Roman Walch BSc

Design and Implementation of a Picnic Coprocessor

Master's thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Dipl.Ing. Daniel Kales BSc
Dipl.Ing. Mario Werner BSc
Univ.-Prof. Dipl-Ing. Dr.techn. Christian Rechberger

Institute for Applied Information Processing and Communications
Faculty of Computer Science and Biomedical Engineering

Graz, May 2019

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

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum	Unterschrift

Abstract

Digital signatures play an important role in today's society. We use them to secure internet connections, in online banking, and even to create legally binding signatures. However, existing signature schemes face the threat of quantum computers. Once a sufficiently powerful quantum computer is developed, we can use Shor's algorithm for factoring and discrete-logarithm computations to break most of the digital signature schemes we use today. Therefore, the National Institute of Standards and Technology (NIST) has started a standardization project to find new signature schemes which won't be broken by quantum computers. In January 2019, the second round candidates of the post-quantum project were announced. At this point in the project, one of the factors taken into consideration by NIST to evaluate the suitability of a candidate is the availability of efficient hardware implementations.

One of the advancing designs is Picnic, an algorithm based on transformed zero-knowledge proofs and symmetric-key primitives. In this thesis, we present the first FPGA-based hardware implementation of Picnic. We describe how we are able to efficiently implement LowMC, a block cipher used as a one-way function in Picnic, in VHDL despite the large number of constants required for computation. We then extend this design to implement the full Picnic algorithm in VHDL and synthesize it on a Kintex-7 FPGA board. Furthermore, we explain how we can connect our design to a PC via the PCIe interface and how we can sign messages and verify signatures using a C-Library we developed. We also compare our Picnic coprocessor to implementations of other signature schemes and show how we are able to speed up signing and verification in comparison to state-of-the-art Picnic software implementations.

Kurzfassung

Digitale Signaturen spielen eine wichtige Rolle in der heutigen Gesellschaft. Wir verwenden sie um Internetverbindungen abzusichern, beim Onlinebanking, und in vielen Ländern ist es sogar möglich, gesetzlich bindende Unterschriften zu erzeugen. Jedoch sind aktuelle Signaturverfahren durch Quantencomputer gefährdet. Sollte ein ausreichend starker Quantencomputer entwickelt werden, können wir Shor's Algorithmus zum Faktorisieren und Lösen von diskreten Logarithmen verwenden, um die meisten heutzutage verwendeten digitalen Signaturverfahren zu brechen. Darum hat das Nationale Institut für Standards und Technologie (NIST) ein Projekt gestartet, um neue Signaturverfahren zu finden, welche nicht durch einen Quantencomputer gebrochen werden können. Im Jänner 2019 wurden die Kandidaten der zweiten Runde des Projekts verkündet. Zu diesem Zeitpunkt im Projekt berücksichtigt NIST auch, ob sich die Kandidaten effizient in Hardware implementieren lassen. Picnic, ein Signaturverfahren, welches auf zero-knowledge Beweisen und symmetrischen Verschlüsselungsverfahren aufbaut, ist einer der verbleibenden Zweitrundenkandidaten. In dieser Arbeit präsentieren wir die erste Hardwareimplementierung von Picnic. Wir beschreiben, wie wir LowMC, eine in Picnic verwendete Blockverschlüsselung, effizient in VHDL implementieren können, obwohl diese eine große Anzahl an Konstanten zur Berechnung benötigt. Wir erweitern dieses Design, um den vollständigen Picnic Algorithmus zu implementieren and synthetisieren diesen dann auf ein Kintex-7 FPGA Board. Des Weiteren erklären wir, wie wir unsere Implementierung mit einem PC über das PCIe Interface verbinden können und wie wir Signaturen über ein C-Programm erzeugen und verifizieren können. Wir vergleichen unseren Coprozessor mit Implementierungen von anderen Signaturverfahren und zeigen, dass der entwickelte Coprozessor soagar schnellere Laufzeiten für das Signieren und Verifizieren von Picnic Signaturen hat, als die besten existierenden Softwareimplementierungen.

Contents

Abstract iii

Kurzfassung iii

1 Introduction 1
- 1.1 Digital Signatures . . . 1
- 1.2 Post-Quantum Signatures . . . 2
- 1.3 Goal and Contribution . . . 3
- 1.4 Outline . . . 3

2 The Picnic Signature Scheme 4
- 2.1 Formal Definition of Digital Signatures . . . 4
- 2.2 Sigma Protocols . . . 5
- 2.3 Fiat-Shamir Transformation . . . 5
  - 2.3.1 FS Signatures - Schnorr . . . 6
- 2.4 ZKBoo . . . 8
- 2.5 LowMC . . . 10
  - 2.5.1 Sbox Layer . . . 10
  - 2.5.2 Linear Layer . . . 11
  - 2.5.3 Constant Addition . . . 11
  - 2.5.4 Key Addition . . . 11
  - 2.5.5 Instances used in Picnic . . . 12
- 2.6 Keccak . . . 12
- 2.7 Picnic Instances and Parameters . . . 14
- 2.8 Signature Creation . . . 16
  - 2.8.1 Seed Generation . . . 16
  - 2.8.2 Circuit Decomposition . . . 16
  - 2.8.3 Challenge Generation (H3) . . . 19
  - 2.8.4 Signature Serialization . . . 20

2.9 Signature Verification . . . 21
2.9.1 Signature Deserialization . . . 21
2.9.2 Recomputing the Circuit Decomposition . . . 21
2.9.3 Recomputing the Challenge . . . 22

3 Optimizations of LowMC's Linear Layer 23
3.1 Motivation . . . 23
3.2 Optimized Key Matrices and Round Constants . . . 24
3.3 Optimized Linear Layer . . . 26
3.4 Optimized VHDL Implementation of LowMC . . . 28
3.5 Optimized Hardware Utilization . . . 28

4 The Picnic VHDL Design 32
4.1 Submodule Implementation . . . 32
4.1.1 Keccak . . . 32
4.1.2 Multi-Party Computation of LowMC . . . 33
4.1.3 Tapes and Commitments . . . 35
4.1.4 Seed Generation . . . 36
4.1.5 H3 - Challenge Generation . . . 36
4.1.6 BRAM . . . 38
4.1.7 Serialization and Deserialization of the Signature . . . 39
4.2 High-Level Design . . . 40
4.3 AXI4-Stream Interface . . . 41
4.4 Communication Protocol . . . 45
4.4.1 Instructions, Headers and Status Codes . . . 45
4.4.2 Setting the Keys . . . 47
4.4.3 Message Signing . . . 48
4.4.4 Signature Verification . . . 48

5 The Picnic-PCIe Coprocessor 50
5.1 Hardware and Software . . . 50
5.2 PCIe/DMA Subsystem . . . 50
5.3 High-Level Coprocessor Design . . . 51
5.4 Software Access to the Coprocessor . . . 52
5.4.1 Driver Setup . . . 54
5.4.2 C-Library . . . 54

6 Practical Evaluation **57**
6.1 Hardware Utilization 57
6.1.1 Picnic Submodules 57
6.1.2 Picnic Coprocessors 60
6.2 Benchmarks 61
6.2.1 Benchmark Platforms 62
6.2.2 Timing Benchmarks 62
6.3 Comparison to other Signature Schemes 64

7 Conclusion and Future Work **66**
7.1 Conclusion 66
7.2 Future Work 67

Bibliography **68**

List of Figures

2.1 Σ-protocol and FS transformation. 6
2.2 Schnorr signature. 7
2.3 One round of encryption with LowMC. 11
2.4 The sponge structure of Keccak. 13

3.1 One encryption round before and after the splitting of the roundkey addition. 25
3.2 LowMC implementation without optimizations. 29
3.3 LowMC implementation with optimizations. 30

4.1 Interface of the Keccak core. 33
4.2 High-level design of Picnic signing (left) and verification (right) with parallel *Commits*/*Tapes*. 42
4.3 High-level design of Picnic signing (left) and verification (right). 43
4.4 Interface of the Picnic VHDL implementation. 44
4.5 Timing of an AXI4-Stream transaction. 45
4.6 Instruction and status format. 46
4.7 Segment header format. 46
4.8 Protocol of setting the public key (left) and private key (right). 47
4.9 Protocol of signing a message. 48
4.10 Protocol of verifying a signature. 49

5.1 High-level design of the Picnic-PCIe coprocessors. 52
5.2 High-level design of the low-frequency Picnic-PCIe coprocessors. 53

List of Tables

2.1 LowMC instances used in Picnic. 12
2.2 Parameters of different Picnic instances. 15
2.3 Key and signature sizes of different Picnic instances in bytes. 15
2.4 Picnic-FS signature sizes in bytes. 21

3.1 Reduction of LowMC constants. 28
3.2 LUTs of LowMC with and without optimizations. 31

4.1 Different configurations of the Picnic implementations. . . . 41
4.2 Segment Type Encoding 47

6.1 Hardware Utilization for different parts of the L1 design. . . . 58
6.2 Hardware Utilization for different parts of the L5 design. . . . 59
6.3 Hardware Utilization for different parts of the coprocessors. . 60
6.4 Hardware Utilization for the complete coprocessors. 61
6.5 Different Benchmark Platforms. 62
6.6 Runtime comparison of the coprocessors on benchmark platform A. 63
6.7 Runtime comparison of optimized software implementations. 63
6.8 Comparison of different FPGA signature scheme implementations. 65

1 Introduction

In this Chapter, we give an introduction to digital signatures and state the goal of this thesis. We then report our contribution and lay out the outline of the present document.

1.1 Digital Signatures

Digital data and online communication increasingly gained importance over the last decades, and without any protection, malicious parties can easily impersonate users or modify data in today's online ecosystem. Digital signatures protect the correct origin and integrity of data and can, therefore, be used to detect such malicious behavior.

Digital signatures are the electronic counterpart of the classical, handwritten signatures, and they are an essential part in securing internet connections using Transport Layer Security (TLS) and for Public Key Infrastructures (PKI). In some countries, digital signatures can even be used for authentication in government services and to create legally binding alternatives to the handwritten signatures [21].

The signature algorithms used today are mainly based on classical asymmetric cryptography, such as RSA and elliptic-curve based signature schemes. In these cryptosystems, the signer owns a private key, which he can use to create a signature of digital data. A verifier can then use the corresponding, publicly available, public key to verify if the signature was created with the correct private key and if the signed data has not been altered.

The security of these asymmetric cryptographic signature schemes relies on two hardness assumptions, the integer factorization problem (IFP) and the

discrete logarithm problem (DLP). We do not know of an efficient algorithm to solve these problems on classical computers. However, once developed, quantum computers, new types of computers which use phenomena of quantum physics to run algorithms, can be used to run already known quantum algorithms to break both, the IFP and the DLP.

1.2 Post-Quantum Signatures

In 1994, Peter Shor published his quantum algorithm [35] for factoring and discrete-logarithm computations in polynomial-time. This algorithm would be capable of breaking most of the asymmetric cryptography schemes which we use today, once a sufficiently powerful quantum computer is developed [35]. This also includes the cryptographic schemes we use to create and verify digital signatures, like DSA, its elliptic curve variant ECDSA, and RSA.

It is still unclear if or when such sufficiently powerful quantum computers will be developed. However, it is crucial to find new post-quantum secure asymmetric cryptographic schemes now to ensure a smooth transition from classical algorithms to post-quantum schemes before the classical ones become fully broken. As a consequence, the National Institute of Standards and Technology (NIST) started a project to find, review, and standardize new algorithms for public key encryption, key exchange protocols, and digital signatures [27]. In January 2019, NIST announced the second round candidates of the standardization project [1]. Advancing submissions for digital signatures include schemes based on lattices (qTESLA [3], CRYSTALS-Dilithium [20], etc.), schemes based on multivariate systems of quadratic equations (MQDSS [15], etc.), hash-based schemes (SPHINCS [7]), and schemes based on transformed zero-knowledge proofs and symmetric-key primitives (Picnic [11]).

According to the tentative timeline of NIST [27], the third round candidates will be decided until 2021 and the competition will end before 2024.

1.3 Goal and Contribution

In the second round of the NIST project [1], hardware implementation of all the remaining candidates are required. In this thesis, we aim to implement Picnic in VHDL and design and implement an FPGA-based coprocessor which is capable of creating and verifying Picnic signatures via the PCIe interface. We compare the performance of our developed coprocessor to state-of-the-art software implementations of Picnic and also compare it to hardware implementations of other signature schemes.

In this thesis, we first created an area-optimized VHDL design of LowMC and used it to create VHDL implementations for two specific instances of Picnic, namely Picnic-L1-FS and Picnic-L5-FS. We then synthesized the design onto an FPGA-board that can be connected to a PC via the PCIe interface, and we created a C-Library to sign messages and verify signatures using our coprocessor. We evaluated the hardware utilization of our design and compared the performance to software implementations of Picnic and to other signature schemes, including an implementation of SPHINCS-256 [5], another post-quantum signature scheme. We show that using our FPGA implementations, signing takes 0.25 ms for Picnic-L1-FS and 1.24 ms for Picnic-L5-FS, which is about four times faster than existing, optimized software implementations.

1.4 Outline

In Chapter 2 of this thesis, we introduce the Picnic signature scheme and its building blocks. In Chapter 3, we explain some optimizations for LowMC's linear layer and how we implement an area-optimized VHDL design of LowMC. In Chapter 4, we describe our Picnic VHDL implementation. In Chapter 5, we explain how we extend this implementation to an FPGA-based PCIe coprocessor and how we can use it from a C-program. In Chapter 6, we evaluate the hardware utilization and performance of our coprocessor, before we conclude the thesis and discuss possible future work in Chapter 7.

2 The Picnic Signature Scheme

In this Chapter, we give an introduction to the PICNIC signature scheme and its building blocks. We first describe digital signatures, Σ-protocols and the Fiat-Shamir transformation, then we present ZKBoo and how to build a signature scheme based on transformed Σ-protocols. Then we describe LowMC and KECCAK, two cryptographic primitives used in PICNIC, before we explain how signatures are created and verified using the PICNIC algorithm.

