \cdot e_s}||m\right) \\ &= R(g^{r_s}||m) = R(a_s||m) = e_s \end{aligned}$$

A one-way function can be described as a function which is easy to compute and hard to invert [MOV96]. This means that computing $y = f(x)$ given x is easy, whereas finding x given y such that $y = f(x)$ is hard. A signature scheme can be created from such a one-way function and a NIZK proof in the following way. Let f be a one-way function, x the secret key which is used for signing the message, and y the public key which is used for verifying the signature. y is defined as $y = f(x)$. The signature is the NIZK proof that proves the knowledge of the secret key x such that $y = f(x)$. As previously mentioned, the message m to be signed is included in the challenge calculation of the NIZK proof.

In Picnic we use the LowMC encryption as the one-way function f. We denote the encryption of a value p to the ciphertext c under the key k as $c = f_k(p)$. The secret key x for signing the message m is generated randomly. As described above, the public key y can be calculated by applying the one-way function f on x, *i.e.*, $y = f(x)$. In Picnic y is the LowMC encryption of a randomly generated value r with the secret key x, *i.e.*, $y = f_x(r)$. The signature is the transcript of the NIZK proof which proves the knowledge of the secret key x [Der+16].

The Fiat-Shamir transformed signature scheme achieves security in the classical random oracle model (ROM) [Cha+17a]. The Picnic version which uses

the Fiat-Shamir transform will be referred to as Picnic-FS in the remainder of the thesis.

Unruh Transform. The Fiat-Shamir transform makes Picnic secure in the classical ROM. However, schemes which are secure in the ROM are not necessarily secure in the quantum random oracle model (QROM) [Bon+11]. Therefore, we need a new approach if we want to achieve security in the QROM. Unruh proposed a transformation which also transforms interactive proofs into non-interactive proofs [Unr12; Unr15; Unr16]. In contrast to the Fiat-Shamir tansform, the Unruh transform provides security in the QROM. The Picnic variant using the Unruh transform will be referred to as Picnic-UR.

The Unruh transform requires a statement x, a challenge space $\mathcal{C}$, an integer t, a random permutation G, and a one-way function f. First, t values $r_1, \cdots, r_t$ are generated by executing the first step of the Σ-protocol t times. Second, for each $i \in [1,t]$ and for each challenge $c \in \mathcal{C}$ the response s_{ic} is computed. Then the permutation G is applied to all values s_{ic} by calculating $g_{ic} = G(s_{ic})$. As the next step, the values $J_1, \cdots, J_t = H(x, r_1, \cdots, r_t, g_{11}, \cdots, g_{t|\mathcal{C}|})$ are calculated. These values $J_1, \cdots, J_t$ are used as indices for the responses in the final signature. They determine for each $i \in [1,t]$ which of the $|\mathcal{C}|$ responses is used for the signature, *i.e.,* J_1 determines which element of $\{s_{11}, \cdots s_{1|\mathcal{C}|}\}$ is included in the signature. In the end, it outputs the signature $\sigma = \{r_1, \cdots, r_t, s_{1J_1}, \cdots, s_{tJ_t}, g_{11}, \cdots, g_{t|\mathcal{C}|}\}$. To verify the signature, the verifier V first recomputes the indices $J_1, \cdots, J_t$. Then V verifies that the values g_{iJ_i} are generated by the corresponding values s_{iJ_i}. Furthermore, the values s_{iJ_i} values need to be valid responses for the values r_i. To create a signature, we can include the message in the calculation of the challenge, *i.e.,* the indices $J_1, \cdots, J_t$.

2.4.2. ZKBoo

ZKBoo is a Σ-protocol which allows for the construction of zero-knowledge proofs on arbitrary circuits [GMO16]. In the case of Picnic, the knowledge which should be proven is the knowledge of the secret key sk. The idea is based on MPC-in-the-HEAD which is described in [Ish+09].

Multiparty Computation. Multiparty computation (MPC) describes a process to compute $y = f(x)$ where $f(x)$ is not evaluated directly but the computation is split over multiple parties. In general, x is split into multiple shares x_i and each party evaluates $f(x_i)$ for one share x_i. In the end, y is reconstructed from the results of all parties. One important thing to note is that each party does not learn anything about the original input x. Therefore, one corrupted party alone cannot disclose any sensitive information because one share of x does not reveal any secret. In MPC-in-the-HEAD the prover simulates the multiparty computation by calculating all shares himself and committing to their so-called views, which are the intermediate states. The verifier chooses two views randomly, requests the prover to 'open' these views and checks if they are calculated correctly. The opening of the views serves the purpose that the verifier can convince himself that the prover did not cheat in the calculation of the views. The only possibility that the prover cheats undetected is in the view that is not opened. If the prover corrupts the not opened view, he can cheat on the verifier and convince him of his honesty at the same time. However, repeating the protocol decreases the chance of a cheating prover because he would have to guess the share that is not opened in each run of the protocol correctly. The procedure is repeated as often as necessary to get a certain probability that the prover did not cheat.

(2,3)-Decomposition of Circuits in Picnic. ZKBoo uses a so-called (2,3)-decomposition of the LowMC circuit. A (2,3)-decomposition is a protocol where three simulated parties evaluate the shares of the input. Intuitively, correctness is the most important property of this decomposition. Informally, this means that if the calculation $y = f(x)$ is split over the three shares, the result y still has to be correct if the individual results of the shares are combined to the final result. Another essential property is 2-privacy, which means that leaking two of the three shares does not disclose any information. Therefore, even if two out of three shares are leaked the adversary does not learn anything about the initial input to the decomposition. These properties and their validity are described in more detail by Giacomelli, Madsen, and Orlandi [GMO16].

The decomposition consists of the four functions `Share`, `Update`, `Output`, and `Reconstruct` [GMO16; Cha+17a]. Figure 2.1 shows the (2,3)-decomposition of a circuit using these four functions. The functions are defined as follows.

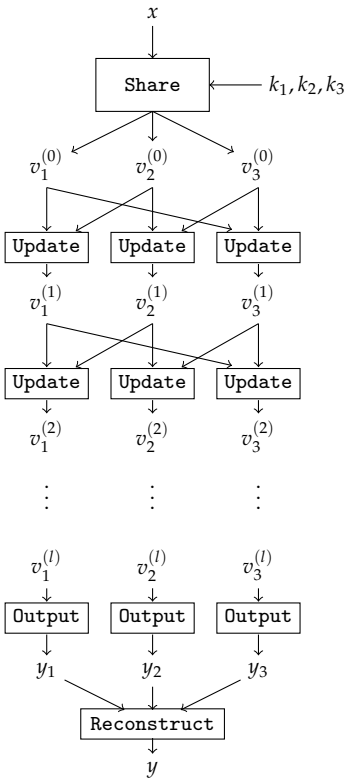


Figure 2.1.: (2,3)-decomposition of a circuit calculating $f(x)$.

$$
\begin{aligned}
(\mathsf{view}_1^{(0)}, \mathsf{view}_2^{(0)}, \mathsf{view}_3^{(0)}) &\leftarrow \texttt{Share}(x, k_1, k_2, k_3) \\
\mathsf{view}_i^{(j+1)} &\leftarrow \texttt{Update}(\mathsf{view}_i^{(j)}, \mathsf{view}_{i+1}^{(j)}, k_i, k_{i+1}) \\
y_i &\leftarrow \texttt{Output}(\mathsf{view}_i) \\
y &\leftarrow \texttt{Reconstruct}(y_1, y_2, y_3)
\end{aligned}
$$

The `Share` function takes three random bit streams k_1, k_2, and k_3 and the input value x as parameters. It splits x into three shares using the random bit streams and outputs the initial shares, which will be the starting view $\mathsf{view}_i^{(0)}$ for all parties i. In Picnic, the `Share` function uses only k_1 and k_2 for sharing the secret key x. The first and the second share are initialized with the first k bits of k_1 and k_2. The third share is calculated as $x \oplus k_1 \oplus k_2$.
The `Update` function updates all views by simulating the gates of the implemented circuit. If the simulated gate is an XOR, we do not have to include the intermediate value into the views. An XOR of two values a and b can directly be computed by applying the XOR on the shares of the two values, *i.e.,* $c_i = (a \oplus b)_i = a_i \oplus b_i$. If all shares c_i are combined again, we have the direct XOR of the non-shared values a and b. In contrast, an AND gate cannot be computed that easily. Computing a AND b in the shared setting cannot be done by simply applying the AND to the shares of a and b. That is because combining all shares after applying the AND to all shares of a and b does not yield the AND of the non-shared values a and b. To calculate a AND b in the shared setting we have to include the value of the neighboring share for calculating each share. Furthermore, we use the random bit streams k_1, k_2, and k_3, which were already used in the `Share` function, to blind the AND operation. The notation $\overline{k_i}$ refers to the next unused bit of the bit stream k_i. This results in $c_i = (a \wedge b)_i = (a_i \wedge b_i) \oplus (a_i \wedge b_{i+1}) \oplus (a_{i+1} \wedge b_i) \oplus (\overline{k_i} \wedge \overline{k_i}) \oplus (\overline{k_i} \wedge \overline{k_{i+1}}) \oplus (\overline{k_{i+1}} \wedge \overline{k_i})$. If we now combine all shares c_i we get the result $a \wedge b$, which is the evaluation of the AND on the non-shared values a and b. All layers of Picnic except the S-box layer use only XOR operations which operate on one share. However, the S-box includes AND gates which have to be simulated in the views. As mentioned above, an AND operation does not only operate on one share but requires multiple shares. Therefore, we have to store the output of each S-box as the next view for each share. This guarantees that all shares are synchronized all the time.
After all views have been calculated completely, the `Output` function takes

Block size	**Key size**	**S-boxes**	**Rounds**
n	k	m	r
128	128	10	20
192	192	10	30
256	256	10	38
384	384	10	57
512	512	10	78

Table 2.2.: Parameters of LowMC instances for different Picnic instances.

the whole view from one share and calculates the final output value for this share. Hence, the `Output` function returns the calculated value $f(x_i)$ for each share. In the end, all output values of the three shares are combined in the `Reconstruct` function, which returns the final result of the circuit calculating $y = f(x)$. In Picnic, the `Reconstruct` function takes all three output shares from the `Output` function and applies an XOR on them.

2.4.3. Parameter Instances of Picnic

Table 2.2 shows the LowMC instances that are used in the Picnic implementation. We focus on instances with 10 S-boxes only because it has significant implementational advantages. These advantages will be discussed in detail in Section 3.1.1. We can use instances which have a data complexity d of 1. The data complexity d of 1 allows the adversary to see $2^d = 2^1$ plaintext-ciphertext pairs for the attack. $d = 1$ is sufficient for Picnic because it only generates one plaintext-ciphertext pair which can be seen by the adversary. This plaintext-ciphertext pair is the public key which is the encryption of a random value.

2.4.4. Variations of Picnic

The construction of Picnic is highly modular which facilitates the exchange of several building blocks. The used one-way function does not necessarily have to be LowMC, ut can be replaced by for example MiMC [Alb+16a]. Chase et al. evalute several possible one-way functions [Cha+17a]. Furthermore, it

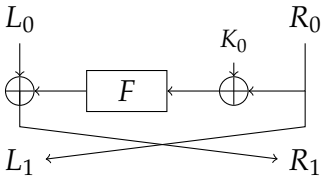


Figure 2.2.: One round of a Feistel network

is possible to create ring signatures if accumulators are used [RST01; BD93; DRS17]. The zero-knowledge proof of the preimage of the one-way function is replaced by a zero-knowledge proof of the membership of an accumulated value in the accumulator.

2.5. Generalized Feistel Networks

Feistel networks are iterative structures which are commonly used in block ciphers [MOV96]. Horst Feistel proposed them first in the cipher Lucifer [Fei73]. One of the most famous examples of a cipher using a Feistel network is DES, which was the state-of-the-art encryption algorithm for many years. It was standardized by NIST in 1976 [ST77]. Until 1999 the standard was published in three updated versions, *e.g.,* TripleDES [ST99]. Other examples of block ciphers, which are based on Feistel-like structures, include Blowfish [Sch93], GOST 28147-89 [GOS89], and RC5 [Riv94]. Figure 2.2 shows one round of a standard Feistel network. The input is split into two blocks, L_0 and R_0, which are usually equally sized. In each round i, R_i and the round key K_i are used as input for the round function F. The result of the round function is then combined with L_i using an XOR. After this XOR, R_i and the modified L_i are swapped. Hence, the complete round of the Feistel network can be described as $L_{i+1} \leftarrow R_i$, $R_{i+1} \leftarrow L_i \oplus F(R_i \oplus K_i)$.

The two blocks L and R usually have the same size and the round function F has exactly as many output bits as input bits. However, Schneier and Kelsey suggested so-called unbalanced Feistel networks where both blocks are not equally sized and F has a different number of input and output bits [SK96].

One important characteristic of a Feistel network is that it provides full diffusion [ST14], *i.e.,* all output bits depend on all input bits. If one input bit changes, each output bit has a probability of $\frac{1}{2}$ of changing. We describe two approaches to generate this diffusion.

One approach is that two neighboring S-boxes partly use the same input bits. For example, the input to one S-box consists of two bits which are also input to the previous S-box, two unique bits, and two bits which are also used in the next S-box. This sharing of the input generates diffusion and is used in DES. Additionally, in DES all bits are permuted after the S-boxes to increase the diffusion. The permutation is done in such a way that the four output bits of one S-box are spread across four different S-boxes in the next round [ST77].

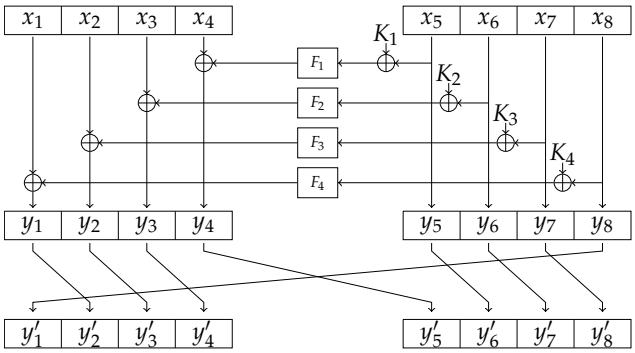


Figure 2.3.: One round of a generalized Feistel network. Both input halves are split into four blocks. After the round function, the blocks are rotated by one. [Nyb96]

Kaisa Nyberg proposed a new variant of balanced Feistel networks which she called Generalized Feistel Networks (GFN) [Nyb96]. A GFN relies on a different approach to generate diffusion than DES which was described above. The round function in this network consists of several parallel S-boxes. The most significant difference to a standard Feistel network is that the output of the S-boxes is split into smaller blocks which are then permuted. After this permutation, the two halves are swapped like in a standard Feistel network. This structure also guarantees a full diffusion in the network. A generalized Feistel network where both input halves are split into four blocks each is shown in Figure 2.3.