2.1 Formal Definition of Digital Signatures

Digital Signatures are cryptographic schemes to authenticate digital information. The signer can use a secret key (sk) to sign a message (M); the verifier can use the corresponding public key (sk) to verify the correctness of the signature. A valid signature provides the following three security principles:

- Authenticity
  The signature assures that the signer is who he claims to be.
- Integrity
  The signature assures that the signed message has not been altered.
- Non-repudiation
  The signer can not deny signing the message.

A typical digital signature scheme consist of three algorithms:

- A key generation algorithm that outputs a key pair (sk, pk) given a security parameter κ:

$$(sk, pk) \leftarrow KeyGen(1^\kappa) \tag{2.1}$$

- A signing algorithm that outputs the signature σ of a message M given the secret key sk:

$$\sigma \leftarrow Sign(M, sk) \tag{2.2}$$

- A verification algorithm that verifies the signature σ of a message M given a public key pk and outputs either valid ($\top$) or invalid ($\bot$):

$$\top / \bot \leftarrow Verify(M, \sigma, pk) \tag{2.3}$$

2.2 Sigma Protocols

A sigma protocol (Σ-protocol) is a protocol for proof of knowledge of the secret input x for the public relation $\phi(x) = y$ and a public known value y. The protocol consist of the following three steps:

- The prover makes a commitment r and sends it to the verifier
- The verifier chooses a challenge c and sends it to the prover
- The prover assembles a proof s based on x, r and c and sends it to the verifier

The verifier then checks if the verification relation $\phi_{\text{verify}}(y, r, c, s) = 1$ holds and accepts or rejects the proof accordingly.

2.3 Fiat-Shamir Transformation

With the Fiat-Shamir (FS) transformation, it is possible to build a proof of knowledge protocol by making a Σ-protocol non-interactive. Instead of relying on the verifier to choose a challenge, the prover generates it as the

result of a cryptographically secure hash algorithm $c \leftarrow H(r)$ calculated on the commitment r. This method creates a non-interactive protocol that is secure in the random oracle model [23].

Figure 2.1 depicts a Σ-protocol and the FS transformation.

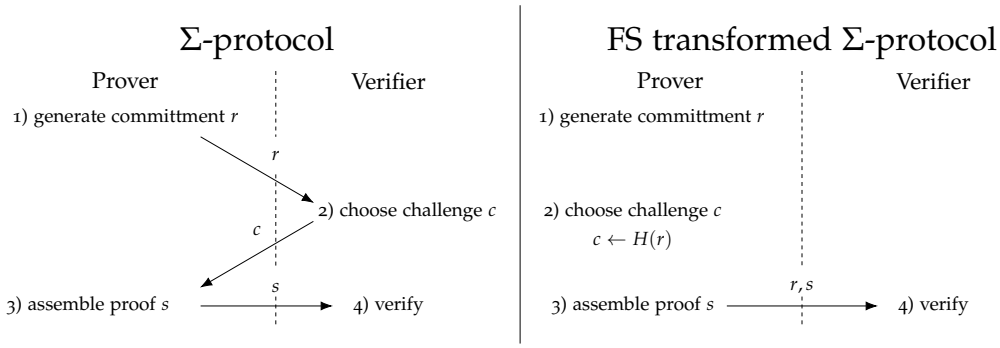


Figure 2.1: Σ-protocol and FS transformation.

2.3.1 FS Signatures - Schnorr

We can use FS-transformed Σ-protocols to build signature schemes which are secure in the random oracle model:

- The signer creates a commitment r.
- The signer uses the random oracle to create the challenge c based on the message m and the commitment r.

$$c \leftarrow H(m, r) \tag{2.4}$$

- The signer assembles the signature s based on the commitment r, the private key x and the challenge c.

$$s = \text{proof}(r, c, x) \tag{2.5}$$

- The verifier can recompute the commitment r' based on the signature s, the challenge c and the public key y and verify if it creates the correct challenge using the random oracle.

$$r' = \text{verify}(s, c, y) \tag{2.6}$$

$$c \stackrel{?}{=} H(m, r') \tag{2.7}$$

One example of building a signature based on FS-transformed Σ-protocols is the Schnorr signature. It is built by applying the FS transformation to the Schnorr identification scheme [32, 33, 34].

The Schnorr signature is defined as shown in Figure 2.2, where g is a generator of a cyclic group $\mathbb{G}$ with prime order q, and $H : \{0,1\}^* \times \mathbb{G} \to \mathbb{Z}_q$ is a hash function. The private key of Schnorr's signature is $x \leftarrow \mathbb{Z}_q \setminus \{0\}$, the public key is $y \leftarrow g^x$ and $m \in \{0,1\}^*$ is the message to be signed.

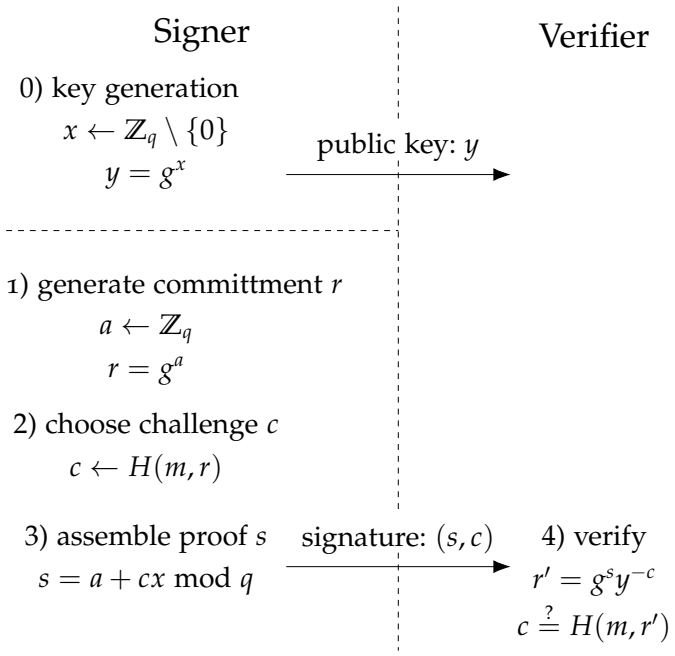


Figure 2.2: Schnorr signature.

The Schnorr signature is quite efficient and straightforward to calculate; however, it relies on the discrete logarithm problem, which will be broken by a sufficiently powerful quantum computer.

2.4 ZKBoo

The PICNIC signature scheme is based on ZKBoo, a zero-knowledge proof system, and its optimized version ZKB++. ZKBoo builds on a Σ-protocol that does not rely on mathematical problems, such as the discrete logarithm problem. Therefore, contrary to Schnorr's identification scheme, it will not be broken by a sufficiently powerful quantum computer. PICNIC uses the FS transformation in the random oracle model to make ZKBoo non-interactive.

ZKBoo improves on the MPC-in-the-head paradigm introduced by Ishai et al. [24], which describes how to build zero-knowledge proofs based on multiparty computation protocols (MPC). As an example, we use MPC to compute the relation $y = \phi(x)$, where y is public, and x is a private key. Every player of the protocol has a share of the key x and the prover simulates the multiparty computation of all players and commits to the view of all players consisting of the key share, a communication transcript, and the output share. The verifier then selects (corrupts) a random subset of the players for which the input shares get published. The verifier then can recompute the multiparty computation for the corrupted players and verify the commitments and therefore gets some assurance that the prover really knows the secret key x and performed the computation correctly. Repeating this process for different random key shares gives the verifier a higher assurance [24].

ZKBoo replaces the MPC computation with circuit decompositions which relaxes the properties of MPC protocols but still leads to secure and more efficient proofs. In ZKBoo, the number of players is fixed to three and the circuit to calculate the relation $y = \phi(x)$ is decomposed in the following way [11]:

- A `Share` function splits the secret key into three key shares
- An `Output` function that produces an output share based on a key share and some randomness
- A `Reconstruct` function that recomputes the output from the three output shares

The decomposition of the circuit has to satisfy correctness and 2-privacy [11]:

- *Correctness*
  The reconstruction of all output shares y_i must always be the result of the original relation $y = \phi(x)$.

$$\begin{aligned} \forall \phi, \forall x : Pr[x_0, x_1, x_2 \leftarrow \texttt{Share}(x), \\ y_0, y_1, y_2 \leftarrow \texttt{Output}(x_0, x_1, x_2): \\ \phi(x) = \texttt{Reconstruct}(y_0, y_1, y_2)] = 1 \end{aligned} \tag{2.8}$$

- 2-*Privacy*
  It should not be possible to reveal information about the private key x by publishing any information on any two players.

To create a proof, the prover runs ϕ T times using the circuit decomposition. For each run, the prover commits to the view of each player consisting of the input share, a transcript, and the output share. After all T runs, the prover sends all the output shares and commitments for each run and player to a random oracle to calculate the challenge based on the Fiat-Shamir transformation heuristic. The challenge tells the prover which two players should be corrupted per run and therefore which views should be published. Since the decomposition satisfies the 2-privacy property, no information is leaked on the secret key by publishing the views of two players [11].

The verifier then recalculates the two opened views and checks the following properties:

- The opened views were calculated correctly
- The challenge was computed correctly
- The three output shares of each run can be reconstructed to y

Each run gives some assurance that the prover really knows the secret key x, therefore increasing the number of runs T decreases the probability that the prover can cheat without the verifier catching him at least once. Since the prover can cheat for 2 of the 3 possible challenges per run, we can calculate the probability for him to cheat without getting caught with formula 2.9, where S is the security level in bits [11].

$$2^{-S} = \left(\frac{2}{3}\right)^T \tag{2.9}$$

In PICNIC, we use the relation $C = \text{LowMC}(p, k)$ for the circuit decomposition, where k is a private key and (C, p) a corresponding plain-/ciphertext pair which is known publicly. For the challenge generation, according to the FS transformation heuristic, SHAKE, a cryptographic hash algorithm based on KECCAK, is used to instantiate the random oracle [11].

2.5 LowMC

This Section is based on a description of LowMC by Walch [37].

LowMC is a block cipher designed to reduce the number of AND gates needed for symmetric encryption. The cipher is based on a substitution-permutation network (SPN), and the parameters block size (n), key size (k), number of Sboxes (m) and the number of rounds (r) are parameterizable according to the LowMC v3 round formula [2]. This formula considers all recent attacks [30, 18, 17] to calculate the lowest number of rounds necessary to provide secure encryption for the given parameter set. The script for determining the number of rounds for a given set of LowMC parameters can be found in the official Github repository [26].

A LowMC encryption starts with an initial whitening by XORing the first roundkey to the plaintext, followed by r rounds. As depicted in Figure 2.3, one round consists of four steps [2]:

- SBOXLAYER
- LINEARLAYER
- CONSTANTADDITION
- KEYADDITION

2.5.1 Sbox Layer

In the SBOXLAYER m 3-bit Sboxes are applied to the least significant $s = 3 \cdot m$ bits of the state. The remaining bits of the state are not affected by the

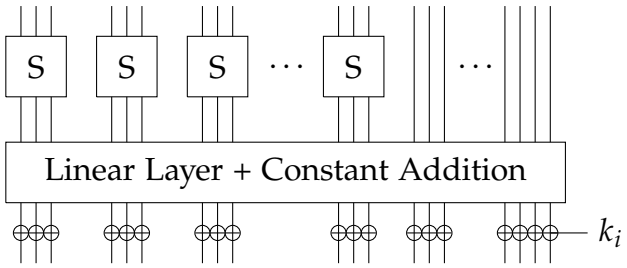


Figure 2.3: One round of encryption with LowMC. (Modified from [2].)

SboxLayer. The following equation specifies one Sbox and shows that only 3 AND gates are required [2]:

$$S(a,b,c) = \Big(a \oplus (b \wedge c), a \oplus b \oplus (a \wedge c), a \oplus b \oplus c \oplus (a \wedge b)\Big). \tag{2.10}$$

2.5.2 Linear Layer

In the LinearLayer the state is multiplied with a pseudorandomly generated $n \times n$ binary matrix $Lmatrix[r]$ in GF(2), where r is the current round. The matrices are chosen pseudorandomly from the set of invertible $n \times n$ matrices during the instantiation of LowMC [2].

2.5.3 Constant Addition

During the ConstantAddition the vector $roundconstant[r]$ of length n is XORed to the state, where r describes the current round. The vectors are chosen pseudorandomly during the instantiation of LowMC [2].

2.5.4 Key Addition

During KeyAddition, the roundkey of the current round is XORed to the state. All roundkeys are generated as a result of the GF(2) multiplication of the master key with the $n \times k$ binary matrix $Kmatrix[r]$, where r is the

current round. The matrices are chosen pseudorandomly from the set of $n \times k$ matrices during the instantiation of LowMC [2].

2.5.5 Instances used in Picnic

Table 2.1 shows the LowMC instances that are used in PICNIC.

Table 2.1: LowMC instances used in PICNIC.

Sec. Lvl.	blocksize n	keysize k	sboxes m	rounds r	Data complexity	# of ANDs	ANDs per bit
L1	128	128	10	20	2^1	600	4.69
L3	192	192	10	30	2^1	900	4.69
L5	256	256	10	38	2^1	1140	4.45

2.6 Keccak

KECCAK is a family of hash functions and the winner of the Secure Hash Algorithm 3 (SHA-3) competition, for which it got standardized by NIST in their publication FIPS 202 [28] in August 2015.

The core design of KECCAK is a sponge structure which can be seen in Figure 2.4. KECCAK's internal state consists of b bits which are initialized with zeros and separated into a bitrate of r bits and a capacity of c bits. The hash construction is divided into two phases, the absorbing and squeezing phases [8].

During the absorbing phase, the input is first padded and divided into blocks of r bits. Then for every input block, the block is XORed to the bitrate part of the state and the whole state is transformed using the KECCAK-f function [8].