2.6. Single Instruction, Multiple Data Instruction Set Extensions

Flynn introduced a classification of computer architectures which is known as Flynn's taxonomy [Fly72]. The taxonomy divides computer architectures into four types:

1. **Single instruction, single data (SISD).** A SISD architecture executes a sequential stream of instructions on a single data element. This architecture does not offer any type of parallelism.

2. **Single instruction, multiple data (SIMD).** In a SIMD architecture, a sequential instruction stream is applied to multiple data elements in parallel. Parallelizing the same instruction on multiple data instances is particularly useful in multimedia and graphics processing in the GPU. For example, a SIMD architecture can apply a transformation to a picture much fast than regular instructions because they process parts of the picture in parallel.

3. **Multiple instruction, single data (MISD).** A MISD architecture executes multiple instructions on the same data element. This architecture may be used for fault-tolerant systems by executing the same instruction on the same data element several times and checking the consistency of the results. A famous example of a MISD system is the Space Shuttle [GS84].

4. **Multiple instruction, multiple data (MIMD).** A MIMD architecture achieves parallelism by applying multiple instruction streams on multiple data elements. Such an architecture typically is a multiprocessor system where each processor operates independently. The processors may share their memory or use separate memory spaces.

Instruction Set Extensions. Modern CPUs provide SIMD instruction set extensions to optimize their performance by using the provided parallelism. Two commonly used instruction set extensions on Intel CPUs are the Stream-

ing SIMD Extensions (SSE)[1] and the Advanced Vector Extensions (AVX)[2]. SSE uses 128-bit registers which are interpreted as four 32-bit floating-point numbers, *i.e.,* one operation can be performed on four floating-point numbers at once. SSE2 extends the use of the 128-bit registers by two 64-bit floating-point numbers, two 64-bit integers, four 32-bit integers, eight 16-bit short integers or sixteen 8-bit characters. The integer interpretation of the registers in SSE2 allows to perform SIMD operations on integers, which is not possible in SSE.

Example 2.4. An easy example for an operation exploiting the SIMD parallelism is vector addition. We will present the vector addition based on a four-dimensional example. Adding the vectors $v = (v_1, v_2, v_3, v_4)$ and $w = (w_1, w_2, w_3, w_4)$ conventionally with SISD requires four additions, $v_1 + w_1, v_2 + w_2, v_3 + w_3, v_4 + w_4$. Furthermore, the CPU has to load the two elements v_i and w_i of both vectors for the addition and store the result of the addition afterwards. Using a SIMD instruction set extension enables us to operate on all four elements of one vector at once, *e.g.,* in AVX we can use the 256-bit register as four 64-bit floats where each of the four floats stores one element of the vector. If both vectors have been loaded by the CPU, we can add them using one single addition. This addition interprets the AVX register as four independent values and executes the addition on all four values at once.

```
1 __mm256d v = _mm256_set1_pd(16.0);
2 __mm256d w = _mm256_set1_pd(15.0);
3 __mm256d result = _mm256_add_pd(v, w);
```

Listing 2.1: Code that initializes two AVX variables and adds them as two 4-dimensional float vectors.

The code in C could look as in Listing 2.1. The initialization of the AVX registers in lines 1 and 2 is only done for demonstration purposes. All elements of vector v and w are set to 16 and 15, respectively. Adding the four elements of the two vectors is done in line 3. To sum up, instead of eight load operations the SIMD variant needs two, instead of four additions it needs one, and instead of four store operations it needs one.

[1] https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html

[2] https://software.intel.com/en-us/isa-extensions/intel-avx

SSE3, SSSE3 (Supplemental SSE3) and SSE4 are enhancements of SSE2 which add more instructions. SSE4 was announced in late 2006 by Intel in a white paper [Ram+06]. It can be split into two instruction set extensions, SSE4.1 and SSE4.2, which offer a different subset of instructions and together form SSE4.

AVX is an extension of the SSE family which increases the register size from 128 to 256 bits. It was first introduced in Intel's Sandy Bridge microarchitecture. AVX also introduces three-operand instructions, which are not available in any SSE version. This means that an instruction can involve three registers instead of two. Therefore, calculations like $a = b + c$ are possible with AVX, whereas SSE only provides the possibility to calculate $a = a + b$, which always changes the content of one of the involved register. AVX2 was introduced with Intel's Haswell microarchitecture and enhances the AVX instruction set extensions. In 2015, Intel released its Knights Landing microarchitecture, which is part of the Xeon Phi product series, with the latest AVX version, AVX-512. AVX-512 extends the register size to 512 bits and implements new instructions.

Also other CPUs include SIMD instruction set extension. PowerPC offers AltiVec[3] which is also called Vector Multimedia Extension (VMX). AltiVec uses 128-bit registers which can mostly be used in the same variants as the SSE2 registers. The main difference in the register usage is that AltiVec does not offer 64-bit floating-point numbers, but instead a pixel data type which facilitates the storage and handling of of RGB values. ARM CPUs offer a SIMD instruction set extension called NEON[4]. NEON is included in the Cortex-A series and Cortex-R52 processors in ARMv7 and ARMv8. It supports 64- and 128-bit registers. The registers can be used as 8-, 16-, 32-, and 64-bit integers and 32-bit floating-point values. The 64-bit integer representation of the 128-bit registers is only available on ARMv8, which is the first ARM 64-bit architecture.

[3] https://www.nxp.com/docs/en/reference-manual/ALTIVECPIM.pdf
[4] https://developer.arm.com/technologies/neon

3. Optimizations of the Linear Operations of LowMC

In this chapter, we describe the optimizations which were made on LowMC in the course of this master's thesis. The optimizations target different problems which will be described in the corresponding sections. However, they have one thing in common. They all try to optimize the linear computations on the state in LowMC.

The optimizations in this chapter were proposed in [Per+17] and are covered here in more detail. The notations which are used in this chapter are also based on [Per+17].

3.1. Splitting the Round Key Computation

One major bottleneck in the LowMC algorithm is the calculation of the round key addition in every round. The round key matrix K_i has to be multiplied by the secret key y in each round to compute the round key. An intuitive solution to this problem would be that $K_i \cdot y$ is precomputed once and that this precomputed value is used for all following encryptions. The precomputation could be done before any encryption takes place but as soon as the LowMC instance is fixed and the secret key is known. This proposal would remove the matrix multiplication from each LowMC round, and only the addition of the already computed round key to the current state would be necessary. However, Picnic uses a (2,3)-decomposition of the state (cf. Section 2.4.2) which does not allow this precomputation because y is divided into three new shares for each encryption process. Therefore, we try to exploit the fact that LowMC

only uses a partial S-box layer which does not operate on the whole state but only on a part of the state.

We denote the lower $3m$ bits of state s, the bits on which the S-boxes operate, as $\rho_N(s)$ and call this part of the state the *non-linear* part. The upper $n-3m$ bits are referred to as the *linear* part and $\rho_L(s)$. In general, $\rho_i^j(v)$ denotes the bits from index i to index j of the vector v. Therefore, the non-linear and the linear part can also be described as $\rho_N = \rho_1^{3m}$ and $\rho_L = \rho_{3m+1}^n$.

The linear part of the round key never interacts with any S-box. This fact makes it possible to reorder each LowMC round in such a way that the linear round key part is added before the S-box. One original LowMC round can be described as $s_i = L_i \cdot \text{SBOX}(s_{i-1}) + K_i \cdot y + C_i$. To achieve the reordering we first have to reorder the linear layer and the round key addition. For the reordering to be correct, the round key has to be multiplied by the inverse of the linear layer L_i^{-1}. Now the whole round key is added after the S-box layer and before the linear layer. The modified LowMC round s_i is $L_i(L_i^{-1} \cdot K_i \cdot y + \text{SBOX}(s_{i-1})) + C_i$ The linear part of the round key addition can now be moved past the S-box while the non-linear part stays after the S-box layer. Figure 3.1 shows one modified round.

The procedure of moving the linear part of the round key past the S-boxes can be repeated until the linear part of all round keys is at the beginning of the encryption.

So far, the linear part of the round key addition has been moved to the beginning of the encryption, and the non-linear part remains in each LowMC round. However, the original goal, to remove the multiplication $K_i \cdot y$ from each round, has not been achieved yet as the non-linear part of the round key is still calculated in each round. Before we have a look at the removal of this multiplication we discuss the calculation for the linear part in more detail. As we have to move the round key past the linear layer of each round, we need the inverse matrices of L_i for all rounds i. Furthermore, we only need the part of the inverse which influences the linear part of the round key so that we can set the first $3m$ rows to 0. These rows only influence the non-linear part and are therefore not relevant for the calculation of the linear part. This modified inverse is denoted as $\overline{L_i^{-1}}$ for round i. The modified inverse can be used to combine the linear part of the round keys of all rounds. This combined

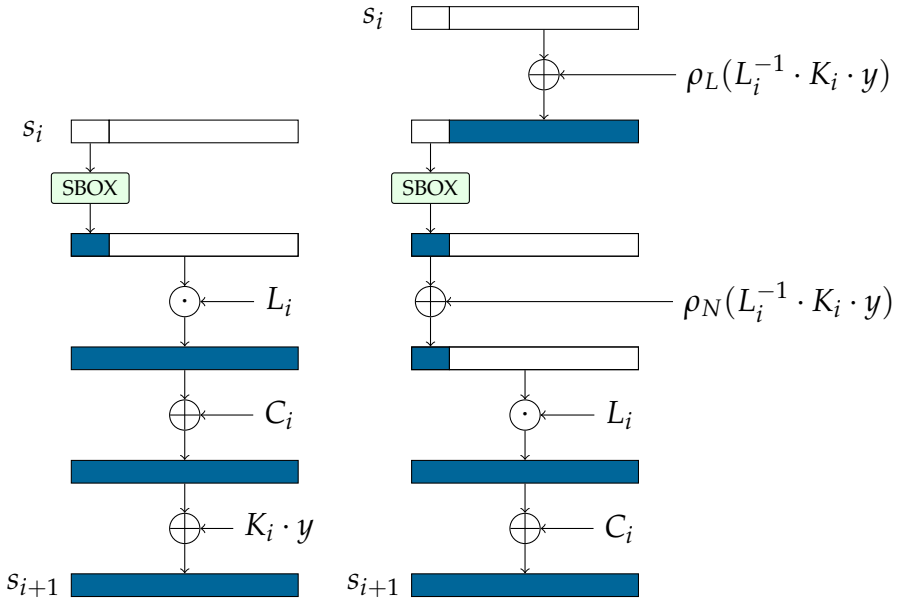


Figure 3.1.: One round of LowMC before (left) and after (right) the splitting of the round key [Per+17]

linear part includes the calculation of the linear parts of the round keys of all rounds if they are moved to the beginning of the encryption. The combination of the linear round key parts works as follows. For the first round, the round key can be calculated as $\overline{L_1^{-1}} \cdot K_1 \cdot y$. Starting from the second round, we also have to consider all rounds up to the currently evaluated round in our computation because we move the round key past the linear layer of each round. Therefore, the linear round key part for the second round is calculated as $(\overline{L_1^{-1}} \cdot K_1 + \overline{L_1^{-1}} \cdot \overline{L_2^{-1}} \cdot K_2) \cdot y$. If all round key parts have been moved to the beginning of the encryption, we can add the matrices for all round keys without considering y yet. This combined matrix P_L can according to [Per+17] be described as

$$P_L = \overline{L_1^{-1}} \cdot K_1 + \sum_{j=2}^{r} \left(\prod_{k=1}^{j} \overline{L_k^{-1}} \right) \cdot K_j.$$

P_L only involves the matrices K_i and $\overline{L_i^{-1}}$, which are based on L_i, which allows us to precompute it once as soon as the LowMC instance is fixed. Hence,

before an encryption starts, P_L is multiplied by the secret key y to get the linear round key part for the specific shares of y and this specific encryption. $P_L \cdot y$ is then added to the initial state s_0 which means that the linear part of all round keys is added at once.

Now we have to adapt the non-linear part of the round key accordingly because moving the linear round key part past the linear layer gives us some calculation artifacts. If $\overline{L_2^{-1}} \cdot K_2 \cdot y$ is moved past the linear layer of round 1 we multiply it by $\overline{L_1^{-1}}$. This structural change also influences the non-linear part of round 1 because the result of the multiplication by $\overline{L_1^{-1}}$ also has bits with value one in the lower $3m$ bits (the non-linear part) of the state. Therefore, in each round j we receive all artifacts from rounds $j+1$ to r as all of these rounds are moved past round j. This results in the matrix P_{N_i} for round i which can be calculated as

$$P_{Ni} = \overline{L_i^{-1}} \cdot K_i + \sum_{j=i+1}^{r} \left(\prod_{k=i}^{j} \overline{L_k^{-1}} \right) \cdot K_j.$$

These matrices P_{N_i} are multiplied by y in each round. The first $3m$ bits of the resulting vector are added to the non-linear part of the state after the S-box in each round.

Now we still have a correct encryption algorithm, but the linear part of the round key addition of each round was moved to the beginning of the encryption. The round key addition in each round was reduced to the non-linear part. However, we still did not remove the multiplication by the secret key from each LowMC round. This removal of the multiplication can be done as follows. We only use the first $3m$ bits of $P_{N_i} \cdot y$ which means that we also only need the first $3m$ rows of P_{N_i}. In the following, these described three rows of P_{N_i} are denoted as $\overline{P_{Ni}}^{3m}$. The matrices $\overline{P_{Ni}}^{3m}$ of all rounds can be combined to one matrix C_N which has the dimensions $(3m \cdot r \times k)$, where m is the number of S-boxes, k the key size, and r the number of LowMC rounds.

$$C_N = \underbrace{\begin{pmatrix} \overline{P_{N1}}^{3m} \\ \vdots \\ \overline{P_{Nr}}^{3m} \end{pmatrix}}_{k \text{ cols}} \begin{matrix} \} 3m \text{ rows} \\ \\ \} 3m \text{ rows} \end{matrix}$$

This matrix C_N can, like P_N, be precomputed as soon as the LowMC instance is fixed because it also only depends on all K_i and $\overline{L_i^{-1}}$ matrices. In the same manner as for the linear part, we calculate $v = C_N \cdot y$ at the beginning of an encryption to get the correct non-linear part for the current shares of y. Now we can add the bits $\rho^{3m \cdot i}_{1+3m \cdot (i-1)}(v)$ to the non-linear part of the state after the S-box in round i.

Algorithm 2 shows the full LowMC encryption with the split round key computation. The major differences to Algorithm 1 include the multiplications of the precomputed values by the secret key in lines 1 and 2 and the addition of the non-linear part of the state in line 5.