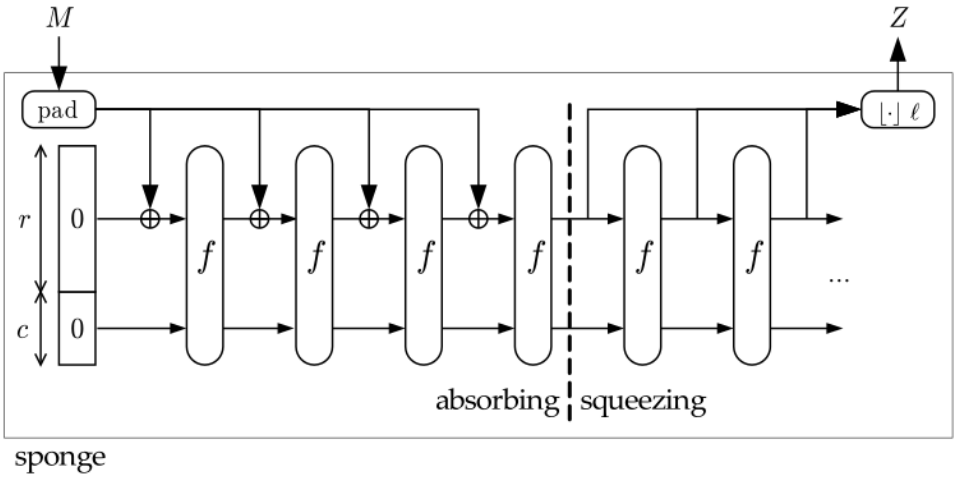


Figure 2.4: The sponge structure of KECCAK. Picture taken from `https://keccak.team/sponge_duplex.html` [9].

In the squeezing phase, the bitrate part of the state is used to construct the hash. The hash is first initialized as an empty string, before the bitrate part of the state is appended to the hash. Then the entire state is transformed using the KECCAK-f function and the bitrate part of the state is attached to the hash. This appending and transforming is repeated until the hash has the desired length. With this technique, it is possible to construct a key derivation function which produces a variable number of output bits [8].

The security of KECCAK is defined by the capacity c. For a long enough output, the sponge construction is provably secure against generic attacks up to a complexity of $2^{c/2}$ [8].

KECCAK is designed to be very flexible. The state size is fixed to $b \in \{200, 400, 800, 1600\}$ bit for all configurations, but the capacity c, and output length l can be freely chosen. PICNIC uses the following KECCAK configurations:

- SHAKE128: KECCAK[$r = 1344$, $c = 256$] with variable output length l
- SHAKE256: KECCAK[$r = 1088$, $c = 512$] with variable output length l

2.7 Picnic Instances and Parameters

In this Section, we give an overview of the different PICNIC instances and their parameter sets. For each of the three security levels $S \in \{128, 192, 256\}$, there exist two different PICNIC algorithms. One algorithm is based on the Fiat-Shamir transformation (FS) described in Section 2.3; the other is based on the Unruh (UR) transformation [36], another approach to make Σ-protocols non-interactive to build a signature scheme. In contrary to the FS transformation, which makes the resulting non-interactive Σ-protocol secure in the random oracle model, the UR transformation additionally provably secures the protocol in the quantum random oracle model (QROM) [10], where an adversary can query the random oracle in quantum superposition. By the time PICNIC was developed, the security of the FS transformation in the QROM was not well understood [10]; therefore, the UR transformation was used to build a post-quantum signature scheme. However, a recent paper by Don et al. [19] claims that the specific use of the FS transformation in PICNIC is also secure in the quantum random oracle model. Therefore, we do not need the more costly PICNIC variants which are based on the UR transformation and can rely on the FS transformation to build a post-quantum signature scheme.

Table 2.2 shows the parameters of all the different PICNIC versions. The expected security of the various instances corresponds to S bits against classical attacks and $S/2$ bits against quantum attacks. The parameter T describes how often the circuit decomposition of ZKB00 is performed to make it infeasible for the prover to cheat during signature creation [12].

Table 2.3 shows the different key and signature sizes for the PICNIC instances. For the instances based on the FS transformation, the actual signature size is based on the calculated challenge. Therefore, the expected signature size will be slightly smaller than the maximum value specified in the table [12].

In this thesis, we focus our implementation on the PICNIC instances with security levels $S \in \{128, 256\}$ based on the FS transformation, namely PICNIC-L1-FS and PICNIC-L5-FS.

Table 2.2: Parameters of different PICNIC instances.

			LowMC				Hash/KDF	
Parameter Set	S	T	n	k	m	r	Algorithm	l
PICNIC-L1-FS	128	219	128	128	10	20	SHAKE128	256
PICNIC-L1-UR	128	219	128	128	10	20	SHAKE128	256
PICNIC-L3-FS	192	329	192	192	10	30	SHAKE256	384
PICNIC-L3-UR	192	329	192	192	10	30	SHAKE256	384
PICNIC-L5-FS	256	438	256	256	10	38	SHAKE256	512
PICNIC-L5-UR	256	438	256	256	10	38	SHAKE256	512

Table 2.3: Key and signature sizes of different PICNIC instances in bytes.

Parameter Set	Public Key	Private Key	Signature
PICNIC-L1-FS	32	16	≤ 34000
PICNIC-L1-UR	32	16	53929
PICNIC-L3-FS	48	24	≤ 76740
PICNIC-L3-UR	48	24	121813
PICNIC-L5-FS	64	32	≤ 132824
PICNIC-L5-UR	64	32	209474

2.8 Signature Creation

In this Section, we describe the signature creation algorithm for PICNIC.

2.8.1 Seed Generation

The circuit decomposition of ZKBoo requires some randomness to be performed. To make signature creation deterministic, a seed is created for each of the three players of each of the T runs of the circuit decomposition. The players then can pseudorandomly generate the required randomness $rand_i$ based on these seeds [12].

These seeds are also pseudorandomly generated at the beginning of the algorithm by calling the defined key derivation function (KDF, SHAKE128/256). Each seed is S bit long; therefore the KDF produces $3 \cdot T \cdot S$ bits of output [12].

As equation 2.11 shows, the KDF is fed with the secret key sk, the message M which should be signed, and the public key $pk = (C, p)$.

$$\begin{aligned} &\text{Seed}(0)(0) \,||\, \text{Seed}(0)(1) \,||\, \text{Seed}(0)(2) \,|| \\ &... \,||\, \text{Seed}(t)(0) \,||\, \text{Seed}(t)(1) \,||\, \text{Seed}(t)(2) \,|| \,..\, || \qquad \leftarrow \text{KDF}(sk||M||pk||S) \\ &\text{Seed}(T-1)(0) \,||\, \text{Seed}(T-1)(1) \,||\, \text{Seed}(T-1)(2) \end{aligned} \tag{2.11}$$

2.8.2 Circuit Decomposition

The circuit decomposition of PICNIC consists of three players and it is repeated for T runs. During one run, the three players simulate a MPC-protocol to calculate a LowMC encryption. The secret key of the LowMC encryption is set to be the PICNIC secret key sk, the plaintext p which should be encrypted is part of the PICNIC public key pk. The secret key is split among the three players using additive secret sharing in GF(2^k) [12]:

$$sk_0 \oplus sk_1 \oplus sk_2 = sk \tag{2.12}$$

Each player then calculates his output share $C_i = \text{LowMC}_{\text{MPC}}(p, sk_i)$ as result of the LowMC encryption of the plaintext p and his key share sk_i according to the MPC rules:

- XOR gates can be calculated locally without communication between the players
- An XOR of a state with a constant is only calculated by one of the players (player 0)
- For every AND gate communication between the players and a random bit for each player is required. The result of $c = a \wedge b$ is calculated according to the equation 2.13, where a_i and b_i are the inputs of player i, r_i is player i's random input and c_i is players i's output share.

$$c_i = (a_i \wedge b_{(i+1)\%3}) \oplus (a_{(i+1)\%3} \wedge b_i) \oplus (a_i \wedge b_i) \oplus (r_i \wedge r_{(i+1)\%3}) \tag{2.13}$$

  Equation 2.13 ensures that the output bits c_i of the players can be combined to the original output bit c of the LowMC encryption without MPC:

$$c_0 \oplus c_1 \oplus c_2 = c = a \wedge b \tag{2.14}$$

If the computations were done correctly, the output shares C_i can be combined to the ciphertext part C of PICNIC's public key pk:

$$C_0 \oplus C_1 \oplus C_2 = C \tag{2.15}$$

During the computation of the circuit decomposition, randomness is required to calculate the key shares of the players and to calculate the AND gates during the $\text{LowMC}_{\text{MPC}}$ encryption. At the beginning of the signing process, a seed was already generated for each player of each run t. This seed is hashed first using the defined hash algorithm (SHAKE128/256) H and then fed to the KDF (SHAKE128/256), to produces the key share sk_i and randomness $rand_i$ for each player i. To satisfy equation 2.12, only the KDF of player 0 and player 1 produce a key share, the remaining share of player 2 is calculated according to equation 2.19. A LowMC encryption consists of $3 \cdot r \cdot s$ AND gates with $s = 3 \cdot m$, therefore each player requires $3 \cdot r \cdot s$ random bits for the encryption. Equations 2.16, 2.17, and 2.18 show the KDF

of each player of each run t; the constant $0x02$ is prepended to the seed for domain separation [12].

$$sk_0 \,||\, rand_0 \leftarrow \mathrm{KDF}\Big(H\Big(0x02 \,||\, \mathrm{Seed}(t)(0)\Big)\Big) \tag{2.16}$$

$$sk_1 \,||\, rand_1 \leftarrow \mathrm{KDF}\Big(H\Big(0x02 \,||\, \mathrm{Seed}(t)(1)\Big)\Big) \tag{2.17}$$

$$rand_2 \leftarrow \mathrm{KDF}\Big(H\Big(0x02 \,||\, \mathrm{Seed}(t)(2)\Big)\Big) \tag{2.18}$$

$$sk_2 \leftarrow sk_0 \oplus sk_1 \oplus sk \tag{2.19}$$

The result of the circuit decomposition is a $\mathrm{View}(t)(i)$ for each player i of each run t. This view consists of three parts [12]:

- An input share iShare:

$$\mathrm{View}(t)(i).\mathrm{iShare} \leftarrow sk_i \tag{2.20}$$

- A transcript for the MPC communication. This transcript consists of all output bits c_i of each of the $3 \cdot r \cdot s$ AND gates during encryption.
- An output share oShare:

$$\mathrm{View}(t)(i).\mathrm{oShare} \leftarrow C_i \tag{2.21}$$

The view is part of the PICNIC signature, and the size of the transcript depends on the number of AND gates in the decomposed circuit. Other relations $y = \phi(x)$ (f.e. $C = \mathtt{AES}(p, k)$) can be used as the circuit, however, using LowMC, a cipher designed to reduce the number of AND gates for secure encryption, reduces the size of the transcript and therefore also the size of a PICNIC signature [12].

Each player additionally commits to his view by hashing it together with a hash of the seed used for randomness calculation; the constants $0x00$ and $0x04$ are again added for domain separation [12]:

$$Com(t)(i) \leftarrow H\Big(0x00 \,||\, H\Big(0x04 \,||\, \mathrm{Seed}(t)(i)\Big) \,||\, \mathrm{View}(t)(i)\Big) \tag{2.22}$$

2.8.3 Challenge Generation (H3)

For the FS transformation of the Σ-protocol, a random oracle (RO) is used to generate the challenge based on the commitments. In PICNIC, the RO is instantiated using the hash algorithms SHAKE128/256. The hash function is fed with the output shares of each player of each run, as well as the corresponding commitments, the public key pk and the message M which should be signed. Equation 2.23 shows the hash function used for challenge generation; the constant $0x01$ is added for domain separation [12].

$$\begin{aligned} h \leftarrow H\Big(&0x01 \,||\, \text{View}(0)(0).\text{oShare} \,||\, \text{View}(0)(1).\text{oShare} \,||\, \\ &\text{View}(0)(2).\text{oShare} \,||\, \\ &... \,||\, \\ &\text{View}(T-1)(0).\text{oShare} \,||\, \text{View}(T-1)(1).\text{oShare} \,||\, \\ &\text{View}(T-1)(2).\text{oShare} \,||\, \\ &... \,||\, \\ &Com(0)(0) \,||\, Com(0)(1) \,||\, Com(0)(2) \,||\, \\ &... \,||\, \\ &Com(T-1)(0) \,||\, Com(T-1)(1) \,||\, Com(T-1)(2) \,||\, \\ &pk \,||\, M\Big) \end{aligned} \tag{2.23}$$

The challenge of PICNIC tells the signer which two of the three views of the players of each run should be opened. Therefore, the challenge should consist of T values $\in \{0,1,2\}$. The output h of the hash function in equation 2.23 produces a bitstream $\{0,1\}^l$ which has to be mapped to $\{0,1,2\}^T$. This is done in the following steps [12]:

- Compute h according to equation 2.23
- Inspect bit-pairs (h_0, h_1), (h_2, h_3), ... of the hash h
  If the pair is:
  - (0, 0), append $e_t = 0$ to the challenge e
  - (0, 1), append $e_t = 1$ to the challenge e

  - (1, 0), append $e_t = 2$ to the challenge e
  - (1, 1), discard the pair

  Return if e consists of T values
- If h is fully consumed, a new hash h is calculated: $h \leftarrow H(0x01 \,||\, h)$
- Repeat bit-pair inspection (step 2) on the new hash h

2.8.4 Signature Serialization

A PICNIC signature consists of the challenge as well as all values required to recompute and verify the two opened players of each round. The commitment of the third player also has to be part of the signature, since it is required to recompute the challenge during verification. Serialization of the signature consists of the following steps [12]:

- Pad the challenge e with zeros to the next number of bytes and add the result to the signature
- For each run t append values depending on the challenge e_t to the signature:
  - Append the commitment of the closed player: $C(t)(e_t + 2 \bmod 3)$
  - Append the padded transcript of the second opened player: $\text{View}(t)(e_t + 1 \bmod 3)$.transcript
  - Append the seed of the first opened player: $\text{Seed}(t)(e_t)$
  - Append the seed of the second opened player: $\text{Seed}(t)(e_t + 1 \bmod 3)$
  - If $e_t \neq 0$ append the key share of player 2: $\text{view}(t)(2)$.iShare

Since the key share of player 2 is not pseudorandomly generated based on a seed, it has to be appended to the signature if the view of player 2 is opened ($e_t \neq 0$). As a result, the signature size of PICNIC based on the FS transformation depends on the calculated challenge and is therefore, not constant. Table 2.4 shows the maximum, average, and standard derivation of the PICNIC signature sizes [12].

2.9 Signature Verification

In this Section, we describe the algorithm to verify signatures created by the FS versions of PICNIC.

2.9.1 Signature Deserialization

The signature size of the PICNIC algorithms based on the FS transformation is not constant and depends on the challenge. Therefore, the challenge has to be read first, and an expected signature length can be calculated based on the challenge. The signature can already be rejected if the expected signature size does not match with the size of the given signature [12].

If the lengths match, the rest of the signature is read, and for each run t, the values of the signature are assigned to the players based on the challenge e_t.

If the padding bits of the challenge and each transcript are not all equal to zero, the signature is rejected as well.

2.9.2 Recomputing the Circuit Decomposition

After the signature is parsed, the circuit decomposition of the two opened players can be recomputed. Similar to the circuit decomposition during the signing process described in Section 2.8.2, the key shares and the required randomness is computed first based on the seeds, then an MPC-protocol of a

Table 2.4: PICNIC-FS signature sizes in bytes.

Parameter Set	Max	Average	Std. Dev.
PICNIC-L1-FS	34000	32838	107
PICNIC-L3-FS	76740	74134	198
PICNIC-L5-FS	132824	128176	315

LowMC encryption is simulated before the view, and the seed is committed for each player [12].

The main difference between this recomputation and the circuit decomposition during the signing process is that we only simulate the LowMC encryption for the two opened players during the recomputation. According to equation 2.13, the first opened player only communicates with the second opened player during the AND gate calculation; therefore there is no problem for the recomputation. However, the second opened player would interact with the still closed player; thus this AND gate calculation cannot be simulated. For this reason the PICNIC algorithm memorizes the results of the AND gate calculation of each player for each run t in the transcript of the view and the transcript of the second opened player is added to the signature [12].

2.9.3 Recomputing the Challenge

After the circuit decomposition, the challenge can be recomputed as described in Section 2.8.3. For the recomputation, the output share and the commitment of all players are required. The shares and commitments of the opened players have been recomputed and the commitment of the closed player is part of the signature. The output share of the closed player can be recomputed by rewriting equation 2.15 [12]:

$$C_{(e_t+2)\%3} = C_{e_t} \oplus C_{(e_t+1)\%3} \oplus C \tag{2.24}$$

If the recomputed challenge is equal to the challenge of the given signature the signature is accepted, otherwise the signature will be rejected.

3 Optimizations of LowMC's Linear Layer

In this Chapter, we describe how we optimized an existing VHDL LowMC implementation. We first give a motivation as to why these optimizations are necessary, then we explain how the optimizations work. Then, we compare the VHDL designs with and without the optimizations before we analyze the hardware utilization of both designs.

3.1 Motivation

In a previous project by Walch [37], a LowMC coprocessor was implemented. However, due to the significant number of constants required by the matrices of a LowMC encryption, the hardware utilization of the LowMC design is quite big. In Picnic, we need multiple instances of LowMC within the design to efficiently compute the MPC simulation of a LowMC encryption during signing and verification. Without any further improvement of this LowMC implementation, it would not be possible to fit multiple instances of LowMC onto the targeted FPGA. Fortunately, Dinur et al. [16] released a new paper at the start of the project in which they describe how to reduce the size of the constants required by LowMC and therefore, how to minimize the significant hardware utilization of the VHDL implementation.

The optimizations of LowMC can be split into two major parts, which we describe in the following two Sections.

3.2 Optimized Key Matrices and Round Constants

Algorithm 1 describes the basic LowMC encryption procedure, where k_i is the roundkey of round i, C_i the round constant of round i, x_0 the plaintext and x_{r+1} the ciphertext. $x^{(0)}$ denotes to the non-linear part of x which is affected by the Sboxes S and $x^{(1)}$ the remaining linear part of x. $L_i(y)$ describes the matrix multiplication of y with the $Lmatrix$ of round i.

Algorithm 1 Basic LowMC encryption [16]

Input: x_0
Output: x_{r+1}
1: **begin**
2: $x_1 \leftarrow x_0 \oplus k_0$
3: **for** $i \in \{1, 2, ..., r\}$ **do**
4: $y_i \leftarrow S(x_i^{(0)})||x_i^{(1)}$
5: $x_{i+1} \leftarrow L_i(y_i) \oplus k_i \oplus C_i$
6: **end for**
7: **return** x_{r+1}
8: **end**

The first optimization aims to reduce the constants required to compute the roundkeys based on the master key. With a small modification to line 5 of Algorithm 1 we can calculate $k_i' = L_i^{-1} \cdot k_i$ and $C_i' = L_i^{-1} \cdot C_i$ and therefore get $x_{i+1} = L_i(y_i \oplus k_i' \oplus C_i')$ [16].

The expression $L_i^{-1} \cdot k_i$ can be further split int a non-linear part $(L_i^{-1} \cdot k_i)^{(0)}$ and a linear part $(L_i^{-1} \cdot k_i)^{(1)}$. The linear part is not affected by the Sbox and can, therefore, be calculated at the beginning of a round. Figure 3.1 illustrates the splitting of the roundkey addition. Furthermore, the addition of the linear part of the roundkey can be combined with the addition of the complete roundkey of round $i-1$. Applying this optimization recursively down to round 0 we get one addition of roundkey k_0' with n bits at the beginning of the algorithm, all the other roundkeys k_i' have a size of $s = 3 \cdot m$ bits. The same optimizations can be applied to the roundconstants C_i.

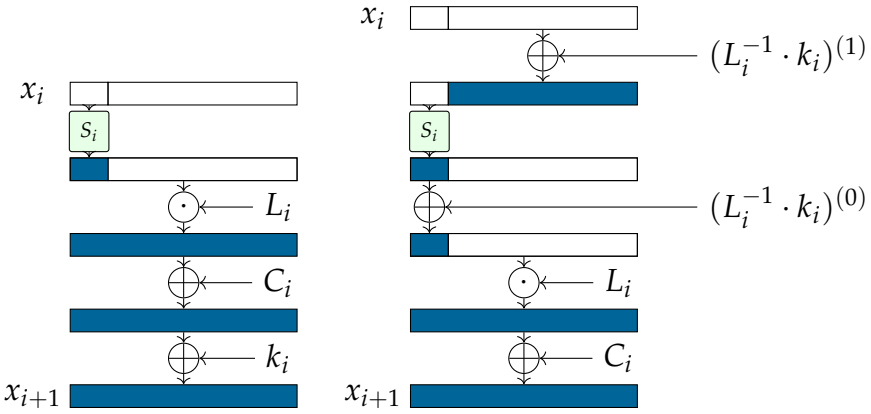


Figure 3.1: One encryption round before (left) and after (right) the splitting of the roundkey addition. The picture was taken from [16].

As a result, we obtain the modified LowMC encryption algorithm depicted in Algorithm 2 [16].

Algorithm 2 LowMC encryption with compressed roundkeys and constants [16]

Input: x_0
Output: x_{r+1}

begin
  $x_1 \leftarrow x_0 \oplus k'_0 \oplus C'_0$
  for $i \in \{1, 2, ..., r\}$ **do**
    $y_i \leftarrow (S(x_i^{(0)}) \oplus k'_i \oplus C'_i)||x_i^{(1)}$
    $x_{i+1} \leftarrow L_i(y_i)$
  end for
  return x_{r+1}
end

With the improvements described in this section we only have to calculate a s bit roundkey based on the k bit masterkey. Therefore, we can reduce the $n \times k$ bit roundkey matrices to $s \times k$ bit matrices. The new, optimized matrices $P_{N,i}$ can be calculated from the key matrices K_i according to the following formula, where $\overline{L_l^{-1}}$ is the inverse linear leayer matrix L_i with the

first s rows set to 0 [16]:

$$P_{N,i} = \sum_{j=i}^{r} (\prod_{l=i}^{j} \overline{L_l^{-1}}) \cdot K_j \tag{3.1}$$

The first roundkey k_0 still requires a $n \times k$ matrix P_L to be computed, this matrix can be calculated according to the following formula [16]:

$$P_L = K_0 + \sum_{j=1}^{r} (\prod_{l=1}^{j} \overline{L_l^{-1}}) \cdot K_j \tag{3.2}$$

The new roundkeys k_i are calculated as the result of the product of the new matrices with the master key, where $(P_N, i)^{0*}$ are the first s rows of the matrix (P_N, i) [16]:

$$k'_0 = P_L \cdot k \tag{3.3}$$

$$k'_i = (P_N, i)^{0*} \cdot k \tag{3.4}$$

The matrices $(P_N, i)^{0*}$ and P_L can be precomputed for every LowMC instance and they reduce the size of the constants from $(r+1) \cdot (n \cdot k)$ bits to $(n \cdot k) + r \cdot (n \cdot s)$ bits. Additionally, the roundconstants are reduced from $(r \cdot n)$ bits to $n + r \cdot s$ bits [16].

3.3 Optimized Linear Layer

The second optimization aims to reduce the linear layer of LowMC. The optimized algorithm is described by Algorithm 3. The algorithm to calculate the matrices L' and $\hat{L}$ from the original linear matrices can be found in the original paper by Dinur et al. [16].

In rounds $i \in \{1, 2, ..., r-1\}$, the linear layer can be described by the linear transformation

$$\left(\begin{array}{c} x_{i+1}^{(0)} \\ \hline x_{i+1}^{(1)} \end{array}\right) = \left[\begin{array}{c|c} L_i'^{00} & L_i'^{01} \\ \hline \hat{L}_i^{10} & \hat{L}_i^{11} \end{array}\right] \left(\begin{array}{c} y_i^{(0)} \\ \hline y_i^{(1)} \end{array}\right). \tag{3.5}$$

Algorithm 3 Fully optimizedLowMC encryption [16]

Input: x_0
Output: x_{r+1}
1: **begin**
2: $x_1 \leftarrow x_0 \oplus k'_0 \oplus C'_0$
3: **for** $i \in \{1, 2, ..., r-1\}$ **do**
4: $y_i \leftarrow (S(x_i^{(0)}) \oplus k'_i \oplus C'_i)||x_i^{(1)}$ ▷ Round i
5: $x_{i+1} \leftarrow L_i'^{0*}(y_i)||\hat{L}_i(y_i)$
6: **end for**
7: $y_r \leftarrow (S(x_r^{(0)}) \oplus k'_r \oplus C'_r)||x_r^{(1)}$ ▷ Round r
8: $x_{r+1} \leftarrow L'_r(y_r)$
9: **return** x_{r+1}
10: **end**

On first inspection of the optimized LowMC encryption, there are still r linear transformation of dimension $n \times n$ necessary. However, due to the linear algebra described in [16] the matrices $\hat{L}_i^{11}$ of size $(n-s) \times (n-s)$ will contain the identity matrices. Therefore, the constants required to compute the linear layer of LowMC are reduced from $r \cdot n^2$ bits to $r \cdot n^2 - (r-1) \cdot (n-s)^2$ bits.

Dinur et al. [16] remark, that if the original matrices L_i are selected uniformly at random from all invertible binary matrices of dimensions $n \times n$, the proposition that $\hat{L}_i^{11}$ consists of the identity matrix is not true in general. However, they show that the columns of the matrix $\hat{L}_i$ only have to be permuted to make the proposition right. Furthermore, they prove that, on average, only 3 pairs of columns have to be permuted. Therefore, to compute the linear transformation $\hat{L}_i(y_i)$ of Algorithm 3, the state y_i simply has to be permuted and the matrices $\hat{L}_i$ can be reduced to $n^2 - (n-s)^2$ bits in general.

3.4 Optimized VHDL Implementation of LowMC

Figure 3.2 shows the VHDL design of LowMC without the optimizations; Figure 3.3 shows the design with the optimizations. Without the optimizations, there is only one implementation for all the rounds, and the matrix multiplications affect the entire state. In the optimized implementation, there are 5 different matrix multiplication modules, each for a matrix with different dimensions. The Sbox layer, roundkey and constants of the new implementation only affect the first s bits of the state and, as described in the previous section, the state has to be permuted before the multiplication with the matrix $\hat{L}_i$ in order reduce the dimension of this matrix. The identity matrix part of $\hat{L}_i$ is simply implemented as one additional XOR per one bit of the state.

3.5 Optimized Hardware Utilization

Table 3.1 summarizes the size reduction of the LowMC constants due to the described optimizations. The impact of the optimizations depends on the LowMC instance, especially on the number of Sboxes. The fewer Sboxes per round, the more significant the effect of the optimizations. This can also be seen in Table 3.2 where the required lookup-tables (LUTs) of the LowMC VHDL implementation on a Xilinx Kintex7 FPGA KC705 Evaluation Kit (203800 LUTs available) are shown for three different LowMC instances.

Table 3.1: Reduction of LowMC constants.

Part	Before	Reduction
key matrices	$(r+1) \cdot n \cdot k$	$r \cdot n \cdot (k-s)$
round constants	$r \cdot n$	$r \cdot (n-s) - n$
linear layer	$r \cdot n^2$	$(r-1) \cdot (n-s)^2$

Instance nr. 1 of Table 3.2, which was the main focus of the previous design, has 25 Sboxes per round. Therefore, 75 out of the 128 bits of the state are

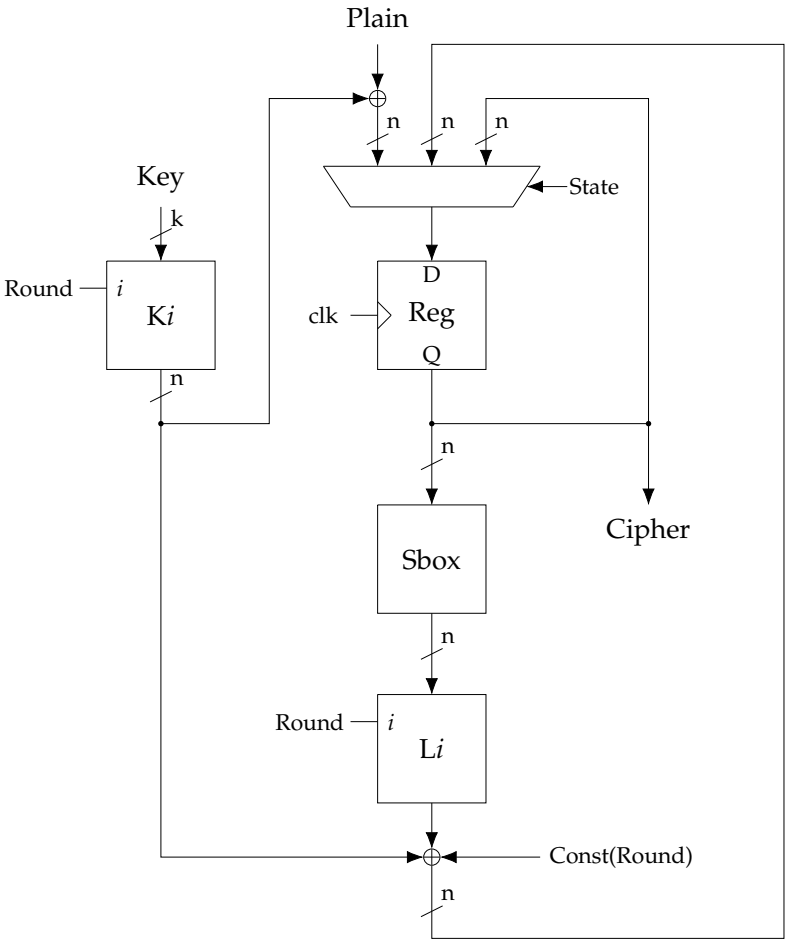


Figure 3.2: LowMC implementation without optimizations.