Algorithm 2 LowMC encryption with the split round key computation for key matrices $K_i \in \mathbb{F}_2^{n \times k}$ for $i \in [0, r]$, linear layer matrices $L_i \in \mathbb{F}_2^{n \times n}$ and round constants $C_i \in \mathbb{F}_2^n$ for $i \in [1, r]$, and the precomputed matrices P_L and C_N. [Per+17]

Require: plaintext $p \in \mathbb{F}_2^n$ and key $y \in \mathbb{F}_2^k$

1: $v \leftarrow C_N \cdot y$
2: $s \leftarrow (K_0 + P_L) \cdot y + p$
3: **for** $i \in [1, r]$ **do**
4: $\quad s \leftarrow \mathrm{SBOX}(s)$
5: $\quad s \leftarrow \rho^{3m \cdot i}_{1+3m \cdot (i-1)}(v) + s$
6: $\quad s \leftarrow L_i \cdot s$
7: $\quad s \leftarrow C_i + s$
8: **end for**
9: **return** s

We now successfully removed the multiplication $K_i \cdot y$ from each LowMC round. Instead of $r(n \times k) \cdot (k \times 1)$ multiplications we only have $1(n \times k) \cdot (k \times 1)$ multiplication for the linear part and $1(3m \cdot r \times k) \cdot (k \times 1)$ multiplication for the non-linear part. Additional to the two multiplications we use a $3m$-bit addition in each round and one n-bit addition at the beginning of the encryption. The original algorithm required an n-bit addition in each of the r rounds, so we did not reduce the number of additions, but we decreased their size immensely.

Also, the memory requirements are reduced drastically because the round key matrices are not required during the encryption anymore, and we do not

	LowMC	**LowMC with RRK**
Linear layer	r $(n \times n)$	r $(n \times n)$
Round key matrices	$(r+1)$ $(n \times k)$	1 $(n \times k)$ (linear part)
		1 $(3m \cdot r \times k)$ (non linear part)
Round constants	r $(1 \times n)$	r $(1 \times n)$
Additional memory		1 $(3m \cdot r \times 1)$ (vector v)

Table 3.1.: Necessary matrices for general LowMC and LowMC with RRK. [Per+17]

have to store them. Table 3.1 compares the required matrices for the original LowMC algorithm and our modified version with the reduced round key computation (RRK). It shows that the memory requirements for the modified algorithm depend on the number of S-boxes m. The original algorithm always uses the same amount of memory independent of m.

The precomputation of P_L and C_N is only useful if the instance of LowMC is fixed. If the instance were not fixed, the precomputation would have to be done for each encryption freshly. This would undo all the benefits we gained from the improved round key addition in each round. The precomputation involves many matrix multiplications and is therefore very expensive. However, fixed instances allow us to compute P_L and C_N once and use them for many encryptions. It is not a problem if different secret keys are used for different executions of the encryption with the same instance because P_L and C_N do not use the secret key during the precomputation. As shown above the secret key and its shares are only included at the beginning of each encryption.

Correctness. Now we briefly broach the correctness of the proposed optimization. First, we show that all transformations applied to one round of LowMC are equivalent transformations. Second, we discuss the correctness of passing the linear part of the round key across several rounds. We continue to use the notation $\overline{L_i^{-1}}$ for the inverse of L_i whose first $3m$ rows are set to zero. For a vector v, v_N denotes a vector with the first $3m$ bits of v followed by $n-3m$ zeros, and v_L denotes $3m$ zeros followed by the $n-3m$ last bits of v. For simplicity, k_i denotes the round key of round i, *i.e.*, $k_i = K_i \cdot y$.

The equivalent transformations on one round are shown in Equation 3.1. The

original LowMC encryption is shown in Equation 3.1a. Equation 3.1b shows the splitting of the input to the S-box. Only the non-linear part of the state has to be the input to the S-box. The linear part uses an identity mapping and is therefore not influenced by the S-boxes. In Equation 3.1c, the round key $K_i \cdot y$ is split into a non-linear and linear part.

$$\begin{aligned} s_i &= L_i \cdot \mathrm{SBOX}(s_{i-1}) + K_i \cdot y + C_i && \text{(3.1a)} \\ &= L_{iL} \cdot s_{i-1_L} + L_{iN} \cdot \mathrm{SBOX}(s_{i-1_N}) + K_i \cdot y + C_i && \text{(3.1b)} \\ &= L_{iL} \cdot s_{i-1_L} + (K_i \cdot y)_L + L_{iN} \cdot \mathrm{SBOX}(s_{i-1_N}) + (K_i \cdot y)_N + C_i && \text{(3.1c)} \end{aligned}$$

Up to now, we showed that the transformations which are applied to one LowMC round are correct. As the next step, we show that moving the linear part of the round key to the beginning of the encryption is also correct. To show this correctness we look at a general case where the linear part of the rounds $i-1$ and i is moved to the "initial state" s_{i-2}. For simplicity reasons we omit the round constant in this procedure and note that this does not influence the correctness of the transformations. Equation 3.2 shows the passing of the linear part of the round key across two rounds. The S-box layer is denoted as S. First, in Equation 3.2b the linear layer and the round key layer of both LowMC rounds are exchanged. In order to do so, the round key has to be added before the linear layer is applied and multiplied by the inverse linear layer of the corresponding round. The next step in Equation 3.2c splits the round key of round $i-1$ into a linear and non-linear part. Furthermore, the linear part $(L_{i-1} \cdot k_{i-1})_L$ is moved into the S-box function, *i.e.,* before the S-box layer. Now we have a look on the round key of round i. In Equation 3.2d, the round key of round i is split into a linear and non-linear part. Equation 3.2e moves the linear part $(L_i \cdot k_i)_L$ across the S-box layer of round i. This step does not require any further adaptions except for reordering the two calculations. In Equation 3.2f, the linear round key part $(L_i \cdot k_i)_L$ is moved across the linear layer of round $i-1$. To reorder the calculations we have to multiply $(L_i \cdot k_i)_L$ by $\overline{L_{i-1}^{-1}}$. However, this calculation generates artifacts for the non-linear part. Therefore, we have to add the term $(\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot k_i)_N$ to correct the non-linear part. In Equation 3.2g, the term $(\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot k_i)_L$ is moved across the S-box layer of round $i-1$. Equation 3.2h simply changes the notation of the round

key back to the original notation for a better comparison with the previously introduced formulas for P_N and P_{Ni}. In Equation 3.2i, y is factored out.

The formula we use for precalculating the linear part of all round keys at the beginning of the encryption is

$$P_L = \overline{L_1^{-1}} \cdot K_1 + \sum_{j=2}^{r} \left(\prod_{k=1}^{j} \overline{L_k^{-1}} \right) \cdot K_j.$$

If we compare this formula to our two round transformation in Equation 3.2 we can observe the following. The term $(\overline{L_{i-1}^{-1}} \cdot K_{i-1})_L + (\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot K_i)_L$ has to be represented by the formula. If we insert $r = 2$ into the formula we can observe that $(\overline{L_{i-1}^{-1}} \cdot K_{i-1})_L$ is the same as the term $(\overline{L_1^{-1}} \cdot K_1)_L$ in the formula. The second part of the formula, $\sum_{j=2}^{r} \left(\prod_{k=1}^{j} \overline{L_k^{-1}} \right) \cdot K_j$, has to generate the term $(\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot K_i)_L$. If we again insert $r = 2$ we exactly observe the desired result. For the artefacts of the non-linear part, which are calculated with C_N, the same observations hold.

To sum it up, we showed that passing the linear part of the round key across two rounds only involves equivalent transformations, *i.e.,* Equation 3.2a and Equation 3.2g are equivalent.

3.1.1. Implementation

We now describe a few remarks for the implementation of the proposed optimization. The basis for our implementation of the precomputation of P_L and C_N is the LowMC reference on GitHub[1]. The whole modified algorithm is implemented in the Picnic implementation of the Institute of Applied Information Processing and Communications (IAIK)[2]. We implement the precomputation in Python and use the matrix operations available in Sage. The full code for the precomputation can be found on Github [3].

[1] `https://github.com/LowMC/lowmc`
[2] `https://github.com/IAIK/Picnic`
[3] `https://github.com/IAIK/Picnic-LowMC`

$$
\begin{aligned}
s_i &= L_i \cdot \mathrm{S}(L_{i-1} \cdot \mathrm{S}(s_{i-2}) + k_{i-1}) + k_i && \text{(3.2a)}\\
&= L_i \left(\mathrm{S}\left(L_{i-1} \left(\mathrm{S}(s_{i-2}) + \overline{L_{i-1}^{-1}} \cdot k_{i-1} \right) \right) + \overline{L_i^{-1}} \cdot k_i \right) && \text{(3.2b)}\\
&= L_i \left(\mathrm{S}\left(L_{i-1} \left(\mathrm{S}(s_{i-2} + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_L) + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_N \right) \right) + \overline{L_i^{-1}} \cdot k_i \right) && \text{(3.2c)}\\
&= L_i \left(L_{i-1} \left(\mathrm{S}(s_{i-2} + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_L) + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_N \right) \right) + (\overline{L_i^{-1}} \cdot k_i)_L + (\overline{L_i^{-1}} \cdot k_i)_N \Big) && \text{(3.2d)}\\
&= L_i \left(\mathrm{S}\left(L_{i-1} \left(\mathrm{S}(s_{i-2} + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_L) + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_N \right) + (\overline{L_i^{-1}} \cdot k_i)_L \right) + (\overline{L_i^{-1}} \cdot k_i)_N \right) && \text{(3.2e)}\\
&= L_i \left(\mathrm{S}\left(L_{i-1} \left(\mathrm{S}(s_{i-2} + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_L) + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_N + (\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot k_i)_L \right) + (\overline{L_i^{-1}} \cdot k_i)_N \right) \right) + (\overline{L_i^{-1}} \cdot k_i)_N \Big) && \text{(3.2f)}\\
&= L_i \left(L_{i-1} \left(\mathrm{S}(s_{i-2} + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_L + (\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot k_i)_L) + (\overline{L_{i-1}^{-1}} \cdot k_{i-1})_N + (\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot k_i)_N \right) \right) + (\overline{L_i^{-1}} \cdot k_i)_N \Big) && \text{(3.2g)}\\
&= L_i \left(\mathrm{S}\left(L_{i-2} + (\overline{L_{i-1}^{-1}} \cdot K_{i-1} \cdot y)_L + (\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot K_i \cdot y)_L) + (\overline{L_{i-1}^{-1}} \cdot K_{i-1} \cdot y)_N + (\overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot K_i \cdot y)_N \right) \right) + (\overline{L_i^{-1}} \cdot K_i \cdot y)_N \Big) && \text{(3.2h)}\\
&= L_i \left(L_{i-1} \left(\mathrm{S}(s_{i-2} + ((\overline{L_{i-1}^{-1}} \cdot K_{i-1} + \overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot K_i) \cdot y)_L) + ((\overline{L_{i-1}^{-1}} \cdot K_{i-1} + \overline{L_{i-1}^{-1}} \cdot \overline{L_i^{-1}} \cdot K_i) \cdot y)_N \right) \right) + (\overline{L_i^{-1}} \cdot K_i \cdot y)_N \Big) && \text{(3.2i)}
\end{aligned}
$$

All matrices for the whole LowMC algorithm are stored as their transposed representation. This eases, for example, the multiplication of a matrix M by a vector v. We denote the transposition of M as N, *i.e.,* $M^T = N$. The only required mathematical operation is summing up the necessary rows of the matrix N. If bit i is set in vector v, row i of matrix N is added to the result vector. In the following equation we show that this implementation of the matrix multiplication is correct and favorable from an implementation point of view. M_i denotes row i of matrix M, M^j denotes column j of matrix M. $< v_1, v_2 >$ denotes the scalar product of the two vectors v_1, v_2.

$$M \cdot v = \begin{pmatrix} < M_1, v > \\ \vdots \\ < M_n, v > \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{n} (M_1^k \cdot v_k) \\ \vdots \\ \sum_{k=1}^{n} (M_n^k \cdot v_k) \end{pmatrix} = \sum_{k=1}^{n} \begin{pmatrix} M_1^k \cdot v_k \\ \vdots \\ M_n^k \cdot v_k \end{pmatrix} \tag{3.3}$$

$$= \sum_{k=1}^{n} (M^k \cdot v_k) = \sum_{k=1}^{n} ((M^T)_k \cdot v_k) = \sum_{k=1}^{n} (N_k \cdot v_k) \tag{3.4}$$

Equation 3.3 rewrites the multiplication first to a vector of scalar products. Second, the scalar product is rewritten to a sum. In the third transformation the sum is not calculated in each element of the result vector, but the sum is taken over the vectors $(M_1^1 \cdot v_1, \cdots, M_n^1 \cdot v_1), \cdots, (M_1^n \cdot v_n, \cdots, M_n^n \cdot v_n)$. The vector $(M_1^k \cdot v_k, \cdots, M_n^k \cdot v_k)$ can be rewritten to $M^k \cdot v_k$, which is shown in Equation 3.4. In other words, each column i of the matrix M is multiplied by element i of vector v. All of these resulting vectors are summed up to the result of the multiplication $M \cdot v$. To facilitate the implementation of this multiplication variant we store the transposed variant of M, which we denote as N. We can now multiply each row i of matrix N by element i of vector v to calculate $M \cdot v$.

In general, the number of S-boxes can be chosen freely. However, as Table 2.2 shows, we propose to always use 10 S-boxes for the LowMC instantiation in Picnic. When C_N is assembled in the Python script, we add 2 columns of padding after the columns for each round. Note that we use columns here instead of rows as described in the previous paragraph because we calculate with the transposed versions of all matrices. Once C_N is multiplied with the secret key y we get a vector of length $(3 \cdot 10 + 2) \cdot r$. This makes extracting the non-linear part of one round much easier because modern operating systems use 64-bit datatypes. In our implementation, we use `uint64_t` (unsigned 64-bit

integers) arrays to store the data of our vectors and matrices. Therefore, we can simply extract the upper or lower half of one of these integers to get the non-linear part of one round. The two bits of padding are always the last two bits of both halves of the integer. This makes it easy to use bit masks to get the necessary 30 bits. Only storing the necessary 30 columns for each round makes it much more difficult to extract the required $3m$ bits out of the vector v. The bits for the non-linear part of one round would in many cases be spread across two integers. More complex bit operations than a simple bit mask would be necessary to connect the necessary bits.

3.2. Replacing L_i with a Feistel Network

The most expensive operation which remains in each LowMC round after the previous optimization has been applied is the linear layer. The linear layer in each round consists of one $(n \times n) \cdot (n \times 1)$ matrix multiplication, where n is the block size. Therefore, we propose to replace the matrix multiplication with a Feistel network [Per+17]. The main idea is to replace L_i by several smaller matrices which are used in the round functions of the Feistel network. More specifically we propose to use a Fibonacci-Feistel network (Ffn) which is based on Generalized Feistel networks (Section 2.5). The Ffn is based on the Fibonacci sequence, which is defined as $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2} \quad \forall n \geq 2$. We introduce the notation $i = \Lambda_f(b)$ where i is the smallest integer such that $f_i > b$, *e.g.,* $f_5 = 5, f_6 = 8$, so $\Lambda_f(7) = 6$.

The Ffn works as follows. The parameter that can be chosen for the Ffn is the branch size $w \geq 4$. The constraints it has to fulfill are that

- $w \leq \frac{n}{4}$ because at least 4 branches are required for the Feistel network, and
- $n \bmod w = 0$ because we have to split the whole input into branches and cannot leave some bits untouched.

The input s of size n is split into $2b$ branches of size w such that $2b \cdot w = n$. Hence, the number of branches $2b$ can be calculated as $\frac{n}{w}$ and b (*i.e.,* the number of branches on one half of the input) is always greater or equal to 2.

s_i $(0 \leq i \leq 2b-1)$ denotes the i-th branch of the input, *i.e.,* bits $i \cdot w$ to $(i+1) \cdot w - 1$. Figure 3.2 shows the splitting of the input into the branches.

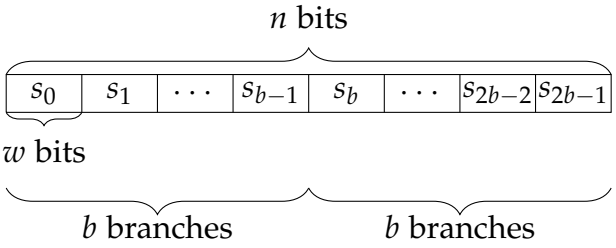


Figure 3.2.: Splitting the n-bit input s into $2b$ branches, each branch s_i consisting of w bits.

The number of necessary rounds r in the Ffn can be calculated as $\Lambda_f(b)$. Furthermore, we need $r \cdot b(w \times w)$ matrices which are used as a linear mapping in the Ffn. All of these matrices have to be invertible. We denote the $(w \times w)$ matrix which is applied to branch i in round j as M_i^j.

In each round $i \in [0, r-1]$ of the Ffn the first b branches are multiplied with the corresponding $(w \times w)$ matrix M_i^j. After these multiplications, all branches are rotated by the Fibonacci number f_i, *i.e.,* $B_k \leftarrow B_{(k+f_i) \bmod b}$. According to the Fibonacci sequence, the branches are not rotated in the first round. For the next few rounds, the rotations in the Feistel network are as follows. In the second and third round the branches are rotated by 1, in the fourth round by 2, and in the fifth by 3.

The whole Feistel network is shown in Algorithm 3. Line 3 shows the application of the matrices M_i^j on the state. In line 6 the state is rotated. In the end, in lines 9 and 10 the two halves of the state are swapped.

Figure 3.3 shows the first two rounds of a Ffn with 8 branches. The round functions contain the matrices M_i^j. The Fibonacci permutation can first be seen in round two when the result of $F_{2,1}$ is not applied to y'_5, but to y'_6. In round 3, $F_{3,1}$ is again applied to the sixth branch, whereas in round 4 ($f_4 = 2$) $F_{4,1}$ is applied to the seventh branch.

To replace the multiplications by L_i with a Ffn we need to represent each matrix L_i as $r \cdot b(w \times w)$ matrices M_i^j which can be used in the Ffn. This

Algorithm 3 FFN for input $s \in \mathbb{F}_2^n$, temporary storage $tmp \in \mathbb{F}_2^n$, matrices $M_i^j \in \mathbb{F}_2^{w \times w}$ for $i \in [0, b-1]$ and $j \in [0, r-1]$, and f_i for $i \in [0, r-1]$ being the Fibonacci sequence described above.

for $j \in [0, r-1]$ **do**
  for $i \in [0, b-1]$ **do**
    $tmp_i = M_i^j \cdot s_i$ ▷ store the state temporarily for the rotation
  end for
  for $i \in [0, b-1]$ **do**
    $s_{b+((i+f_j) \bmod b)} += tmp_i$
  end for
  $tmp = s$ ▷ store the state temporarily for swapping the two halves
  $s_{0,1,\cdots,b-2,b-1} = tmp_{b,b+1,\cdots,2b-2,2b-1}$
  $s_{b,b+1,\cdots,2b-2,2b-1} = tmp_{0,1,\cdots,b-2,b-1}$
end for
return s

representation cannot be computed easily. To be more specific, it is not always possible to calculate the matrices M_i^j of an already known matrix L_i. Therefore, we have to generate the matrices M_i^j first. These matrices can then be used to calculate the corresponding $(n \times n)$ matrix which is represented by the small matrices M_i^j. Computing the $(n \times n)$ matrix out of the matrices M_i^j can be done as follows. First, $r \cdot b$ random $(w \times w)$ matrices M_i^j are generated. We need the identity matrix of size $(n \times n)$ as the basis for the calculation of the $(n \times n)$ matrix L_i. The FFN, with the generated matrices M_i^j as round functions, is applied to each row (= column) of the identity matrix, *i.e.,* the FFN is executed n times, each time with one row of the identity matrix as its input. The result of the FFN with input row i is then stored in column i of the output matrix L_i. After all rows of the identity matrix have been processed, we obtained the matrix L_i. We note that it is necessary to transpose this calculated representation of L_i to be compliant with the implementation we described in Section 3.1.1.

The permutations which are used in the proposed FFN allow a very fast diffusion. This approach was already suggested in 2010 by Suzaki and Minematsu [SM10]. Their proposal requires $2 \cdot \log_2(b)$ rounds for a full diffusion operating on $2b$ branches. Our proposal uses $\Lambda_f(b)$ rounds for $2b$ branches,

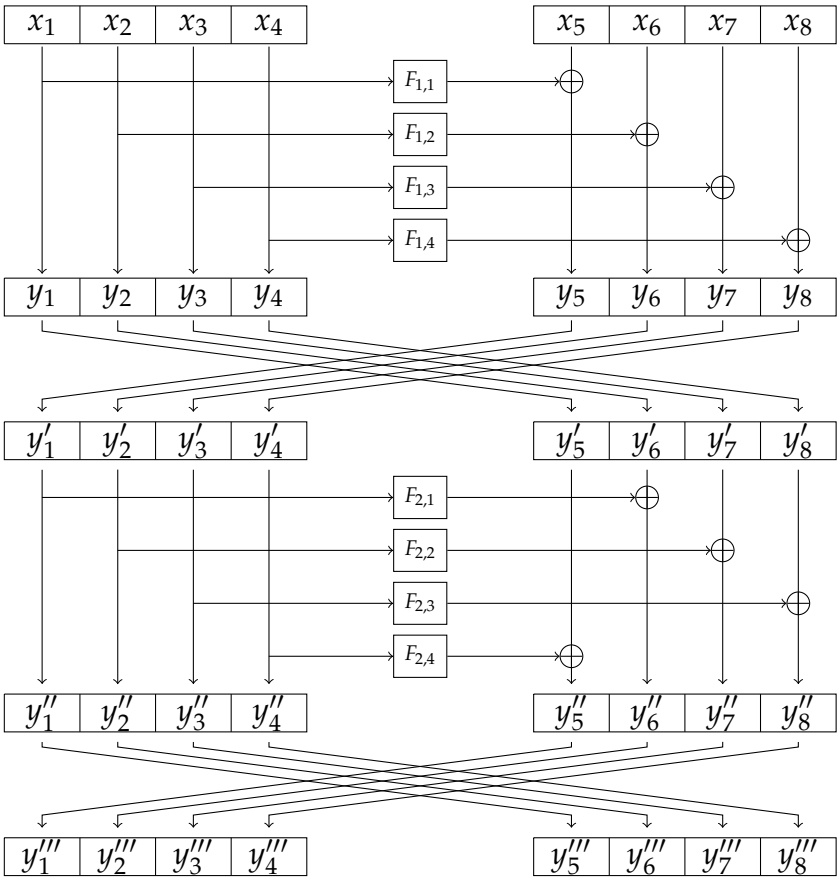


Figure 3.3.: Two round of the proposed Feistel network. In the first round the branches are not permuted ($f_0 = 0$), in the second round the branches are permuted by one ($f_1 = 1$). In the round function $F_{j,i}$ the matrix M_i^j is applied to the state.

which also provides a full diffusion [Per+17].

3.2.1. Implementation

This section focuses on the implementation details of the Ffn explained in the previous section. We implemented the Ffn using C code in the setting of

the PICNIC implementation already described in Section 3.1.1. Furthermore, a Python implementation of the FFN is available in Appendix A.

One massive bottleneck in the implementation of the FFN as it is described in Algorithm 3 is the rotation and the swapping of the branches. It would not be very efficient to copy all branches to their destination in each round of the FFN. Therefore, we propose a variant of the algorithm which does not copy any branches but solely relies on the indexing of the branches. Algorithm 4 shows the improved algorithm. The main difference is that the two halves of the state are never actually swapped in memory. In general, after swapping the two halves, the bits that were in branch s_0 in round 0 are found in branch s_n in round 1. In our implementation the representation in memory stays the same and we have to address s_0 to get the bits of s_n in round 1. In round 2, we can address the bits that were in s_0 in round 0 and in s_n in round 1 again as s_0 because the two halves were "swapped" again. Every second round requires this change in indexing. To facilitate these index changes in the implementation, we introduce two macros $perm_L$ and $perm_R$. Their parameter is the current round. Depending on the round it returns 0 or b, 0 meaning that we can use the "normal" index of the branch and b meaning that we have to adapt the index. $perm_L$ and $perm_R$ work exactly in the opposite way. This is necessary because we use $perm_R$ for reading the branch that is multiplied by M_i^j and $perm_L$ for adding the result of the multiplication to the correct branch. The two branches which are involved in this operation will never be in the same half of the state. Therefore, $perm_L$ and $perm_R$ are used to address the correct half of the state. $perm_L$ is always used on the left side of the calculation and $perm_R$ on the right side. The rest of the two indices for the branches is just used for indexing the correct branch within the corresponding half and is not modified. These unmodified indices are $(f_j + i) \bmod b$ for the branch that is modified, and b for the branch that is used for the multiplication.

The statement $s_{perm_L(j)+((f_j+i) \bmod b)} += M_i^j \cdot s_{perm_R(j)+b}$ in Algorithm 4 now combines all elements of the FFN:

- the multiplication by the matrix M_i^j on the right-hand side,
- the rotation using f_j in the index of the branch on the left-hand side,
- the swapping of the two halves of the state using $perm_L$ and $perm_R$, and

- combining the round function (the multiplication by M_i^j) with the correct branch using the addition of the left and right-hand side.

As we never actually swap the two halves in memory, we produce the wrong result if the number of rounds in the Feistel network is odd. Therefore, after applying all rounds on the initial state, the two halves need to be swapped in memory.

Algorithm 4 Ffn for input $s \in \mathbb{F}_2^n$, matrices $M_i^j \in \mathbb{F}_2^{w \times w}$ for $i \in [0, b-1]$ and $j \in [0, r-1]$, and f_i for $i \in [0, r-1]$ being the Fibonacci sequence described above.

def $perm_L(a) : a \bmod 2 \ ? \ 0 \ : \ b$
def $perm_R(a) : a \bmod 2 \ ? \ b \ : \ 0$

for $j \in [0, r-1]$ **do**
  for $i \in [0, b-1]$ **do**
    $s_{perm_L(j)+((f_j+i) \bmod b)} += M_i^j \cdot s_{perm_R(j)+b}$
  end for
end for
if $r \bmod 2 = 1$ **then** ▷ Swap halves in memory if r is odd
  $tmp = s$
  $s_{0,1,\cdots,b-2,b-1} = tmp_{b,b+1,\cdots,2b-2,2b-1}$
  $s_{b,b+1,\cdots,2b-2,2b-1} = tmp_{0,1,\cdots,b-2,b-1}$
end if
return s

A crucial point in the implementation of this algorithm is the multiplication. There are $r \cdot b$ $(w \times w) \cdot (w \times 1)$ multiplications in one execution of the Ffn. An easy constant-time approach implemented in C for $w = 4$ is shown in Listing 3.1. This implementation uses the following representation of the data. v is the w-bit vector which is multiplied by the $(w \times w)$ matrix M. The result of the multiplication is stored in c. v and c are 64-bit integers. M is an array of 4 64-bit integers, where each integer stores one row of the matrix. We need to emphasize here again that we store the transposed variants of all matrices. Therefore the bits in the first integer actually represent the first column of M. The vector v is connected via the bitwise AND operator with the numbers $1, 2, 4$, and 8. This is used to check which bits of v are set, *e.g.,* $(v\&1)$ returns

$0x1$ if bit 0 of v is set, (v&2) returns $0x2$ if bit 1 of v is set, etc.. We will now show the effect of the double negation operator in lines 2-4 on the example of (v&2). As mentioned above, (v&2) returns $0x2$ if bit 1 of v is set. Negating $0x2$ results in $0x0$ because all values not equal to zero evaluate to 0 if they are negated in C. The second negation changes $0x0$ to $0x1$ because the negation of zero is one. The second case, (v&2) returns $0x0$ because bit 1 of v is not set, works likewise. The first negation changes $0x0$ to $0x1$ and the second negation changes it back to $0x0$. This procedure is necessary to get either $0x0$ or $0x1$ as an indicator if a bit of v is set. The resulting $0x0$ or $0x1$ can now be multiplied by the corresponding row of M. The result of the multiplication is either a vector with zeros, or row i of M if bit i was set. Due to the transposed representation we can simply apply an XOR on these 4 multiplication results and get the result c for the $(4 \times 4) \cdot (4 \times 1)$ multiplication.

```
c   = M[0] * (v & 1);
c ^= M[1] * !!(v & 2);
c ^= M[2] * !!(v & 4);
c ^= M[3] * !!(v & 8);
```

Listing 3.1: A constant-time implementation in C for the multiplication $c = v \cdot M$ for c and $v \in \mathbb{F}_2^4$ and $M \in F_2^{4\times4}$.

The data structure which is used for storing matrices in the implementation of Picnic requires a change to Listing 3.1. Each row consists of at least 2 uint64_t, which means that each row is at least 128 bits wide. Consequently, Picnic uses 8 64-bit integers to store a (4×4) matrix. The remaining bits of the first integer and the second integer of each row are used as padding. The number of integers which is used per row is called *rowstride*. The value of *rowstride* is 2 in the case of a (4×4) matrix. We need to use $M[1 \cdot rowstride]$, $M[2 \cdot rowstride]$, and $M[3 \cdot rowstride]$ to address the second, third, and fourth row of M, respectively.

The suggested constant-time implementation can be adapted easily for branch sizes up to 64 bits. The only necessary adaption is that not only the first 4 but all 64 rows of M and all 64 bits of v are considered. If the branch size exceeds 64 bits, one row of M does not fit into one uint64_t anymore. Then we have to consider both integers of one row in the calculation because the second integer is not only padding anymore.