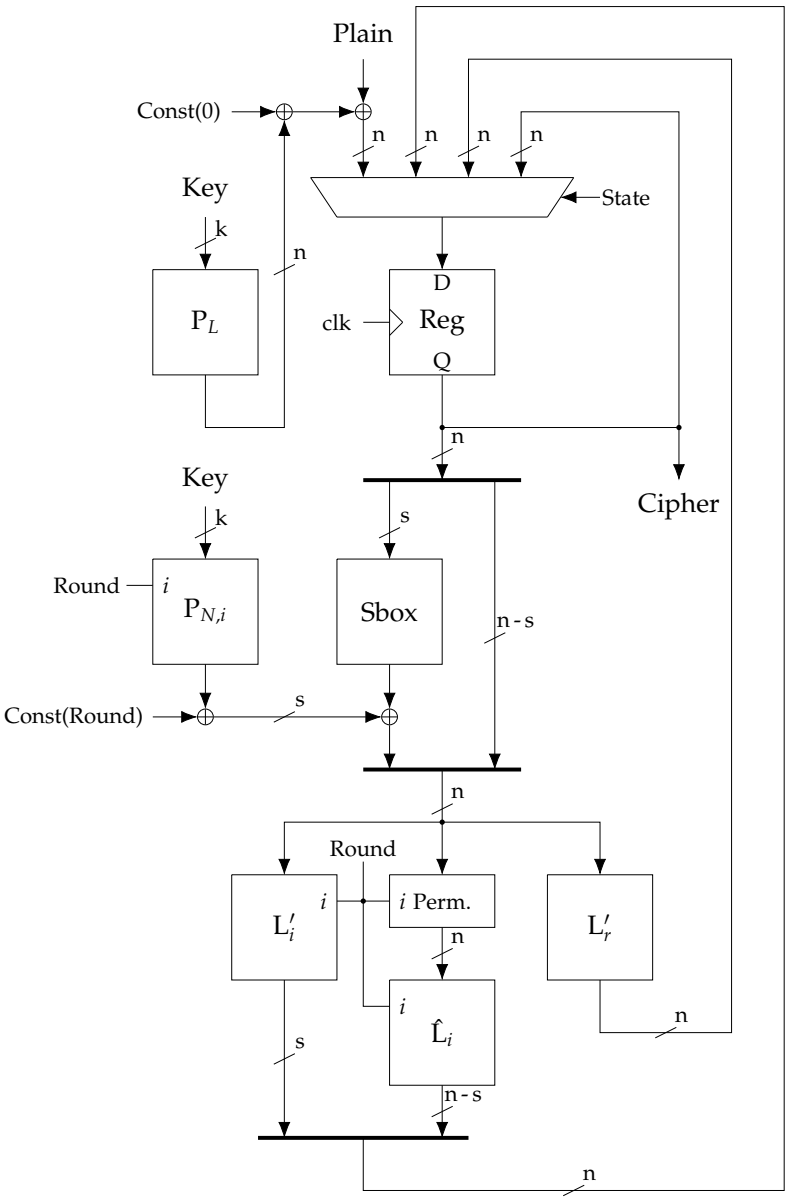


Figure 3.3: LowMC implementation with optimizations.

affected by the Sboxes, and the optimizations do not have a huge impact. In instances nr. 2 and 3, which are used in Picnic, the number of bits affected by the Sboxes is much lower. Therefore, the effect of the optimizations is much more significant. Instance nr. 2 only requires about a third of the LUTs required before, instance nr. 3 about a fifth. Without the optimizations, it would not even be possible to synthesize one LowMC instance of nr. 3, whereas for the Picnic implementation several of these are required.

Table 3.2: LUTs of LowMC with and without optimizations (203800 available).

	LowMC				without opt.		with opt.		Improv.
nr.	n	k	m	r	LUTs	% LUTs	LUTs	% LUTs	%
1	128	128	25	11	22200	10.89 %	18210	8.94 %	17.97 %
2	128	128	10	20	42395	20.80 %	13558	6.65 %	68.02 %
3	256	256	10	38	209348	102.72 %	44431	21.8 %	78.78 %

4 The Picnic VHDL Design

In this Chapter, we describe the VHDL design of our PICNIC implementations. We first describe all the implemented submodules before we depict the high-level design. We then specify the interface to access the PICNIC cores and explain the protocol to set the public and private keys, sign messages, and verify signatures.

4.1 Submodule Implementation

Our PICNIC cores are composed of several different submodules which implement the different parts of the PICNIC algorithm.

4.1.1 Keccak

In PICNIC, KECCAK is used in several different parts of the design, sometimes as a hash function and other times as a KDF with different parameters for the different PICNIC instances. Therefore, we wanted to implement a flexible KECCAK design that can be used for all the various applications in PICNIC.

The core design of our KECCAK implementation consists of a state machine and a register which contains the inner state. The bitrate r and the output length l are generic values that can be chosen during instantiation of the block. The interface of the core is depicted in Figure 4.1. The core has 2 different modes of operation, an *absorb* mode and a *squeeze* mode. Both modes apply the KECCAK-f state transformation to the inner state of the core, with the difference, that during *absorb* mode the input block is XORed to the state first. To be as flexible as possible, the handling of the input

block has to be implemented by other submodules; the KECCAK core design expects an already padded block of r bits. The *Init* input can be used to reset the inner state to zeros.

The KECCAK-f state transformation is implemented as combinatorial logic, that is applied 24 times to the state. Therefore, the design requires 24 cycles for one state transformation to be completed. The *Valid* signal indicates if the core is currently applying the state transformation, or if a valid hash is currently displayed at the *Hash* output, and can, therefore, be used for synchronization.

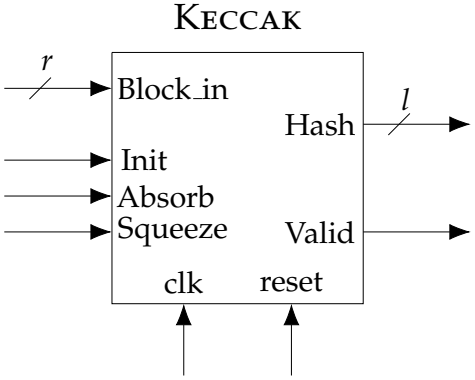


Figure 4.1: Interface of the KECCAK core.

4.1.2 Multi-Party Computation of LowMC

In PICNIC's circuit decomposition, three instances of LOWMC do an MPC encryption during signing, two instances do the encryption during verification. Since LOWMC requires a lot of constants for encryption, the hardware utilization of LOWMC is very high. Therefore, the MPC encryption of LOWMC will have the highest portion of the hardware utilization of PICNIC, even with the implemented optimizations described in Chapter 3.

As described in Section 2.8.2, the MPC protocol affects the AND gates of LOWMC. LOWMC only has AND gates in its Sboxes, therefore, to implement the MPC protocol only the Sbox layer of the optimized LOWMC implementation had to be modified. A naive approach would be to take three instances

of the optimized design, let them do the calculations in parallel and modify the Sbox layer to calculate the AND gates jointly. However, this approach requires three LowMC instances and the hardware utilization of PICNIC would exceed the available resources of the targeted FPGA.

Section 2.8.2 explains that the MPC protocol of PICNIC ensures, that the output bits of the MPC-AND gates can be combined to the original output bits of the LowMC encryption without MPC. This fact can be used to reduce the required LowMC instances for signing to two. In PICNIC, each player of the MPC protocol has to calculate his output share and a communication transcript based on his input share. The output share of the third player can always be recomputed by XORing the output share of the other players with the original ciphertext, which is part of the public key. For the transcript, we can precompute the original intermediate states of LowMC and use it to calculate the third input share of the Sboxes during the MPC encryption.

We implemented the MPC signing module to make use of these facts to only require two instances of LowMC. One instance is used to precompute the original encryption without MPC, and we store the inputs of the Sboxes for each round. This initial encryption can be done in parallel to the seed calculation of PICNIC and therefore won't cost any more clock cycles in the final design. The Sbox layer is implemented to calculate the MPC communication for all three players. During the MPC runs, both LowMC instances perform the encryption based on their share; the third input of the Sbox layer is calculated by XORing the inputs of the two other players with the precomputed input share.

The MPC verification module only calculates the MPC of two players; therefore no precomputation is required, and still, only two LowMC instances are used. Another difference to the signing module is that the AND gates are calculated differently for the verification. Therefore, a different implementation of the Sbox layer is used for the verification module.

We also implemented an MPC module which can be used for both signing and verification. This module is implemented to have three different modes of operations: precomputation, signing and verification. The different modes for signing and verification only differ in using the two different sbox layer implementations, the rest of the design, like all the matrix multiplications,

are shared between the modes of operation. Therefore, the overhead of combining signing and verification into one module is very low.

Remark 1. *We tried to reduce the number of* LowMC *instances required for MPC to only one by consecutively computing the matrix multiplications on the three different intermediate states. However, this resulted in too long critical paths after synthesis, and we had to reduce the clock frequency to less than* 60 *MHz. In combination with the increased number of clock cycles per round by consecutively computing the three shares, this would have resulted in a too great performance penalty. As part of future work, we could try to optimize this approach to implement a low-area version of* PICNIC.

4.1.3 Tapes and Commitments

The *Tape* module, which calculates the random keys and tapes required during the MPC encryption of LowMC, as well as the *Commit* module to calculate the commitments are both implemented as a wrapper for three KECCAK instances. Both modules first hash the seeds using one KECCAK instance per player according to equation 2.22 and equations 2.16, 2.17 and 2.18, before they reuse the same KECCAK instances to calculate either the finished random bits or final commitments. The modules take care of feeding the KECCAK instances with the correct input values, which includes padding and handling multiple absorbing and squeezing phases for PICNIC-L5-FS. The modules are activated by asserting a *start* input signal; a *finish* output signal is asserted after the calculations are finished.

Since both modules include three KECCAK instances, the calculations of the three players are done simultaneously. Therefore, the modules for PICNIC-L1-FS take 51 cycles to finish their calculation, 1 cycle to reset the internal KECCAK states and 25 cycles to create each of the two hashes. The PICNIC-L5-FS *Tape* module requires 75 cycles since one additional squeezing phase is necessary for the KDF in equations 2.16, 2.17 and 2.18, the *Commit* module requires 100 cycles, accounting for two additional absorbing phases during the second hash calculation of equation 2.22.

Remark 2. *Since both, Tape and Commit modules, each use three* KECCAK *instances on their own, we can calculate the random keys and tapes of* PICNIC*'s following*

MPC run $t+1$ in parallel to calculating the commitments of the current run t, saving $T \cdot 51 = 11169$ clock cycles for PICNIC-L1-FS *and $T \cdot 75 = 32850$ clock cycles for* PICNIC-L5-FS *in some high level designs. We also implemented a version in which we reuse the same three* KECCAK *instances for both Tape and Commit modules, but since one* KECCAK *instance requires very little hardware utilization in comparison to the MPC module, the reduced utilization was too small to justify the relatively high performance penalty.*

4.1.4 Seed Generation

The *Seeds* submodule calculates the random seeds at the beginning of PICNIC's signing process according to equation 2.11. It is implemented as a wrapper to an instance of the KECCAK submodule which first absorbs the inputs before producing $3 \cdot T \cdot S$ output bits. Since the seeds are calculated as an output of a SHAKE-KDF, the module has to take care of squeezing the KECCAK instance to get all the random seeds. The module is implemented as a state machine which can be controlled with a *start* and a *next* input signal and a *ready* output signal. Asserting the *start* signal activates the absorbing phase of the input, the *ready* output indicates that three valid seeds are asserted at the output of the module. We can request three new seeds by asserting the *next* signal.

For PICNIC-L1-FS the seed module requires 1732 clock cycles to finish calculating all seeds, for PICNIC-L5-FS 7904 cycles are needed.

Remark 3. *Since the instances* PICNIC-L1-FS *and* PICNIC-L5-FS *have a different S/r relation, we were not able to implement one version of this module, which can be used for both instances.*

4.1.5 H3 - Challenge Generation

The *H3* module is responsible for calculating the challenge, as described in Section 2.8.3. The first part of the challenge generation is producing a hash of all the output shares and commitments of all three players of all runs T combined with the public key and the message. Therefore, the

module includes a KECCAK instance, which is fed continuously with blocks of S bits. The module takes care of correctly concatenating the blocks and absorbing full inputs of r bits. At the end, the module also takes care to pad the input.

The module is synchronized with a *start*, *valid*, and a *ready* signal. At the beginning, the start signal has to be asserted to reset the inner state of the KECCAK instance. The module indicates that it is ready to absorb a new input block of S bits by asserting the *ready* signal. The *valid* signal is used to indicate that a S bit block is given to the module.

Since the module always absorbs S bit blocks, the output shares and the two parts of the public key can be absorbed at once; the commitments need to be split into two parts. As we describe in Remark 7, our overall PICNIC design is limited to only allow 512 bit messages. Therefore, the message needs to be divided into 4 blocks for PICNIC-L1-FS and 2 blocks for PICNIC-L5-FS respectively.