3.3. NEON Instruction Set Extension

Intel CPUs offer SIMD instruction set extensions which are used in the implementation of Picnic. Picnic uses features of SSE2, SSE4.1, and AVX2. The SIMD instructions are mainly used to implement matrix operations like a matrix multiplication. We can exploit the fact that SSE and AVX operate on different register sizes to optimize the performance of Picnic. This can be done by implementing optimized versions of several functions for different block sizes in different instruction set extensions. A 256-bit multiplication is faster with AVX than with SSE because the 256-bit values can be stored in one register. AVX does not provide an additional speedup compared to SSE for a 128-bit matrix multiplication because the register size of SSE suffices for the calculation.

Picnic can be compiled for ARM architectures when the general purpose C implementation is used and SSE and AVX code is disabled. We implement the SSE and AVX based code in NEON to optimize the performance on ARM CPUs because most modern smartphones, tablets, and embedded systems use ARM CPUs[4]. The size of the NEON registers is 128 bits which is the same size as of SSE registers. Therefore, we base most of the NEON code on the SSE implementation and not on the AVX version, which uses 256-bit registers. The 128-bit datatype of SSE is called __m128i. Depending on the function that is applied to a __m128i, it is interpreted as multiple 8-, 16-, 32-, or 64-bit values. NEON offers various 128-bit data types, which already specify what they represent. Examples for such 128-bit data types include uint8x16_t (16 8-bit unsigned integers), uint16x8_t (8 16-bit unsigned integers), uint32x4_t (4 32-bit unsigned integers), and uint64x2_t (2 64-bit unsigned integers). All of these data types are also available as a version with signed integers. In Picnic, uint32x4_t is used in the NEON code. However, any other NEON data type would be possible as well because we do not use the intended representation of the data type.

Parts of the code which are optimized with SSE intrinsics are also straightforward to implement with NEON instructions. This is the case if NEON offers an equivalent intrinsic to the used SSE intrinsic. The equivalent functions of SSE and NEON which are used in Picnic are shown in Table 3.2.

[4] https://www.arm.com/

Description	**SSE intrinsic**	**NEON intrinsic**
128-bit XOR	_mm_xor_si128(a, b)	veorq_u32(a, b)
128-bit OR	_mm_or_si128(a, b)	vorrq_u32(a, b)
128-bit AND	_mm_and_si128(a, b)	vandq_u32(a, b)

Table 3.2.: Equivalent functions in the SSE and NEON instruction set.

```
__m128i* mzd_mul_sse_128(__m128i* c, __m128i* v,
                         __m128i* A) {
  uint64_t* vptr  = (uint64_t*) v;
  uint64_t v1 = *vptr, v2 = *(vptr+1);
  *c = (v1 & 1) * A[0];
  for(int i = 1; i < 64; i++) {
    *c =  _mm_xor_si128(*c,
            ((!!(v1 & ((uint64_t)1 << i)))*A[i]));
  }
  for(int i = 0; i < 64; i++) {
    *c =  _mm_xor_si128(*c,
            ((!!(v2 & ((uint64_t)1 << i)))*A[i+64]));
  }
  return c;
}
```

Listing 3.2: Constant time matrix multiplication implemented with SSE intrinsics.

An example of a function that is converted from SSE to NEON is shown in Listing 3.2 and Listing 3.3. Listing 3.2 shows the matrix multiplication $c = A^T \cdot v$ using SSE intrinsics. In Listing 3.3 the same multiplication is done with NEON intrinsics. The only difference between the two implementations is the used data type and XOR function.

One SSE intrinsic that does not directly map to a NEON intrinsic with the same functionality is _mm_setzero_si128(). It sets a 128-bit value to 0. In NEON there is the intrinsic vmovq_n_u32(uint32_t a). It sets the 4 32-bit integers in the 128-bit value to the integer a. Therefore, executing the NEON instruction vmovq_n_u32(0) has the same effect as the SSE intrinsic _mm_setzero_si128().

```
uint32x4_t* mzd_mul_neon_128(uint32x4_t* c,
                             uint32x4_t* v,
                             uint32x4_t* A) {
  uint64_t *vptr  = (uint64_t*) v;
  uint64_t v1 = *vptr, v2 = *(vptr+1);
  *c = (v1 & 1) * A[0];
  for(int i = 1; i < 64; i++) {
    *c =  veorq_u32(*c,
            ((!!(v1 & ((uint64_t)1 << i)))*A[i]));
  }
  for(int i = 0; i < 64; i++) {
    *c =  veorq_u32(*c,
            ((!!(v2 & ((uint64_t)1 << i)))*A[i+64]));
  }
  return c;
}
```

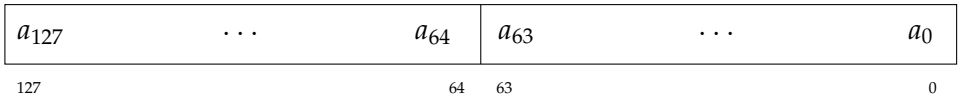
Listing 3.3: Constant time matrix multiplication implemented with NEON intrinsics.

In contrast to the examples presented above, there are also SSE intrinsics which do not have an equivalent NEON intrinsic. It is therefore not straightforward to implement functions, which use these SSE intrinsics, using NEON intrinsics. In these cases, it is necessary to find an alternative implementation of the function using NEON intrinsics. One example for such an alternative implementation is the shifting of a 128-bit value. Neither SSE nor NEON offers an intrinsics which shifts the 128-bit value as one value. However, for the bit-sliced SBOX implementation of LowMC, we require a shift function that interprets all 128-bits as one value. The bit-sliced SBOX implementation uses bitmasks to create three instances of the S-box input, s_1, s_2 and s_3, where one instance stores only the first, second, or third input bit of each S-box. In other words, s_1 stores the first input bit of each S-box and keeps them at their correct place in the state, hence $s_1 = [x_0, 0, 0, x_3, 0, 0, x_6, \cdots]$, $s_2 = [0, x_1, 0, 0, x_4, 0, 0, x_7, \cdots]$, and $s_3 = [0, 0, x_2, 0, 0, x_5, 0, \cdots]$. The SBOX function uses the AND and XOR operation on the three input bits of the S-box. To facilitate the implementation of these operations, we shift s_2 by 1 bit and s_3 by 2 bits to the left. After these shifts all input bits for the S-boxes in s_1, s_2 and s_3 have been moved to the indices $j \bmod 3 = 0$, *i.e.,* , $s_1 = [x_0, 0, 0, x_3, 0, 0, x_6, \cdots]$, $s_2 = [x_1, 0, 0, x_4, 0, 0, x_7, \cdots]$, and $s_3 = [x_2, 0, 0, x_5, 0, 0, x_8 \cdots]$. The resulting structure can be used to easily apply AND and XOR on two input bits of the S-box, *e.g.,* by calculating $s_1 \wedge s_2$ we execute an AND with the first and second bit of each S-box as input. For this implementation to be correct, we need to shift the whole state as one 128-bit value. Therefore, we need a 128-bit shift which does not treat the 128-bit register as 32- or 64-bit values. We want to emphasize here that using the 128 bits as one value is not the intended use case of a SIMD instruction set. For most functions used in Picnic, like AND, OR, and XOR, it does not make a difference if the 128 bits are interpreted as one value or multiple smaller values. Neighboring bits do not influence each other, and each bit is evaluated by itself. On the contrary, shifting changes the position of the bits and therefore it does make a difference how the 128 bits are interpreted.

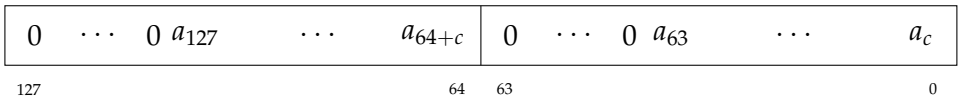
Listing 3.4 and Listing 3.5 show the implementation of a 128-bit right shift using SSE and NEON intrinsics, respectively. c describes by how many bits the vector is shifted. We will highlight the differences and similarities between the two implementations on the example of a right shift by c bits. The starting point for both algorithms is a 128-bit vector which can be split into two 64-bit parts (Figure 3.4a). We define the "lower" 64-bit value as the bits 0 to 63 and

the "upper" 64-bit value as the bits 64 to 127. There are options for shifting the 128 bits as two 64-bit values in both SIMD instruction sets. This step is done in line 7 of the SSE and in line 8 of the NEON code. The result of shifting the 128 bits as two 64-bit is a shifted vector with c zeros in bits $63 - c + 1$ to 63 and $127 - c + 1$ to 127 (Figure 3.4b). The zeros in bits $63 - c + 1$ to 63 should be filled with the original bits 64 to $64 + c - 1$ because they have to "cross the barrier" between the two 64-bit integers. Therefore, we calculate the so-called *carry*, which is used to fill the mentioned zeros. The calculation of the *carry* slightly differs in the SSE and NEON implementation because they offer different intrinsics. SSE provides the instruction _mm_bsrli_si128(__m128i a, int b) which shifts the 128-bit value a to the right by b bytes. It does not split the value into two parts but shifts it as a whole. For our right shift implementation, we shift the original input by 8 bytes in line 5 of the SSE implementation. This moves the upper 64 bits into the lower 64 bits and uses zeros to fill the upper bits. The result of this operation can be seen in Figure 3.4c. NEON does not offer such a shift operation. However, we can use the vextq_u64(uint64x2_t a, uint64x2_t b, int c) instruction to achieve the same result. This intrinsic combines the lower c ($c \in [0, 1]$) 64-bit values of b and the $2 - c$ upper 64-bit values of a to one vector, whereby the parts of b become the upper part of the result and the part of a becomes the lower part. Before we use this instruction, we set the vector which is used to calculate the carry to 0 in line 5 of the NEON implementation. We then use vextq_u64(carry, data, 1) in line 6 to compose a vector which consists of only zeros in the upper 64 bits and bits 64 to 127 in the lower 64 bits. The result is the same as using the SSE implementation with the instruction described above. Therefore, Figure 3.4c shows the intermediate result of the carry calculation for SSE and NEON. The finalization of the carry is the same for SSE and NEON. The intermediate result in Figure 3.4c has to be shifted to the left by $64 - c$ bits. This can be done using the instruction _mm_slli_epi64 in line 6 of the SSE implementation and vshlq_n_u64 in line 7 of the NEON implementation. The final *carry* is shown in Figure 3.4d. As the final step, *carry* (Figure 3.4d) and the result of the initial right shift by c bits (Figure 3.4b) can be combined by an OR to calculate the result of the 128-bit right shift. Line 8 in the SSE code and line 9 in the NEON code show this operation.

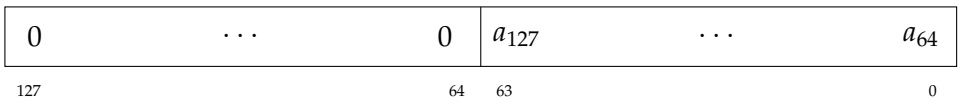
To sum it up, we propose to implement parts of Picnic using NEON intrinsics. We mainly focus on matrix operations because they can be efficiently optimized



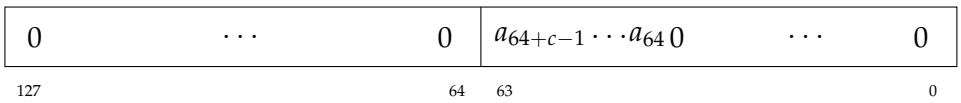
(a) The 128-bit vector split into two 64-bit parts.



(b) Result after shifting the two 64-bit values by c to the right.



(c) The upper 64 bits of the initial vector shifted to the lower 64 bits.



(d) Final carry that is built out of Figure 3.4c and can be added to the state that is shown in Figure 3.4b.

Figure 3.4.: 128-bit right shift by c bits.

```
1  __m128i shift_right(__m128i data, unsigned int c)
2  {
3    if (!c) {
4      return data;
5    }
6    __m128i carry = _mm_bsrli_si128(data, 8);
7    carry = _mm_slli_epi64(carry, 64 - c);
8    data  = _mm_srli_epi64(data, c);
9    return _mm_or_si128(data, carry);
10 }
```

Listing 3.4: 128-bit right shift using SSE intrinsics

```
1  uint64x2_t shift_right(uint64x2_t data, unsigned int c)
2  {
3    if (!c) {
4      return data;
5    }
6    uint64x2_t carry = vmovq_n_u64(0);
7    carry            = vextq_u64(data, carry, 1);
8    carry            = vshlq_n_u64(carry, 64 - c);
9    data             = vshrq_n_u64(data, c);
10   data             = vorrq_u64(data, carry);
11   return data;
12 }
```

Listing 3.5: 128-bit right shift using NEON intrinsics

using SIMD instructions. Therefore, Picnic already implements these operations using SSE and AVX code, which optimizes the signature scheme on Intel CPUs. The implementation using NEON aims to increase the performance of Picnic on ARM-based architectures, *e.g.,* smartphones, tablets and embedded devices.

4. Evaluation

This section evaluates the optimizations described in Chapter 3. We focus on memory requirements and execution time of the optimizations.

4.1. Evaluation Method

We evaluate the proposed optimizations on two different platforms. This section describes the benchmarking platforms and how the results in Section 4.2 and Section 4.3 are measured.

4.1.1. Intel Processor

The first benchmarking platform, which we will refer to as Platform A, features an Intel(R) Core(TM) i7-6700K CPU running at 4 gigahertz (GHz). It operates on 16 gigabytes (GB) RAM and runs Ubuntu 16.04 as its operating system. Picnic is compiled using GCC 5.4.

To measure "time" on the Intel CPU we use the CPU cycles as our timing unit. These cycles can be measured using so-called Performance Counters on Linux (PCL), which are also referred to as perf_events [Man17]. With perf_events it is possible to measure how many CPU cycles it takes to execute a specific instruction, function, or program. Before we can measure CPU cycles in a program, we need to initialize the perf_events using the syscall perf_event_open. This syscall returns a file descriptor. The file descriptor can be used to read the hardware CPU cycles that have passed since perf_event_open was called. To measure how long a part of a program takes, we read from the file descriptor before and after the part which should be timed. Subtracting these two values

results in the number of CPU cycles the execution of the measured program part took.

4.1.2. ARM Processor

The second benchmarking platform (Platform B) is a Raspberry Pi 3 Model B. It features a Quad Core Broadcom BCM2837 64-bit CPU running at 1.2 GHz. This ARM Cortex-A53 CPU implements the 64-bit ARMv8 architecture. It operates on 1 GB RAM and runs openSUSE Leap 42.2 as its operating system. Picnic is compiled using GCC 6.2.