Absorbing all inputs takes 6490 clock cycles for PICNIC-L1-FS and 26220 clock cycles for PICNIC-L5-FS.

Remark 4. *Since the instances* PICNIC-L1-FS *and* PICNIC-L5-FS *have a different S/r relation, we were not able to implement one version of this module, which can be used for both instances.*

The second part of the challenge generation is to map the $\{0,1\}^l$ hash to $\{0,1,2\}^T$, according to Section 2.8.3. The module first stores the generated hash in a register and already starts to generate a new hash $h \leftarrow H(0x01 \,||\, h)$ using the same KECCAK instance as for absorbing the input. In parallel, the module starts to look at the two most significant bits of the hash h. If those bits are not equal to 11, the challenge is shifted left by two and most significant bits of h get appended to the challenge. Then the hash h is shifted left by two, and again the two most significant bits are handled. Once the hash h is entirely handled, the new hash $h \leftarrow H(0x01 \,||\, h)$ is already finished, and the process is started from the beginning.

The module finishes, when the challenge consists of $2 \cdot T$ bits. Since the signature size depends on the challenge, the module also calculates the size

of the signature during the challenge generation. As an output, the module gives the signature size and the finished challenge.

The number of clock cycles required for the second part of the challenge generation depends on the hash generated during the first part of the generation and therefore depends on the public key *pk*, secret key *sk* and the message *M*.

4.1.6 BRAM

PICNIC produces a lot of intermediate values. At the beginning of the signing process $3 \cdot T$ Seeds of size S bit are generated. During each of the T runs 3 input shares of size S bit, 3 transcripts of size $3 \cdot r \cdot m$ bit, 3 output shares of size S bit and 3 commitments of size $2 \cdot S$ bit need to be stored. These intermediate values together have a size of 81468 bit (= 101835 byte) for PICNIC-L1-FS and 3179880 bit (= 397485 byte) for PICNIC-L5-FS.

We use block RAMs (BRAMs) to store all these intermediate values. The targeted FPGA board has true dual-port BRAMs of size 36 kbit each. These BRAMs have two ports with separate addresses, data inputs, data outputs, and write-enable input. Both ports can be written to or read from independently. Both read and write accesses, are synchronous and require a rising edge of the clock. Usually the BRAMs have 32 bit data in/outputs; however, the synthesizer is capable of combining several BRAMs to enable bigger data widths.

We use separate BRAMs for each player containing T values of either the seeds, input shares, output shares, transcripts, or commitments to concurrently read all the values required to complete one run t of PICNIC's circuit decomposition.

Remark 5. *The true dual-port BRAMs allow to read two Seeds from two different addresses concurrently. Together with Remark 2, this enables us to calculate the random keys and tapes of* PICNIC*'s following MPC run* $t + 1$ *in parallel to calculating the commitments of the current run* t*, saving* $T \cdot 51 = 11169$ *clock cycles for* PICNIC-L1-FS *and* $T \cdot 75 = 32850$ *clock cycles for* PICNIC-L5-FS *in some high-level designs.*

4.1.7 Serialization and Deserialization of the Signature

We designed the PICNIC implementation to be capable of sending and receiving the signature split into blocks of size 128 bits. As described in Section 2.8.4, the signature is build from the challenge, and all the intermediate values required to recompute and verify the two opened players. Most parts of the signature have a size which is an integer multiple of 128 bit, however since the sizes of the challenge and the transcripts cannot be divided by 128 we had to implement an *Input-FIFO* to assemble the signature from 128 bit blocks and an *Output-FIFO* to split the signature.

The *Output-FIFO* basically has two data inputs, one with a size of 128 bit, the size of the other input is the remainder of the size of a transcript divided by 128. During serialization, we split the commitments, seeds, input shares, and output shares into blocks of 128 bit and give it to the *Output-FIFO*. For the transcript we basically do the same; however, we feed the remaining data into the other input of the FIFO. The *Output-FIFO* then takes care of correctly assembling an output block of size 128 bit and memorizing the unused data for the next block.

The *Input-FIFO* has an 128 bit data input, an 128 bit data output and an output with a size equal to the remainder of the size of a transcript divided by 128. We can either request data from one (or two if possible) of the two outputs of the FIFO by asserting the correct control signals. The *Input-FIFO* then takes care of memorizing the part of the data, that has not been requested, and assembles correct outputs from the data input and the memorized data. For receiving the commitments, seeds, input shares, and output shares, we only have to request data from the 128 bit output of the FIFO, for the transcript we request data from both outputs.

Both *Input-FIFO* and *Output-FIFO*, are implemented to be capable of handling one 128 bit block per clock cycle.

Remark 6. *As we describe in Section 4.4.1, we can only send/receive at most* 65520 *byte of data for the* PICNIC-L5-FS *before a new segment header has to be transmitted. We implemented the Output-FIFO and Input-FIFO for* PICNIC-L5-FS *to keep track of the sizes of the already transmitted data and to automatically send the new header or process the received header respectively.*

4.2 High-Level Design

In this project, we developed several different VHDL designs for PICNIC-L1-FS and PICNIC-L5-FS. We implemented a standalone version for only message signing and signature verification, as well as a version which is capable of doing both.

Due to the big hardware utilization, especially for the PICNIC-L5-FS instances, we implemented different configurations for the different designs. The designs basically differ in two configurations:

- We implemented the MPC module of LowMC to be capable of calculating one round of encryption during one clock cycle. However, for higher clock frequencies, the resulting critical path after synthesis is too big. Therefore, we also implemented a version of the MPC module in which we split the round of LowMC to be calculated in two clock cycles.
- In Remark 2 and Remark 5, we describe that we are capable of calculating the randomness required for run $t+1$ of PICNIC's circuit decomposition in parallel to calculating the commitments of run t reducing the overall number of clock cycles for signing and verification. However, this optimizations increase the critical path and can, therefore not be used in all designs.

For the PICNIC-L5-FS instance, we developed two versions of the three VHDL designs, a version designed for higher frequencies without the previously described optimizations, and a low-frequency (LF) version which includes the optimizations and therefore reduces the overall number of clock cycles. Table 4.1 shows the different implemented versions of PICNIC and which configurations they use.

The overall design of the implementations is always a nested state machine, where the high-level design basically connects the inputs and outputs of all the submodules described in Section 4.1. Figure 4.2 shows a flowchart for the designs where the randomness for run $t+1$ is calculated in parallel to the commitments of run t, Figure 4.3 shows a diagram of the designs without this optimization. In both figures, the signing process is shown on the left side, the verification process on the right. In the designs which are capable

of doing both signing and verification, both processes are implemented. Most of the submodules can be reused for both signing and verification, only the MPC module has to implement two different Sbox calculations, and the combined design has to include both the *Serialize* and *Deserialize* submodules. Therefore, the difference in hardware utilization between a sign-only design and a sign/verify design is quite low.

Remark 7. *Since* PICNIC *hashes the message twice using* KECCAK*, we limited our* PICNIC *design only to allow signing messages of size* 64 *bytes. This constraint simplifies implementation and also decreases area and runtime of the coprocessor. As described in the official* PICNIC *specification [12], it is recommended to use* SHAKE-256 *with* $l = 512$ *bit,* SHA3-512 *or* SHA-512 *to hash messages before they are signed.*

4.3 AXI4-Stream Interface

The interfaces of the implemented PICNIC modules can be seen in Figure 4.4. The designs basically have three AXI4-Stream interfaces, one public data

Table 4.1: Different configurations of the PICNIC implementations.

Name	LowMC cycles / round	parallel *Tapes* / *Commits*	sign	verify
PICNIC-L1-SIGN	2	✓	✓	×
PICNIC-L1-VERIFY	2	✓	×	✓
PICNIC-L1	2	✓	✓	✓
PICNIC-L5-SIGN	2	×	✓	×
PICNIC-L5-VERIFY	2	×	×	✓
PICNIC-L5	2	×	✓	✓
PICNIC-L5-SIGN-LF	1	✓	✓	×
PICNIC-L5-VERIFY-LF	1	✓	×	✓
PICNIC-L5-LF	1	✓	✓	✓

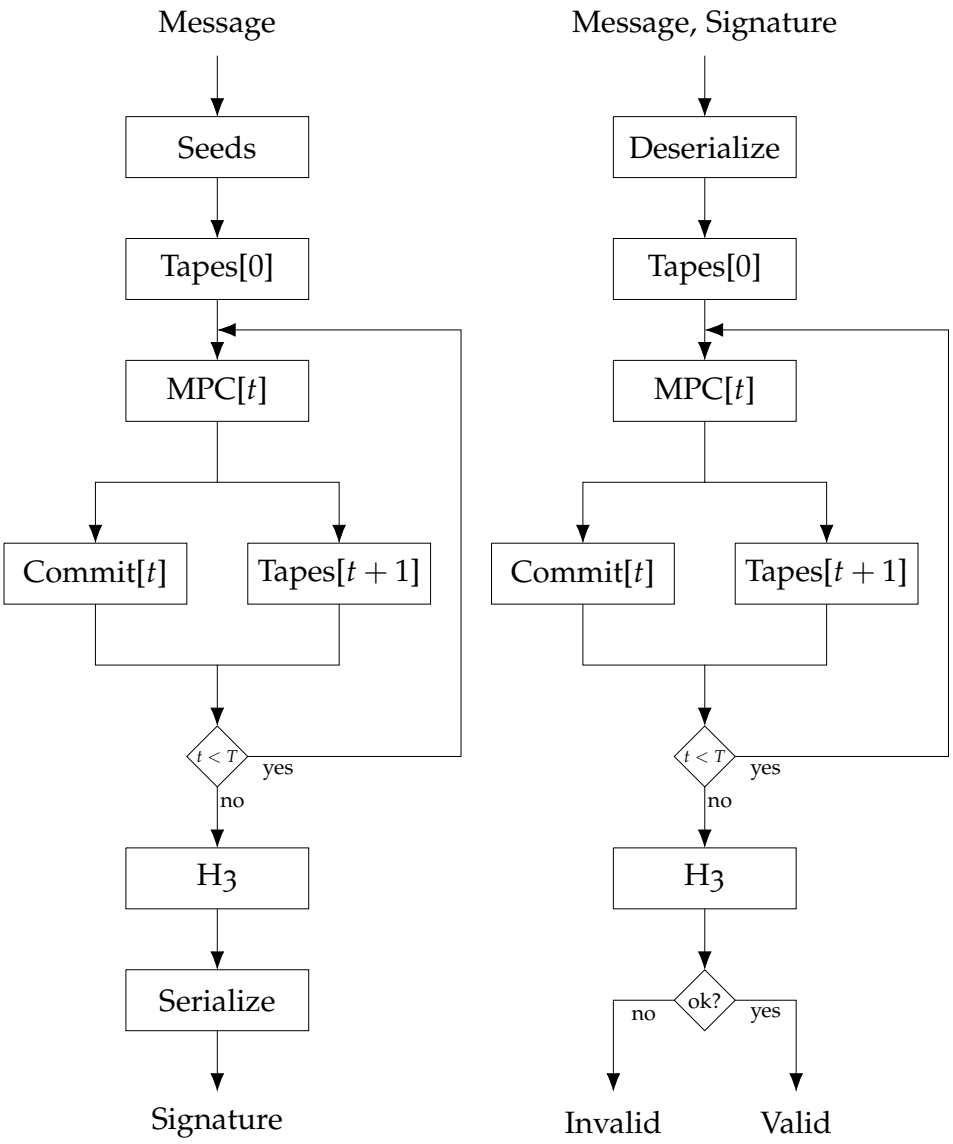


Figure 4.2: High-level design of PICNIC signing (left) and verification (right) with parallel *Commits*/*Tapes*.

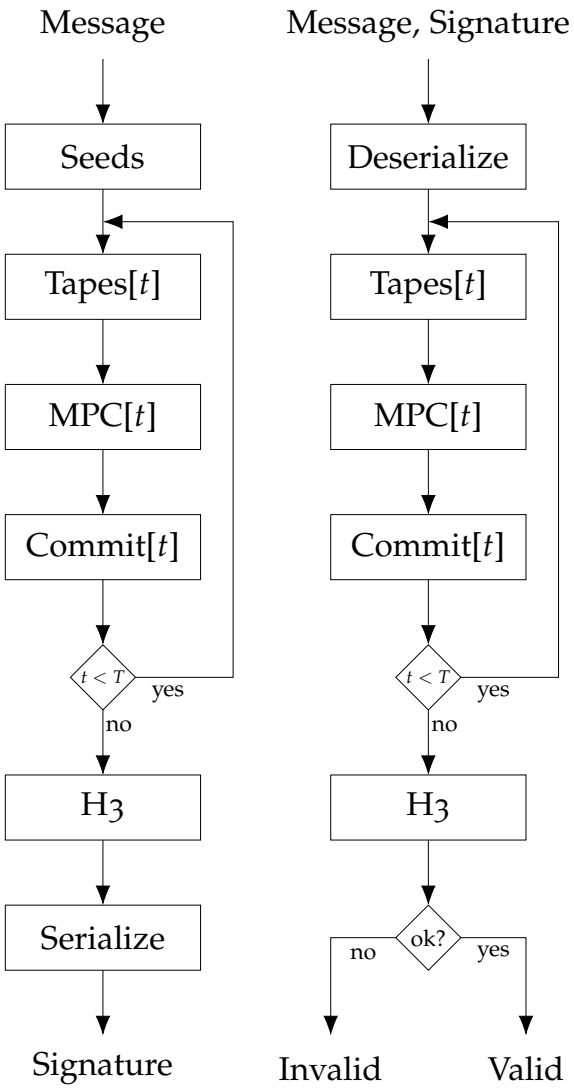


Figure 4.3: High-level design of PICNIC signing (left) and verification (right).

input (*pdi*), one secret data input (*sdi*), and one public data output (*pdo*) interface. The verification-only designs only have two interfaces, since they do not require any secret data.