The ARMv8 architecture offers the Performance Monitor Unit (PMU) to measure CPU cycles. This method of measuring time on an ARM device has been proven to be very accurate [Lip+16]. The PMU consists of several registers which store, among other information, the passed CPU cycles since the PMU has been activated or reset. Several bits in the register PMCR_EL0 (Performance Monitors Control Register, EL0) have to be set to activate the PMU. Furthermore, setting bit 31 in the register PMCNTENSET_EL0 (Performance Monitor Count Enable Set Register) enables the cycle counter. To read the current "time", *i.e.,* CPU cycles since activation of the PMU, the register PMCCNTR_EL0 (Performance Monitor Cycle Count Register) can be read.

4.2. Splitting the Round Key Computation

We evaluate the splitting of the round key computation on both benchmarking platforms using instances with a block size from 128 to 512 bits. The full details of all benchmarked Picnic instances can be found in Table 2.2. The Picnic instance with block size n is in the following referred to as Picnic-n. Furthermore, we only benchmark the Picnic variant Picnic-FS, which is the variant using the Fiat-Shamir transform (cf. Section 2.4). We do not have to benchmark both Picnic variants, Picnic-FS and Picnic-UR, because the proposed optimization is applied to the LowMC primitive which is used in the same way in Picnic-FS and Picnic-UR. We measure the time for signature creation and verification in CPU cycles and convert them to milliseconds for an easier "human-readable" comparison. All measurements are executed

Parameters	with RRK Sign	with RRK Verify	without RRK Sign	without RRK Verify	Perf. gain Sign	Perf. gain Verify
Picnic-128	1.97	1.37	2.13	1.47	8%	5%
(1000 cyc.)	7 079	5 019	7 685	5 239		
Picnic-192	5.59	3.93	9.73	7.05	43%	44%
(1000 cyc.)	20 125	14 153	35 025	25 391		
Picnic-256	14.01	10.11	23.56	16.57	41%	39%
(1000 cyc.)	50 440	36 380	84 829	59 641		
Picnic-384	100.9	69.34	184.83	126.3	45%	45%
(1000 cyc.)	363 229	249 624	665 391	454 696		
Picnic-512	241.61	165.38	444.94	304.38	46%	46%
(1000 cyc.)	869 802	595 383	1 601 767	1 095 770		

Table 4.1.: Averaged benchmarks without and with RRK on Platform A in milliseconds and 1000 cycles. The two rightmost columns show the performance gain for the sign and verify operation.

1000 times and averaged using an automated Python script. Table 4.1 and Table 4.2 show the benchmark results for Platform A and Platform B, respectively. Figure 4.1 shows a graphical representation of the performance gain on Platform A.

The performance gains described in Table 4.1 and Table 4.2 show that the proposed optimization increases the performance by more than 40% for large block sizes. Furthermore, we can observe that the performance gain is 5/9 times higher for signature creation/verification on Platform B than on Platform A for a block size of 128. However, the difference in performance gain between the two platforms becomes smaller as the block size increases.

Memory Requirements. The memory requirements of the optimized algorithm strongly depend on the parameters of LowMC. The number of S-boxes has a significant influence on how much the required memory can be reduced compared to the original algorithm. We will demonstrate how the number of S-boxes influences the reduction of memory requirements based on a block and key size of 256 bits. The original LowMC algorithm with 38 rounds and 10 S-boxes requires 38 (256×256) matrices for the linear layer, 39 (256×256)

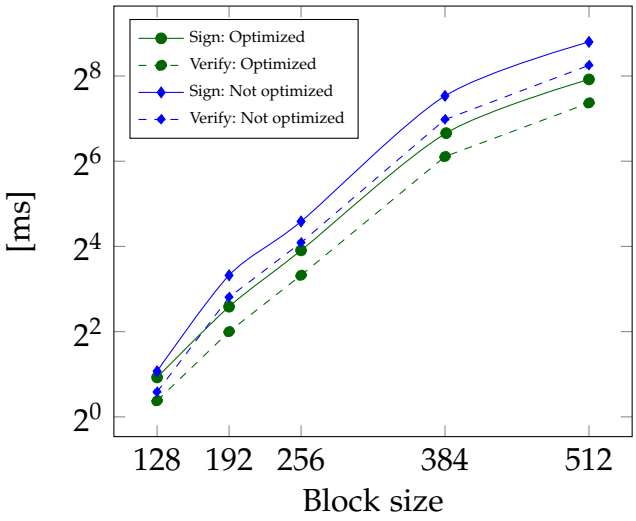


Figure 4.1.: Performance of the original and the optimized LowMC algorithm in milliseconds depending on the block size.

matrices for the round keys, and 38 256-bit vectors for the round constants. This results in a theoretical memory requirement of 632 kilobytes (KB). The required memory for the original algorithm depends on the block and key size but is independent of the number of S-boxes.

The optimized LowMC algorithm with 38 rounds and 10 S-boxes requires 38 (256×256) matrices for the linear layer, 1 (256×256) matrix for the linear part of the round keys, 1 $(30 \cdot 38 \times 256)$ matrix for the non-linear parts of the round keys, 38 256-bit vectors for the round constants, and a $(30 \cdot 38)$-bit vector as temporary storage. In total, this sums up to a memory requirement of 357.33 KB. This is a reduction of the memory requirements by 43%. If we use 20 S-boxes the number of rounds can be decreased to 21 to achieve the same security level as with 10 S-boxes. Using the optimized algorithm with these parameters (21 rounds and 20 S-boxes) requires 221.37 KB of memory. Compared to the original algorithm the required memory is reduced by 65%. This is even 38% less than in the optimized algorithm with the first proposed parameter set (38 rounds and 10 S-boxes).

In the practical implementation, the requirements are slightly higher because every matrix has to store metadata about itself. The metadata includes the

Parameters	with RRK Sign	with RRK Verify	without RRK Sign	without RRK Verify	Perf. gain Sign	Perf. gain Verify
Picnic-128	30.41	21.15	55.49	38.43	45%	45%
(1000 cyc.)	36 495	25 374	66 584	46 121		
Picnic-192	116.22	78.36	187.67	126.73	38%	38%
(1000 cyc.)	139 459	94 037	225 208	152 078		
Picnic-256	241.27	161.47	405.82	272.09	40%	40%
(1000 cyc.)	289 523	193 767	486 989	326 503		
Picnic-384	1259.40	832.41	2363.71	1566.67	46%	46%
(1000 cyc.)	1 511 283	998 888	2 836 456	1 879 999		
Picnic-512	3108.42	2053.59	5929.80	3921.24	48%	48%
(1000 cyc.)	3 730 099	2 464 312	7 115 754	4 705 494		

Table 4.2.: Averaged benchmarks without and with RRK on Platform B in milliseconds and 1000 cycles. The two rightmost columns show the performance gain for the sign and verify operation.

number of rows and columns, the width of one row in memory, and the distance between two rows in memory (rowstride). The unit of the latter two elements is `uint64_t`, *e.g.,* a matrix where each row consists of two `uint64_t` has a rowstride of 2. Each matrix requires 32 bytes of memory for its metadata. This results in 3.59 KB of metadata in the original algorithm and 2.44 KB in the optimized algorithm for a block size of 256 bits.

4.3. Replacing L_i with a Feistel Network

We evaluate the proposed Feistel on a $(n \times n) \cdot (n \times 1)$ multiplication using the block sizes n 128, 256, 512, and 1024 and the branch sizes $2^i, i \in \{2, 3, 4, 5, 6, 7, 8\}$. As a reference, we use the implementation of the matrix-vector multiplication in Picnic. This implementation uses SSE2 code for multiplications with a block size which is a multiple of 128. Furthermore, it uses AVX2 code if the block size is a multiple of 256. We will refer to this implementation as Picnic multiplication. All following benchmarks for the Picnic multiplication and the Feistel network show 500000 averaged executions.

Block size	Cycles	Memory (in bytes)
128	1033	2048
256	2260	8192
512	4356	32768
1024	11580	131072

Table 4.3.: Execution time in cycles and memory requirements in bytes for the Picnic multiplication with different block sizes (Platform A).

Table 4.3 shows the performance and the memory consumption of the Picnic multiplication on Platform A for the block sizes $128, 256$, 512, and 1024. Table 4.4 shows the performance of the Feistel network which represents the $(n \times n) \cdot (n \times 1)$ multiplication with $n \in \{128, 256, 512, 1024\}$. In general, the performance of the Feistel network increases with the branch size w. We can observe that for each block sizes there exists a branch size which results in a performance of the Feistel network that is close to the performance of the Picnic multiplication. For the 128-bit Feistl network a branch size of 32 bits is most favorable in terms of performance. However, this construction is still around 50% slower than the Picnic multiplication. The Feistel structure for 256 bits block size with a branch size of 64 bits achieves a 10% better performance than the Picnic implementation. In contrast, the Feistel structure for the 512-bit multiplication with a branch size of 64 bits is 10% slower than the Picnic implementation. There are four branch sizes 16, 64, 128, and 256 in the Feistel network that implements the 1024-bit multiplication that acchieve a performance that is less than 10% slower than the Picnic multiplication. The performance that is closest to the Picnic multiplication is the Feistel network with a branch size of 128 bits.

Memory Requirements. The Picnic multiplication only requires the $(n \times n)$ matrix, which results in a memory requirement (excluding the metadata) of $\frac{n \cdot n}{8}$ bytes. As Table 4.3 shows, this results in 2048 bytes for 128-bit block size, 8192 bytes for 256-bit block size, 32768 bytes for 512-bit, and 131072 bytes for 1024-bit block size. Table 4.5 shows the required memory for the Feistel network for different branch sizes in bytes. The required memory can be described as $\frac{w \cdot w \cdot b \cdot r}{8}$ bytes. We can observe that for all block sizes and all branch sizes the memory consumption can be lower with the Feistel network than with the

		Block size							
		128		256		512		1024	
		R	**Cyc.**	**R**	**Cyc.**	**R**	**Cyc.**	**R**	**Cyc.**
Branch size	4	9	1785	10	4463	12	9994	13	22586
	8	8	1359	9	3224	10	6874	12	16975
	16	6	960	8	2430	9	5337	10	12189
	32	5	1627	6	3231	8	9159	9	20405
	64	-	-	5	2047	6	4855	8	12622
	128	-	-	-	-	5	7766	6	11695
	256	-	-	-	-	-	-	5	12468

Table 4.4.: Benchmarks of the Feistel network implementing a multiplication with 128, 256, 512 and 1024 bits block size and different branch sizes on Platform A. **R** denotes the number of necessary rounds in the Feistel network and **Cyc.** the number of cycles the execution of the Feistel network takes.

		Block size			
		128	256	512	1024
Branch size	4	288	640	1536	3328
	8	512	1152	2560	6144
	16	768	2048	4608	10240
	32	1280	3072	8192	18432
	64	-	5120	12288	32768
	128	-	-	20480	49152
	256	-	-	-	81920

Table 4.5.: Memory requirements of the Feistel network with different branch sizes and the block sizes 128, 256, 512, and 1024.

Picnic multiplication. For the 128-bit and 256-bit multiplication the required memory is lower in the Feistel network for all possible branch sizes. The Feistel network can reduce the memory consumption by 86% for the 128-bit multiplication, 92% for the 256-bit multiplication, 95% for the 512-bit, and 97% for the 1024-bit multiplication. All of these memory savings occur with a branch size of 4. We note, that depending on the implementation the memory requirements may be higher than theoretically assumed.

4.4. NEON Instruction Set Extension

Table 4.6 and Table 4.7 show the performance of Picnic with and without NEON intrinsics. For the benchmarks in Table 4.6 the optimization of Section 3.1, the reduced round key computation (RRK), was enabled. Table 4.7 shows the benchmarks where the RRK was not enabled. For all benchmarks without NEON the compiler flag "-march=armv8+nosimd" was used to prevent the compiler from generating NEON code on its own. In both cases, with and without the RRK, the performance gain is highest for the 128-bit block size. With the RRK the performance gain is 13% and 10% for signing and verifying, respectively. The performance of the original LowMC algorithm can be increased by 18% for signing and 17% for verifying when NEON instructions are used. For all other block sizes the performance gain does not exceed 5%. If the RRK is disabled the performance for a block size of 192 bits decreases by 4% due to the usage of NEON instructions.

	with NEON		without NEON		Perf. gain	
Parameters	Sign	Verify	Sign	Verify	Sign	Verify
PICNIC-128	30.18	21.10	34.95	23.61	13%	10%
(1000 cyc.)	36 220	25 324	41 937	28 332		
PICNIC-192	121.49	81.96	126.36	85.37	4%	4%
(1000 cyc.)	145 783	98 351	151 637	102 446		
PICNIC-256	235.22	157.42	247.63	166.25	5%	5%
(1000 cyc.)	282 266	188 900	297 158	199 497		
PICNIC-384	1259.61	832.53	1270.71	845.56	1%	2%
(1000 cyc.)	1 511 534	999 031	1 524 848	1 014 669		
PICNIC-512	3107.63	2054.78	3274.69	2193.11	5%	6%
(1000 cyc.)	3 729 152	2 465 733	3 929 628	2 631 737		

Table 4.6.: Averaged benchmarks without and with NEON instructions on Platform B in milliseconds and 1000 cycles. The two rightmost columns show the performance gain for the sign and verify operation. The RRK was enabled for this benchmark

	with NEON		without NEON		Perf. gain	
Parameters	Sign	Verify	Sign	Verify	Sign	Verify
PICNIC-128	48.31	33.30	58.86	40.151	18%	17%
(1000 cyc.)	57 967	39 959	70 626	48 182		
PICNIC-192	208.90	140.55	201.46	135.63	-4%	-4%
(1000 cyc.)	250 684	168 661	241 753	162 761		
PICNIC-256	405.65	271.95	456.66	306.08	11%	11%
(1000 cyc.)	486 777	326 334	547 993	367 301		
PICNIC-384	2376.95	1575.00	2402.92	1610.26	1%	2%
(1000 cyc.)	2 852 335	1 889 999	2 883 505	1 932 307		
PICNIC-512	5967.48	3956.89	6243.58	4189.65	4%	6%
(1000 cyc.)	7 160 976	4 748 273	7 492 300	5 027 582		

Table 4.7.: Averaged benchmarks without and with NEON instructions on Platform B in milliseconds and 1000 cycles. The two rightmost columns show the performance gain for the sign and verify operation. The RRK was disabled for this benchmark

5. Discussion

This chapter discusses the results and benchmarks which are presented in Chapter 4. We will assess performance gains, memory consumption, and other issues of all proposed optimizations.

5.1. Splitting the Round Key Computation

In this section we focus on effects of the LowMC algorithm which uses a split round key. Section 5.1.1 focuses on the memory consumption and the performance gain of the optimized LowMC variant when larger block sizes are used. In Section 5.1.2 we highlight the effects of different numbers of S-boxes. We discuss how the number of S-boxes influences characteristics like memory consumption, ANDdepth, and signature size.

5.1.1. Large Block Sizes

The splitting of the round key becomes particularly useful when the block size increases. If the number of S-boxes does not proportionally rise with the block size, the memory that can be saved by the precomputation increases. As Table 5.1 shows, if the number of S-boxes is fixed the memory that is saved increases with the block size. The algorithm with the split round key saves almost 50% of the required memory compared to the original algorithm when 10 S-boxes are used.

The performance gain can only be achieved if the number of S-boxes stays the same even if the block size increases. The S-box layer is computationally more expensive than all other layers because it involves AND operations. Therefore,

Block size	Without RRK	With RRK	Savings
128	83.29 KB	53.01 KB	36%
192	281.81 KB	165.28 KB	41%
256	632 KB	357.33 KB	44%
384	2122.42 KB	1152.09 KB	46%
512	5149.57 KB	2743.72 KB	47%

Table 5.1.: Memory requirements for 10 S-boxes and different block sizes for the original (without RRK) and the optimized (with RRK) LowMC algorithm.

it is favorable for the performance to decrease the proportion of the state on which the S-boxes are applied. However, one has to keep in mind that a lower number of S-boxes results in more rounds of the LowMC algorithm.

5.1.2. Number of S-boxes

As briefly described in Section 4.2, the choice of the number of S-boxes m for the LowMC algorithm in Picnic has a huge influence on the memory requirements. The memory requirements for the optimized algorithm with 20 S-boxes are 35% lower than for the optimized algorithm with 10 S-boxes. This effect occurs because the number of rounds can be decreased if the number of S-boxes is increased. Therefore, a decreased round number results in faster execution time and reduced memory requirements. The additional S-boxes do, up to the point where $3 \cdot m > 256$, not increase the runtime of the algorithm because the bitsliced S-box implementation can calculate all S-boxes in parallel. While $3m$ is smaller than 256 the bitsliced S-box implementation can use SSE or AVX registers to process all S-boxes at once. When $3m$ bits exceed the capacity of an AVX register the runtime of the S-box layer might increase slightly because we require 2 AVX registers to compute the S-box layer. This constraint only affects the proposed instances with the block sizes 384 and 512.

Although it might look favorable to increase the number of S-boxes to the possible maximum in terms of execution time, this has a significant drawback as well. The number of AND gates in the circuit that implements the LowMC encryption increases. This makes the hardware implementation of the circuit

more expensive because it requires more gates and space. However, the lower number of rounds is favorable for the ANDdepth of the circuit and therefore alleviates the effect of the increased number of AND gates partly.

Another effect of the number of S-boxes which should not be left out of consideration is the signature size. If the number of S-boxes is decreased, the signature size decreases as well. The signature size is dependent on the size of the calculated views, and the size of the views is dependent on the number of S-boxes. A lower number of S-boxes reduces the view size even though the number of rounds of LowMC has to be increased to achieve the same security level.

5.2. Replacing L_i with a Feistel Network

This section discusses results of the Feistel structure that was proposed in Section 3.2. We discuss the memory consumption in relation to the achieved performance. Furthermore, we discuss the usage of the Feistel network when large block sizes are used. At the end, we compare the number of XORs in the Feistel network and a standard constant-time implementation.

5.2.1. Memory Consumption

As Table 4.5 shows, the memory consumption of the Feistel network increases with the used branch size. For all evaluated block sizes the memory consumption is the lowest if a branch size of 4 is used. Compared to the Picnic multiplication the Feistel network can save up to 97% of the required memory in the 1024 bits block size case. If this theoretical saving can be achieved in practice depends on the concrete implementation of the used data structures. However, as Table 4.4 shows, the performance of the Feistel structure increases with the branch size. For all evaluated block sizes, a branch size of 4 has the least favorable performance compared to other branch sizes. Hence, it is not possible to achieve the lowest memory consumption and the best performance at the same time. One has to make a tradeoff between execution time and required memory. In which direction this tradeoff drifts can be made dependent on the target platform or the application of the scheme. If the target

platform is an embedded device with limited memory then it may be favorable to decrease the memory requirements at the cost of a lower performance. In contrast, on a modern laptop or PC a few mega bytes of extra memory are of no consequence with regard to the gigabytes that are availabe. Therefore, the optimization of the execution time may be of more importance.

5.2.2. Large Block Sizes and SIMD Registers

When a block size of 1024 bits is used, the maximum branch size is 256 bits. 256 is exactly the size of an AVX register. Hence, all branch sizes greater or equal to 256 bits can make use of the 256-bit registers. The AVX variant can save up to $\frac{3}{4}$ of the operations for certain functions compared to an implementation which uses 64-bit `uint64_t`. This effect occurs because one 256-bit branch can be handled in one AVX instruction instead of four 64-bit instructions. Therefore, using the AVX registers is of great interest for block sizes > 1024 bits because they can reduce the number of executed instructions and as a result the execution time. For the 512-bit Feistel network the AVX registers cannot be applied because the largest possible branch size is 128. However, the Feistel network can use SSE intrinsics on Intel CPUs and NEON intrinsics on ARM CPUs. These intrinsics reduce the number of instructions by $\frac{1}{2}$ for certain implementation parts.

In general, we can observe that it is possible for all block sizes to achieve a performance of the Feistel network that is close to the performance of the Picnic multiplication. For all evaluated block sizes the Feistel network yields the best performance if the used branch size is the largest or second largest possible branch size. Therefore, for these branch sizes, the use of AVX, SSE, and NEON registers is worth consideration. However, our experiments showed no significant performance gain when SIMD instructions were used in the Feistel network. This may be due to the small amount of data the instructions are applied to. Another possible explanation is that the compiler can generate highly optimized and efficient code when no SIMD instructions are used. In this case, the usage of SIMD instructions may not yield a reasonable performance gain anymore.

5.2.3. Number of XORs

Table 5.2 shows the number of necessary XOR operations in the constant-time multiplication and in the Feistel network. The number of XOR operations in the constant-time multiplication exactly quadruples when the block size is doubled. In contrast, the Feistel network less than triples the number of XOR operations when doubling the block size. The larger the block size is the more XOR operations can be saved using the Feistel network. The numbers in Table 5.2 consider only the XORs that are necessary for the multiplications in the Feistel network. The XOR which adds the result of the multiplication to the current state of the Feistel network can be neglected in the number of XORs. This is possible because the multiplication can be implemented in a way that it directly operates on the state so that no extra addition is necessary.

The multiplications in the branches of the Feistel network can be implemented using SSE intrinsics for all block sizes larger than 512 bits. AVX intrinsics can be used starting from a block size of 1024 bits. These SIMD intrinsics should reduce the number of XORs. Needless to say, the SSE and AVX intrinsics can also be used in the constant-time implementation.

Blocksize	**Rounds**	**XOR operations in Feistel network**	**XOR operations with constant-time mult.**
256	5	640	1024
512	6	1536	4096
1024	8	4096	16384
2048	9	9216	65536
4096	10	20480	262144

Table 5.2.: Number of XOR operations in the Feistel network with 64-bit branches and the constant-time multiplication for different block sizes.

5.3. NEON Instruction Set Extension

Benchmark results of the Picnic variants that use SSE intrinsics show that the SIMD instructions can provide a performance gain of up to 30% [Cha+17b]. As

Table 4.6 and Table 4.7 show, the performance gain with NEON instructions is much lower than 30%. This observation might be caused by different factors. These factors can include, but may be not limited to, compiler optimization issues, out-of-order execution, and hardware limitations.

One explanation for the low performance gain may be the compiler itself. The compiler may not be equally capable of generating highly optimized code if the compiled code includes NEON instructions than if it does not. This might be due to the fact that optimizing code without NEON intrinsics has been done for a much longer time and is, therefore, more powerful. NEON instructions have only been introduced in 2005 which could mean that there is still optimization potential.
A factor that may also influence the performance gain is out-of-order execution. Most Intel CPUs provide out-of-order execution whereas the Raspberry Pi does not [ARM14]. Out-of-order execution enables the CPU to execute instructions not in the provided order, but in a more optimal way if this is possible. For example, the CPU may reorder arithmetic operations followed by a load operation if they are independent of each other. In this case, it is more optimal to start the load operation before the arithmetic operation because the load needs more cycles to complete. Therefore, out-of-order execution can provide a considerable speedup because code that is not written optimally can be reordered. The Raspberry Pi does not offer this feature which means that the performance is highly dependent on the written code.
A Raspberry Pi has, compared to an Intel processor, much weaker and smaller hardware. One of the hardware factors that might influence the performance gain is the time it takes to fetch data from memory. The relation of the duration of a fetch from memory and the duration of executing other instructions plays an important role in the performance gain. If the memory fetch takes longer compared to other instructions on the Raspberry Pi, the executed instructions are not the bottleneck of the algorithm. In this case, it would be more useful to minimize the reads from memory. Another hardware effect that could influence the performance gain is caching. Caches on Intel and ARM CPUs behave differently concerning replacement policy [Lip+16]. Furthermore, the cache on the Raspberry Pi is much smaller than on the Intel processor which means that more data has to be fetched from memory.

6. Conclusion

In this document, we proposed three optimizations on Picnic and especially its component LowMC. The goal of all three optimizations is to decrease the execution time or the memory consumption of Picnic. The first optimization splits the round key into a linear and a non-linear part and the linear part is moved to the beginning of the encryption. This structural change removes the multiplication $K_i \cdot y$ from every LowMC round. The round key computation in each LowMC round is reduced from one $(n \times n) \cdot (n \times 1)$ multiplication and one n-bit addition to one $3m$-bit addition. This has shown to be a very effective optimization. We can increase the performance of Picnic by almost 50% by applying this optimization. Furthermore, also the memory requirements for our fixed LowMC instances with 10 S-boxes can be reduced by almost 50%. The second optimization addresses the linear layer of LowMC. We proposed to replace the multiplication in the linear layer by a Feistel network. The multiplication can be implemented as the round functions of the Feistel network. This Feistel network yields a similar performance to the original linear layer. However, the memory consumption can be reduced by up to 97% if a performance loss is acceptable. This reduction in memory requirements can be achieved because the matrix of the linear layer is not stored as one $(n \times n)$ matrix, but several smaller matrices. The third optimization implements parts of Picnic using NEON intrinsics. NEON intrinsics should reduce the number of necessary XOR operations for matrix operations like multiplication. The usage of the SIMD instructions increases the performance slightly for all evaluated block sizes. The performance gain is highest if the block size is 128 bits.

6.1. Future Work

One research direction that directly arises from this thesis is a deeper look into the NEON instruction set extension. As we did not manage to achieve the expected results, it may be interesting to have a closer look at how the compiler generates code around the NEON instructions. Furthermore, it may also be worth reconsidering the code design of Picnic. The implementation itself could be the problem that it is not possible to use NEON instructions effectively. It could also be possible that on a different ARM platform the NEON optimizations yield a higher performance gain.

A possible application for the Feistel network is Rasta [Dob+18]. Rasta requires randomly sampled matrices for a linear layer. The size of these matrices depends on the block size n of the Rasta instance. As the block size of Rasta can quickly reach more than 1024 bits, the Feistel network could ease the generation of the matrices for the linear layer. It would be more efficient concerning execution time and memory consumption if not a matrix of size $(n \times n)$ but several smaller matrices are generated.

Appendix A.

Feistel Network in Python

This appendix shows an implementation of the Feistel network which is presented in Section 3.2. First, the code generates random $(W \times W)$ matrices (lines 57-59). Second, the matrix M is computed from random these random matrices (lines 62-67). Finally, we apply the Feistel network on a random input vector s to check if it truly computes $M \cdot s$ (lines 71-75).

```
#!/usr/bin/sage
from sage.all import *
import itertools, math
F = GF(2)
N = 128                     # block size N
W = 4                       # branch width
B = int(N/(2*W))            # N = 2*B*W

R = 1
while fibonacci(R) <= B:  #calculate rounds for full diffusion
    R += 1

def rand_linear_permutation(w, density=0.5):
    #Returns a random w x w matrix
    while True:
        result = zero_matrix(F, w, w)
        for i, j in itertools.product(xrange(0, w), repeat=2):
            if random() < density:
                result[i,j] = 1
        if result.rank() == w:
            return result

def permute_left(r):
    return 0 if r % 2 else B

def permute_right(r):
    return permute_left(r + 1)

def feistel_network(i, rot, s, Ls):
    for i in xrange(0, R):
        for b in xrange(0, B):
            L = Ls[b]
            idx = (permute_right(i) + b) * W
            x = s[idx : idx  + W]
            y = L * x
            idx = (permute_left(i) + ((rot + b) % B)) * W
            s[idx : idx + W] += y
    if R % 2 == 1:
        tmp = s
        s[0 : B*W] = tmp[B*W : 2*B*W]
        s[B*W : 2*B*W] = tmp[0 : B*W]
    return s
```

```
43 def random_rotated_feistel_round(rot):
44     small_matrices = []
45
46     for i in xrange(0, B):
47         small_matrices.append(rand_linear_permutation(W))
48     return small_matrices
49
50 if __name__ == "__main__":
51
52     phi = [x % B for x in fibonacci_sequence(R)]
53     print "B = {}, W = {}, N = {} R = {}\nphi = {}".format(B, W, N
       ↪ , R, phi)
54
55     all_round_matrices = []
56     #generate all R*B small matrices
57     for i in xrange(0, R):
58         round_matrices_one_round = random_rotated_feistel_round(
           ↪ phi[i])
59         all_round_matrices.append(round_matrices_one_round)
60
61     #generate matrix M out of small matrices
62     M = matrix(F, N, N)
63     for j in xrange(N):
64         s = vector(F, N)
65         s[j] = 1
66         s = feistel_network(i, phi[i], s, all_round_matrices[i])
67         M[:,j] = s
68
69     #check Feistel network by comparing M*s to result of
70     # multiplication with Feistel network
71     s = random_vector(F, N)
72     t = M * s
73     s = feistel_network(i, phi[i], s, all_round_matrices[i])
74     #print if comparison was successful
75     print(t == s)
```

Listing A.1: Fibonacci network, which is described in Section 3.2, implemented in Python.

Bibliography

[Abd+02] Michel Abdalla, Jee Hea An, Mihir Bellare, and Chanathip Namprempre. "From Identification to Signatures via the Fiat-Shamir Transform: Minimizing Assumptions for Security and Forward-Security." In: *EUROCRYPT*. 2002, pp. 418–433 (cit. on p. 15).

[Abd+12] Michel Abdalla, Pierre-Alain Fouque, Vadim Lyubashevsky, and Mehdi Tibouchi. "Tightly-Secure Signatures from Lossy Identification Schemes." In: *EUROCRYPT*. 2012, pp. 572–590 (cit. on p. 15).

[Ajt96] Miklós Ajtai. "Generating Hard Instances of Lattice Problems." In: *Electronic Colloquium on Computational Complexity (ECCC)* 3.7 (1996) (cit. on p. 8).