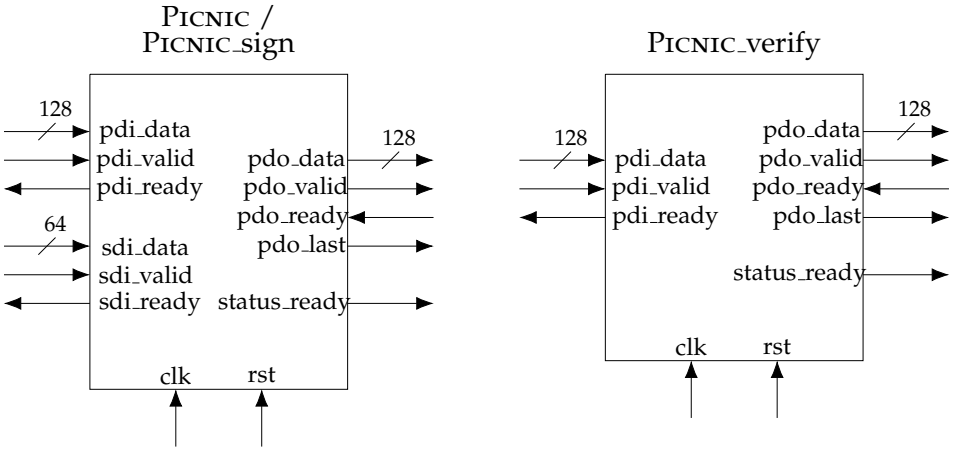


Figure 4.4: Interface of the PICNIC VHDL implementation.

AXI4-Stream is an open standard for on-chip bus connections designed for high data throughput. It is straightforward to use and allows data transmission in parallel to the handshake. In its basic form, the interface has a *data* line and a *valid* and *ready* signal for synchronizing the data.

Figure 4.5 shows the timing of a data transaction over an AXI4-Stream interface. When the master is ready to send data, it tries to send them over the *data* line and also asserts the *valid* signal. The slave asserts the *ready* signal if it is ready to receive the data. If both signals, *ready* and *valid*, are asserted the master, and the slave know that the data transmission was successful. Some AXI4-Stream slaves also require a *last* signal which indicates if the current block is the last one of a packet. Therefore, we implemented the *pdo* interface to also assert a *last* signal when necessary. The advantage of the AXI4-Stream interface is, that it does not require extra clock cycles for the handshake and therefore can be used for high data throughput.

As shown in Figure 4.4, our PICNIC modules use a data width of 128 bit for the public data interfaces and a width of 64 bit for the secret data interface.

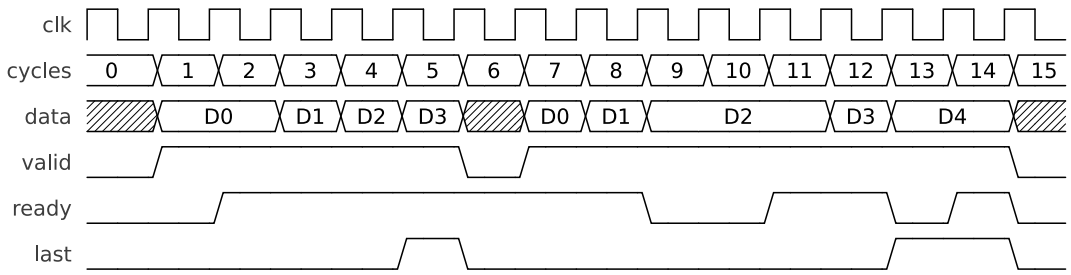


Figure 4.5: Timing of an AXI4-Stream transaction. Picture Taken from [37].

4.4 Communication Protocol

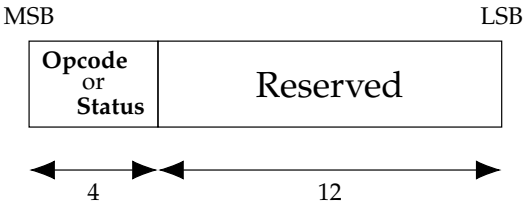
At the time of writing, PICNIC is a second round candidate for the NIST Post-Quantum Cryptography Standardization Process [1]. NIST published an API for hardware implementations [22] in which they define a communication protocol to interact with the post-quantum cryptosystems.

For post-quantum signature schemes, NIST defines that the hardware core should have three AXI4-Stream interfaces, one for public data inputs (`pdi`), one for secret data inputs (`sdi`) and a public data output (`pdo`) interfaces. They allow a data width up to 128 bit for the public data interfaces, and a width of up to 64 bit for the secret data interfaces [22]. As described in Section 4.3 we implemented our PICNIC cores to conform with these definitions.

4.4.1 Instructions, Headers and Status Codes

In the API description, NIST defines special formats for different instructions, data headers, and status codes. Figure 4.6 shows the 16 bit format of the instructions and status codes, which gets padded with zeros to the data width size of the used AXI4-Stream interface [22].

The API defines a segment header which should be sent for every data. The header format can be seen in Figure 4.7; it includes a field for the segment type and a 16 bit field for the segment length [22]. The different segment type encodings can be seen in Table 4.2. Since the field for the segment length is



Opcode	**Status**
0010 - Sign (SGN)	1110 - Success
0011 - Verify (VER)	1111 - Failure
0110 - Load Private Key (LDPRIVKEY)	

Figure 4.6: Instruction and status format.

defined to have a size of 16 bit, the maximum transmitted segment size is 65535 byte. The signature size of PICNIC-L5-FS is $\leq$ 132824 byte. Therefore, for sending the Signature for signing or verification for the PICNIC-L5-FS instance, we have to split the the signature into multiple segments, each with its own header.

Figure 4.7: Segment header format.

The *EOI* (End Of Input) and *Last* fields of the segment header indicate whether the current segment is the last segment of a given instruction. Therefore, in our PICNIC implementations, those two bits are 11 for the message segment during signing and for the last signature segment in the PICNIC-L5-FS implementations. Otherwise, those two bits are always 00.

4.4.2 Setting the Keys

Figure 4.8 shows the protocol to set the public key and the protocol to set the private key. For the public key, first, a segment header is transmitted before the key itself is send via the public data input interface. For the private key, first, the instruction is transmitted via the public data input interface, before the segment header and the private key are transmitted via the secret data input interface.

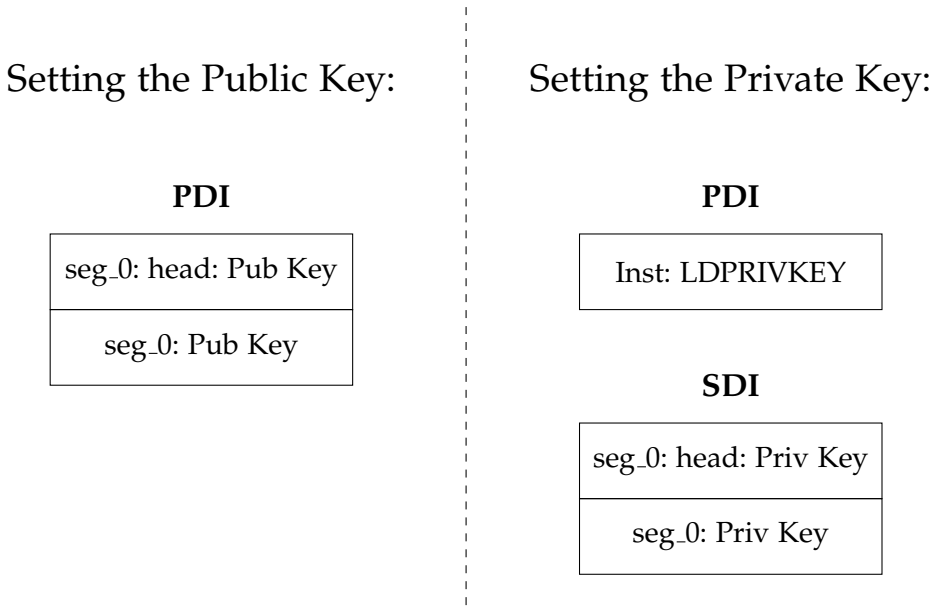


Figure 4.8: Protocol of setting the public key (left) and private key (right).

Table 4.2: Segment Type Encoding

Encoding	Type
001	Message
010	Signature
101	Public Key
110	Private Key

Our PICNIC implementations expect that the keys are transmitted within only one segment.

4.4.3 Message Signing

Figure 4.9 shows the protocol to sign a message. First, the instruction, the segment header for the message and the message are transmitted via the public data interface, then the PICNIC cores output the signature including segment headers via the public data output interface before the status *Success* is transmitted.

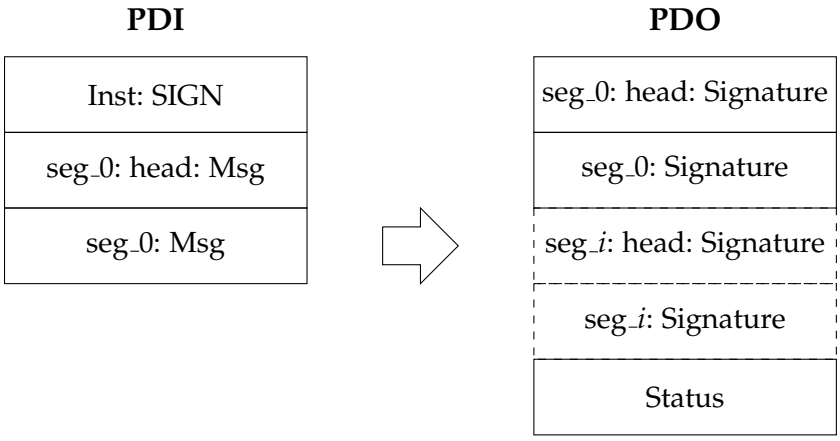


Figure 4.9: Protocol of signing a message.

Our PICNIC implementations expect that the message is transmitted within only one segment. The PICNIC-L1-FS cores output the signature within one segment; the PICNIC-L5-FS cores split the signature into segments of 65520 byte and one segment containing the rest.

4.4.4 Signature Verification

Figure 4.10 shows the protocol to verify a signature. First, the instruction, the message, and the signature are transmitted including segment headers via

the public data input interface before the PICNIC cores transmit the message and a status via the public data output interface. The status indicates whether the signature is valid (*Status Success*) or not (*Status Failure*).

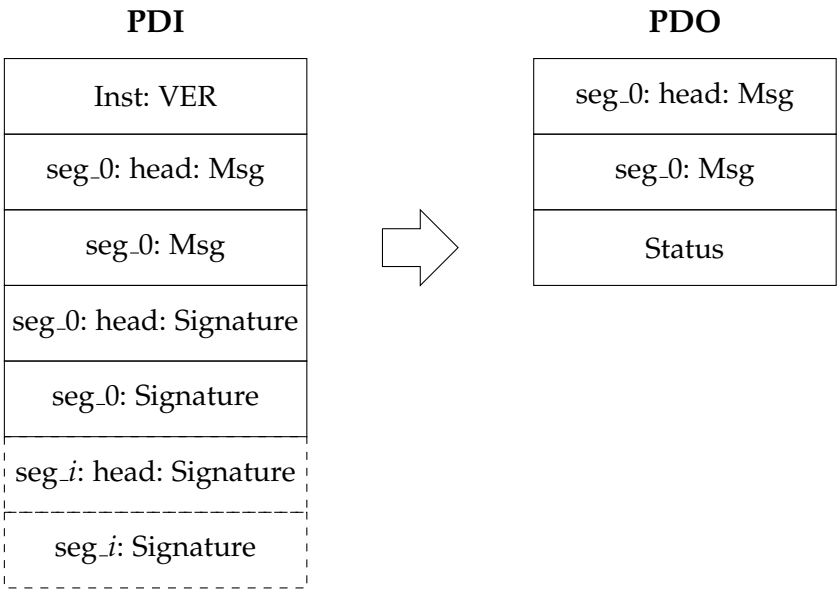


Figure 4.10: Protocol of verifying a signature.

Our PICNIC-L1-FS implementations expect that the message and the signature are both transmitted within only one segment each, our PICNIC-L5-FS implementations allow the signature split into segments with an integer multiple of 16 byte as size and the last segment containing the rest.

5 The Picnic-PCIe Coprocessor

In this Chapter, we describe how we implemented the complete Picnic-PCIe coprocessor based on our Picnic-VHDL design and how we can access it using a C-interface. We first describe the hardware and software we use in this thesis to design the coprocessor before we explain how we are able to connect our Picnic-VHDL design to the PCIe port of the targeted FPGA board. Finally, we explain how the coprocessor can be accessed from a PC and how to sign messages and verify signatures from within a C-program.

5.1 Hardware and Software

In this thesis, we use the Xilinx Kintex7 FPGA KC705 Evaluation Kit as the hardware for the developed Picnic coprocessor. This Evaluation Kit includes an XC7K325T-2FFG900C FPGA which has 326080 programmable logic cells and is also equipped with an 8-lane PCIe connector [39].

We use Xilinx Vivado v2018.2 [41] to design the high-level design of the coprocessor which connects our VHDL design to the PCIe connector of the FPGA. We also use Vivado to synthesize, place, and route the design and to flash the resulting bitstream onto the FPGA.

5.2 PCIe/DMA Subsystem

Vivado [41] already provides several different intellectual properties (IP's) to access the PCIe connectors of the used Evaluation Kit. In this thesis, we use the *DMA/Bridge Subsystem for PCI Express (PCIe)* IP which acts as a full

DMA system that redirects memory accesses from the host PC to the FPGA board via the PCIe interface. This DMA can be configured to use several AXI4-Stream interfaces in master and/or slave mode with frequencies of either 125 MHz or 250 MHz to communicate with other IP's on the FPGA. More information and a usage guide to this DMA Subsystem can be found in the Xilinx product guide [38].

We configured the PCIe/DMA Subsystem to use three 128 bit AXI4-Stream interfaces, to conform with the communication protocol proposed by NIST, which we describe in Section 4.4. The PCIe/DMA is configured to have two master interfaces to write either public and secret data to the coprocessor and one slave interface to read the signature or the status of the verification from the FPGA-board. We drive the DMA with a 125 MHz clock, since our design is not capable of running with 250 MHz.

5.3 High-Level Coprocessor Design

Figure 5.1 shows the high-level design of the implemented PICNIC coprocessors. Basically, every AXI4-Stream interface of the PCIe/DMA Subsystem is connected to an interface of our PICNIC core. However, since the DMA is only capable of being configured to use the same data width for all three interfaces, we had to add a *Key Converter* which basically transforms the 128 bit AXI4-Stream interface to a 64 bit interface to access the *sdi* interface of our PICNIC core.