[Akl+16] Sedat Akleylek, Nina Bindel, Johannes A. Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. "An Efficient Lattice-Based Signature Scheme with Provably Secure Instantiation." In: *AFRICACRYPT*. 2016, pp. 44–60 (cit. on p. 8).

[Alb+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. "Ciphers for MPC and FHE." In: *EUROCRYPT*. 2015 (cit. on p. 9).

[Alb+16a] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity." In: *ASIACRYPT*. 2016, pp. 191–219 (cit. on p. 21).

[Alb+16b] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. "Ciphers for MPC and FHE." In: *IACR Cryptology ePrint Archive* (2016), p. 687 (cit. on pp. 3, 9, 11, 12).

[Alk+15] Erdem Alkim, Nina Bindel, Johannes A. Buchmann, and Özgür Dagdelen. "TESLA: Tightly-Secure Efficient Signatures from Standard Lattices." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 755 (cit. on pp. 3, 8).

[Are+05] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. "DNS Security Introduction and Requirements." In: *RFC* 4033 (2005), pp. 1–21 (cit. on p. 1).

[ARM14] ARM. *ARM Cortex-A53 MPCore Processor*. Technical Reference Manual. 2014 (cit. on p. 66).

[Bar+16] Paulo S. L. M. Barreto, Patrick Longa, Michael Naehrig, Jefferson E. Ricardini, and Gustavo Zanon. "Sharper Ring-LWE Signatures." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 1026 (cit. on p. 8).

[BB13] Rachid El Bansarkhani and Johannes A. Buchmann. "Improvement and Efficient Implementation of a Lattice-Based Signature Scheme." In: *SAC*. 2013, pp. 48–67 (cit. on p. 8).

[BBD09] Daniel J Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-quantum cryptography*. Springer Science & Business Media, 2009 (cit. on pp. 2, 6–8).

[BD93] Josh Benaloh and Michael De Mare. "One-way accumulators: A decentralized alternative to digital signatures." In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1993, pp. 274–285 (cit. on p. 22).

[Ber+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures." In: *J. Cryptographic Engineering* 2.2 (2012), pp. 77–89 (cit. on p. 5).

[Ber+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. "SPHINCS: Practical Stateless Hash-Based Signatures." In: *EUROCRYPT*. 2015, pp. 368–397 (cit. on pp. 3, 7, 8).

[BG92] Mihir Bellare and Oded Goldreich. "On Defining Proofs of Knowledge." In: *CRYPTO*. 1992, pp. 390–420 (cit. on p. 14).

[Bon+11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. "Random Oracles in a Quantum World." In: *ASIACRYPT*. 2011, pp. 41–69 (cit. on p. 17).

[BPP00] Joan Boyar, René Peralta, and Denis Pochuev. "On the multiplicative complexity of Boolean functions over the basis (cap, +, 1)." In: *Theor. Comput. Sci.* 235.1 (2000), pp. 43–57 (cit. on p. 11).

[BPS16] Mihir Bellare, Bertram Poettering, and Douglas Stebila. "From Identification to Signatures, Tightly: A Framework and Generic Transforms." In: *ASIACRYPT*. 2016, pp. 435–464 (cit. on p. 15).

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols." In: *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*. 1993, pp. 62–73 (cit. on p. 15).

[CDM00] Ronald Cramer, Ivan Damgård, and Philip D. MacKenzie. "Efficient Zero-Knowledge Proofs of Knowledge Without Intractability Assumptions." In: *PKC*. 2000, pp. 354–373 (cit. on p. 13).

[Cha+17a] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. "Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives." In: *CCS*. 2017, pp. 1825–1842 (cit. on pp. 3, 13, 16, 18, 21).

[Cha+17b] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. *The Picnic Signature Scheme: Design Document*. `https://github.com/Microsoft/Picnic/blob/master/spec/design-v1.0.pdf`. 2017 (cit. on p. 65).

[Che+16] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. "From 5-Pass *MQ* -Based Identification to *MQ* -Based Signatures." In: *ASIACRYPT*. 2016, pp. 135–165 (cit. on pp. 3, 8).

[Dag+14] Özgür Dagdelen, Rachid El Bansarkhani, Florian Göpfert, Tim Güneysu, Tobias Oder, Thomas Pöppelmann, Ana Helena Sánchez, and Peter Schwabe. "High-Speed Signatures from Standard Lattices." In: *LATINCRYPT*. 2014, pp. 84–103 (cit. on p. 8).

[Dag+16] Özgür Dagdelen, David Galindo, Pascal Véron, Sidi Mohamed El Yousfi Alaoui, and Pierre-Louis Cayrel. "Extended security arguments for signature schemes." In: *Des. Codes Cryptography* 78.2 (2016), pp. 441–461 (cit. on p. 15).

[Der+16] David Derler, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, and Daniel Slamanig. *Digital Signatures from Symmetric-Key Primitives*. Cryptology ePrint Archive, Report 2016/1085. 2016 (cit. on pp. 15, 16).

[DH76] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography." In: *IEEE Trans. Information Theory* 22.6 (1976), pp. 644–654 (cit. on p. 6).

[Dob+18] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. *Rasta: A cipher with low ANDdepth and few ANDs per bit*. Cryptology ePrint Archive, Report 2018/181. `https://eprint.iacr.org/2018/181`. 2018 (cit. on p. 68).

[Dor+14] Yarkin Doröz, Aria Shahverdi, Thomas Eisenbarth, and Berk Sunar. "Toward Practical Homomorphic Evaluation of Block Ciphers Using Prince." In: *FC, BITCOIN and WAHC*. 2014, pp. 208–220 (cit. on p. 11).

[DR08] Tim Dierks and Eric Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.2." In: *RFC* 5246 (2008), pp. 1–104 (cit. on p. 1).

[DRS17] David Derler, Sebastian Ramacher, and Daniel Slamanig. *Post-Quantum Zero-Knowledge Proofs for Accumulators with Applications to Ring Signatures from Symmetric-Key Primitives*. Cryptology ePrint Archive, Report 2017/1154. 2017 (cit. on p. 22).

[Duc+13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. "Lattice Signatures and Bimodal Gaussians." In: *CRYPTO*. 2013, pp. 40–56 (cit. on pp. 3, 8).

[Fau+12] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. "On the Non-malleability of the Fiat-Shamir Transform." In: *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*. 2012, pp. 60–79 (cit. on p. 15).

[Fei73] Horst Feistel. "Cryptography and computer privacy." In: *Scientific american* 228.5 (1973), pp. 15–23 (cit. on p. 22).

[Fly72] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness." In: *IEEE Trans. Computers* 21.9 (1972), pp. 948–960 (cit. on p. 24).

[FS86] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems." In: *CRYPTO*. 1986, pp. 186–194 (cit. on p. 14).

[Gam84] Taher El Gamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms." In: *CRYPTO*. 1984, pp. 10–18 (cit. on p. 5).

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. "Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems." In: *CHES*. 2012, pp. 530–547 (cit. on p. 8).

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. "ZKBoo: Faster Zero-Knowledge for Boolean Circuits." In: *USENIX*. 2016, pp. 1069–1083 (cit. on pp. 17, 18).

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof-Systems." In: *STOC*. 1985, pp. 291–304 (cit. on p. 13).

[GOS89] Gosudarstvennyi Standard GOST. "28147-89,"." In: *Cryptographic protection for data processing systems," Government Committee of the USSR for Standards* (1989) (cit. on p. 22).

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions." In: *STOC*. 2008, pp. 197–206 (cit. on p. 8).

[Gro96] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search." In: *STOC*. 1996, pp. 212–219 (cit. on pp. 3, 7).

[GS84] David K. Gifford and Alfred Z. Spector. "The Space Shuttle Primary Computer System." In: *Commun. ACM* 27.9 (1984), pp. 872–900 (cit. on p. 24).

[Ham15] Mike Hamburg. "Ed448-Goldilocks, a new elliptic curve." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 625 (cit. on p. 5).

[Ish+09] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. "Zero-Knowledge Proofs from Secure Multiparty Computation." In: *SIAM J. Comput.* 39.3 (2009), pp. 1121–1152 (cit. on p. 17).

[JL17] Simon Josefsson and Ilari Liusvaara. "Edwards-Curve Digital Signature Algorithm (EdDSA)." In: *RFC* 8032 (2017), pp. 1–60 (cit. on p. 5).

[Kat10] Jonathan Katz. *Digital signatures*. Springer Science & Business Media, 2010 (cit. on p. 5).

[KD79] John B. Kam and George I. Davida. "Structured Design of Substitution-Permutation Encryption Networks." In: *IEEE Trans. Computers* 28.10 (1979), pp. 747–753. DOI: 10.1109/TC.1979.1675242. URL: https://doi.org/10.1109/TC.1979.1675242 (cit. on p. 9).

[KMP16] Eike Kiltz, Daniel Masny, and Jiaxin Pan. "Optimal Security Proofs for Signatures from Identification Schemes." In: *CRYPTO*. 2016, pp. 33–61 (cit. on p. 15).

[Lam79] Leslie Lamport. *Constructing digital signatures from a one-way function*. Tech. rep. Technical Report CSL-98, SRI International Palo Alto, 1979 (cit. on p. 7).

[Lip+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX*. 2016, pp. 549–564 (cit. on pp. 52, 66).

[Lyu09] Vadim Lyubashevsky. "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures." In: *ASIACRYPT*. 2009, pp. 598–616 (cit. on p. 8).

[Lyu12] Vadim Lyubashevsky. "Lattice Signatures without Trapdoors." In: *EUROCRYPT*. 2012, pp. 738–755 (cit. on p. 8).

[Man17] Linux Programmer's Manual. *perf_event_open*. Linux man page. 2017 (cit. on p. 51).

[McE78] Robert J McEliece. "A public-key cryptosystem based on algebraic coding theory." In: *Coding Thv* 4244 (1978), pp. 114–116 (cit. on p. 9).

[Mer89] Ralph C. Merkle. "A Certified Digital Signature." In: *CRYPTO*. 1989, pp. 218–238 (cit. on p. 7).

[Mil85] Victor S. Miller. "Use of Elliptic Curves in Cryptography." In: *CRYPTO*. 1985, pp. 417–426 (cit. on p. 6).

[Moc87] Paul V. Mockapetris. "Domain names - concepts and facilities." In: *RFC* 1034 (1987), pp. 1–55 (cit. on p. 1).

[MOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996 (cit. on pp. 2, 9, 12, 16, 22).

[Nie86] Harald Niederreiter. "Knapsack-type cryptosystems and algebraic coding theory." In: *Prob. Control and Inf. Theory* 15.2 (1986), pp. 159–166 (cit. on pp. 3, 9).

[Nyb96] Kaisa Nyberg. "Generalized Feistel Networks." In: *ASIACRYPT*. 1996, pp. 91–104 (cit. on p. 23).

[OO98] Kazuo Ohta and Tatsuaki Okamoto. "On Concrete Security Treatment of Signatures Derived from Identification." In: *CRYPTO*. 1998, pp. 354–369 (cit. on p. 15).

[Per+17] Léo Perrin, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. *Improvements to the Linear Layer of LowMC: A Faster Picnic*. Cryptology ePrint Archive, Report 2017/1148. 2017 (cit. on pp. 27, 29, 31, 32, 37, 40).

[PS96] David Pointcheval and Jacques Stern. "Security Proofs for Signature Schemes." In: *EUROCRYPT*. 1996, pp. 387–398 (cit. on p. 15).

[Qui+89] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michael Quisquater, Louis C. Guillou, Marie Annick Guillou, Gaid Guillou, Anna Guillou, Gwenolé Guillou, Soazig Guillou, and Thomas A. Berson. "How to Explain Zero-Knowledge Protocols to Your Children." In: *CRYPTO*. 1989, pp. 628–631 (cit. on p. 13).

[Ram+06] RM Ramanathan, Ron Curry, Srinivas Chennupaty, Robert L Cross, Shihjong Kuo, and Mark J Buxton. "Extending the world's most popular processor architecture." In: *Intel Whitepaper* (2006) (cit. on p. 26).

[Riv94] Ronald L. Rivest. "The RC5 Encryption Algorithm." In: *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*. 1994, pp. 86–96 (cit. on p. 22).

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (1978), pp. 120–126 (cit. on p. 5).

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. "How to Leak a Secret." In: *ASIACRYPT*. 2001, pp. 552–565 (cit. on p. 22).

[Sch89] Claus-Peter Schnorr. "Efficient Identification and Signatures for Smart Cards." In: *CRYPTO*. 1989, pp. 239–252 (cit. on p. 14).

[Sch93] Bruce Schneier. "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)." In: *Fast Software Encryption, Cambridge Security Workshop*. 1993, pp. 191–204 (cit. on p. 22).

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. 1994, pp. 124–134 (cit. on pp. 2, 6).

[SK96] Bruce Schneier and John Kelsey. "Unbalanced Feistel Networks and Block Cipher Design." In: *Fast Software Encryption, Third International Workshop*. 1996, pp. 121–144 (cit. on p. 22).

[SM10] Tomoyasu Suzaki and Kazuhiko Minematsu. "Improving the Generalized Feistel." In: *FSE*. 2010, pp. 19–39 (cit. on p. 39).

[ST13] National Institute of Standards and US Department of Commerce Technology NIST. "FIPS publication 186-4: Digital Signature Standard (DSS)." In: *Federal Information Processing Standards* (2013) (cit. on pp. 5, 6).

[ST14] William Stallings and Mohit P Tahiliani. *Cryptography and network security: principles and practice*. Vol. 6. Pearson London, 2014 (cit. on p. 23).

[ST77] National Institute of Standards and US Department of Commerce Technology NIST. "FIPS publication 46: Data encryption standard (DES)." In: *Federal Information Processing Standards* (1977) (cit. on pp. 22, 23).

[ST93] National Institute of Standards and US Department of Commerce Technology NIST. "FIPS publication 186: Digital Signature Standard (DSS)." In: *Federal Information Processing Standards* (1993) (cit. on p. 5).

[ST99] National Institute of Standards and US Department of Commerce Technology NIST. "FIPS publication 46-3: Data encryption standard (DES)." In: *Federal Information Processing Standards* (1999) (cit. on p. 22).

[Unr12] Dominique Unruh. "Quantum Proofs of Knowledge." In: *EUROCRYPT*. 2012, pp. 135–152 (cit. on p. 17).

[Unr15] Dominique Unruh. "Non-Interactive Zero-Knowledge Proofs in the Quantum Random Oracle Model." In: *EUROCRYPT*. 2015, pp. 755–784 (cit. on p. 17).

[Unr16] Dominique Unruh. "Computationally Binding Quantum Commitments." In: *EUROCRYPT*. 2016, pp. 497–527 (cit. on p. 17).

[Vér96] Pascal Véron. "Improved identification schemes based on error-correcting codes." In: *Appl. Algebra Eng. Commun. Comput.* 8.1 (1996), pp. 57–69 (cit. on p. 9).