As described in Section 4.2, we also implemented a low-frequency version of the PICNIC-L5-FS instance, which includes optimizations to reduce the number of clock cycles for signing and verification. Figure 5.2 shows the high-level design of our low-frequency PICNIC coprocessors. The main difference to the design of Figure 5.1 is that we drive our PICNIC core and the *Key Converter* with a different clock for the low-frequency coprocessors. For this reason, we had to add logic to synchronize the AXI4-Stream interfaces between the two different clock domains. For asynchronous clock conversions ($\frac{f_1}{f} \neq \frac{1}{N}$) the additional logic requires a lot of hardware utilization. Combined with the already significant utilization of our PICNIC core, this

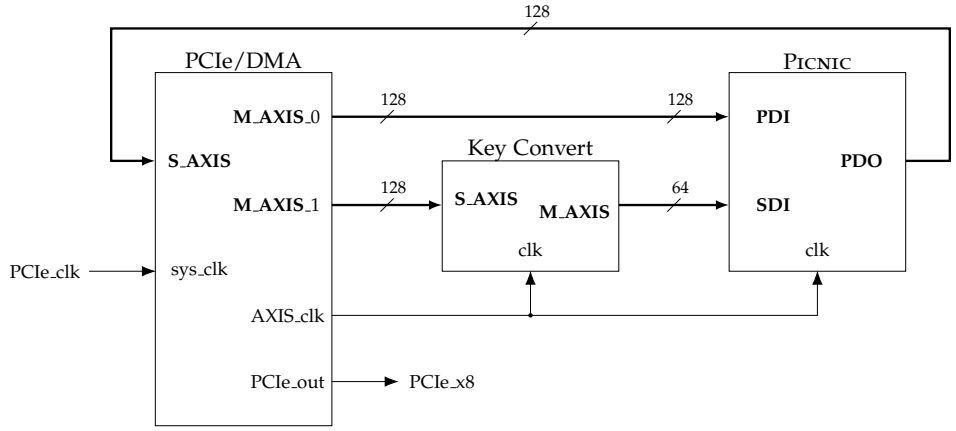


Figure 5.1: High-level design of the PICNIC-PCIe coprocessors.

additional logic would exceed the provided resources of the targeted FPGA. Therefore, to reduce the requirements of the clock conversion logic, we had to use a clock with a frequency of $f_1 = \frac{f}{N}$. In this case, the logic for the clock conversions is very simple and does not require many resources. We, therefore, added a *Clock Wizard* to our design, which reduced the clock frequency for the PICNIC core by half to 62.5 MHz.

In both cases, high and low-frequency designs, the verify-only PICNIC cores do not have a *sdi* interface. Therefore, the PCIe/DMA Subsystems for the verify-only coprocessors are configured to have only one AXI4-Stream master interface. Additionally, the *Key Converter* is not required as well. Besides these changes, there are no differences between the high-level designs of the verify-only coprocessor and the designs shown in Figure 5.1 and Figure 5.2.

5.4 Software Access to the Coprocessor

In this section, we describe how we can communicate with the PICNIC-PCIe coprocessor. In Section 5.4.1, we describe how to install the driver of the PCIe/DMA Subsystem, and in Section 5.4.2, we explain the developed C-library for signing and verification using our coprocessor.

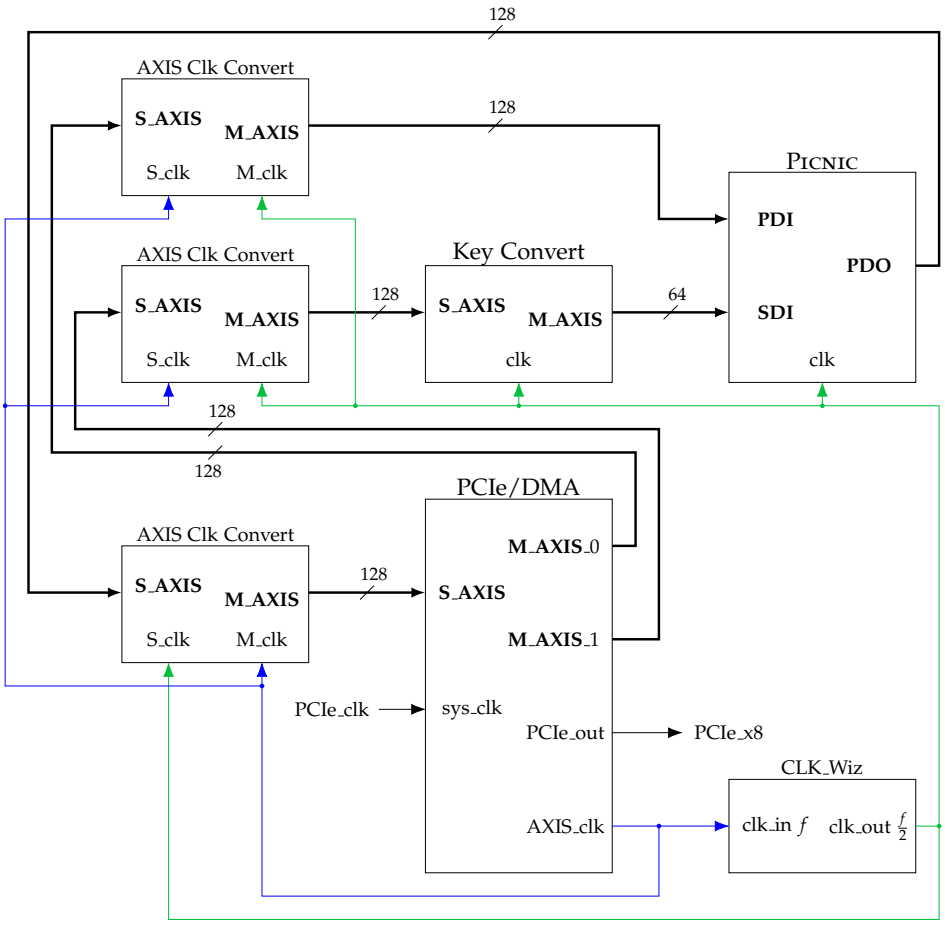


Figure 5.2: High-level design of the low-frequency PICNIC-PCIe coprocessors.

5.4.1 Driver Setup

To access the PCIe/DMA Subsystem of the FPGA, we first have to install the driver software provided by Xilinx which can be downloaded from their support page [40]. The driver is available for Linux and Windows systems.

The driver software detects the used AXI4 interfaces used by the PCIe/DMA Subsystem of the coprocessor and maps them to file descriptors on the host PC. The driver then redirects all read and write accesses to these file descriptors to the PCIe coprocessor. On Linux the AXI4-Stream interfaces used in this thesis are mapped to the following file descriptors:

- `/dev/xdma0_h2c_0`
  Accesses the AXI4-Stream interface that is used as the public data input (*pdi*) of the coprocessor.
- `/dev/xdma0_h2c_1`
  Accesses the AXI4-Stream interface that is used as the secret data input (*sdi*) of the coprocessor.
- `/dev/xdma0_c2h_0`
  Accesses the AXI4-Stream interface that is used as the public data output (*pdo*) of the coprocessor.

The driver software requires all buffers used in the read/write transactions to be page aligned to redirect all data to the PCIe port properly. This driver software also requires the *last* signal of the *pdo* interface to be asserted correctly, otherwise read accesses to the corresponding file descriptors would always fail.

5.4.2 C-Library

We developed a C-Library to easily sign and verify messages using our PICNIC-PCIe coprocessors. The library includes the following functions:

- `int init_picnic_fpga()`
  This function opens the files `/dev/xdma0_h2c_0`, `/dev/xdma0_h2c_1` and

`/dev/xdma0_c2h_0` to communicate either with the developed PICNIC_sign or PICNIC coprocessor. The function returns 0 if no error occurs.

- `int init_picnic_fpga_verify()`
  This function opens the files `/dev/xdma0_h2c_0` and `/dev/xdma0_c2h_0` to communicate with the developed PICNIC_verify coprocessor. The function returns 0 if no error occurs.
- `void release_picnic_fpga()`
  This function closes all open file descriptors.
- `int picnic_fpga_set_key(unsigned char* key, int version)`
  This function follows the protocol described in Section 4.4.2 to push the secret key via the *sdi* interface to the coprocessor. The parameter `version` specifies the security level of the target coprocessor (`L1` / `L5`) and therefore the size of `key`. The function returns 0 if writing was successful.
- `int picnic_fpga_set_pub(unsigned char* pub_plain, unsigned char* pub_ciph, int version)`
  This function follows the protocol described in Section 4.4.2 to push the public key via the *pdi* interface to the coprocessor. The parameter `version` specifies the security level of the target coprocessor (`L1` / `L5`) and therefore, the size of the two parts of the public key `pub_plain` and `pub_ciph`. The function returns 0 if writing was successful.
- `int picnic_fpga_sign(unsigned char* msg, unsigned char* sig, size_t* sig_length, int version)`
  This function follows the protocol described in Section 4.4.3 to sign the message `msg` of size 64 byte using the PCIe coprocessor. The parameter `version` specifies the security level of the target coprocessor (`L1` / `L5`). The function first writes the message to the *pdi* interface before it reads the signature from the *pdo* interface. As described in Section 4.4.3, the signature has to be split into multiple parts for the `L5` security level; this function automatically handles the receiving of the separated parts and assembling of the complete signature. If no error occurs 0 is returned, the signature is written to `sig`, and the length of the signature is written to `sig_length`.
- `int picnic_fpga_verify(unsigned char* msg, unsigned char* sig, size_t sig_length, int version)`
  This function follows the protocol described in Section 4.4.4 to verify

the signature `sig` of length `sig_length` of the message `msg` of size 64 byte. The parameter `version` specifies the security level of the target coprocessor (`L1` / `L5`). The function first writes the message and the signature to the *pdi* interface before it reads the message and the status from the *pdo* interface. As described in Section 4.4.4, the signature has to be split into multiple parts for the `L5` security level; this function automatically handles the splitting and sending of the separated parts of the signature. If no error occurs, the function returns `SIG_VERIFIED` for a valid signature and `SIG_FALSE` for an invalid one.

- `unsigned char* alloc_resource(size_t size)`
  This function allocates a page aligned buffer with `size` bytes on the heap using `posix_memalign(...)`. We provide this function because the driver of the PCIe/DMA Subsystem expects all buffers to be page aligned in order to function properly.

6 Practical Evaluation

In this Chapter, we describe the results of this thesis. We first give an overview of the hardware utilization required by our design, and then we present some benchmark results for signing and verification and compare them to state-of-the-art software implementations of Picnic.

6.1 Hardware Utilization

In this Section, we present the hardware utilization of our Picnic implementations. First, we give an overview of the required utilization of the Picnic submodule; then we show the utilization of the developed coprocessors. The used FPGA has 203800 lookup-tables (LUTs), 407600 flip-flops (FF) and 445 BRAMs available.

6.1.1 Picnic Submodules

Table 6.1 compares the hardware Utilization of the different submodules of our Picnic-L1-FS implementation; Table 6.2 compares them for the Picnic-L5-FS implementation.

Table 6.1 and Table 6.2 show that the LowMC-MPC modules require by far the most hardware utilization. As described in Section 4.1.2 the submodule which is able to do the LowMC-MPC encryption for both signing and verification only requires less than one percent more LUTs then the submodule which can only be used for signing.

Table 6.1: Hardware Utilization for different parts of the L1 design.

Design Part	LUTs	%	FF	%
Keccak	3726	1.83 %	1606	0.39 %
Tapes (3× Keccak)	8241	4.04 %	5589	1.37 %
Commits (3× Keccak)	12221	6.00 %	5589	1.37 %
Seeds (1× Keccak)	5867	2.88 %	1846	0.45 %
H_3 (1× Keccak)	6910	3.39 %	3641	0.89 %
Input-FIFO	1962	0.96 %	125	0.03 %
Output-FIFO	2025	0.99 %	125	0.03 %
LowMC-MPC Sign	31837	15.62 %	3060	0.75 %
LowMC-MPC Verify	29756	14.60 %	1126	0.28 %
LowMC-MPC	32224	15.81 %	3061	0.75 %

Table 6.2: Hardware Utilization for different parts of the L5 design.

Design Part	LUTs	%	FF	%
Keccak	3726	1.83 %	1606	0.39 %
Tapes (3× Keccak)	10465	5.13 %	9621	2.36 %
Commits (3× Keccak)	14160	6.95 %	6357	1.56 %
Seeds (1× Keccak)	8974	4.40 %	2640	0.65 %
H3 (1× Keccak)	8463	4.15 %	4085	1.00 %
Input-FIFO	1608	0.79 %	172	0.79 %
Output-FIFO	2317	1.14 %	155	0.04 %
LowMC-MPC Sign	97066	47.63 %	5940	1.46 %
LowMC-MPC Verify	93959	46.10 %	2246	0.55 %
LowMC-MPC	98319	48.24 %	5958	1.46 %

6.1.2 Picnic Coprocessors

Table 6.3 compares the hardware utilization of the different submodules of the final coprocessors, including our 9 different completed Picnic cores. Our Picnic cores require a lot of LUTs on the used FPGA, especially the Picnic-L5-FS implementations. The PCIe/DMA Subsystem which connects the Picnic cores the PCIe port of the used FPGA board adds about 11 % more of the available LUTs to the design.

Table 6.3: Hardware Utilization for different parts of the coprocessors.

Design Part	LUTs	%	FF	%	BRAM	%
Key Convertor	90	0.04 %	3	0.00 %	0	0 %
Clock Convertor	153	0.08 %	299	0.07 %	0	0 %
PCIe/DMA	22216	10.90 %	22692	5.57 %	42.5	9.55 %
PCIe/DMA Verify	18754	9.20 %	20116	4.94 %	41.5	9.33 %
Picnic-L1	84788	41.6 %	22803	5.59 %	52.5	11.80 %
Picnic-L1-sign	74456	36.53 %	20801	5.10 %	52.5	11.80 %
Picnic-L1-verify	64683	31.74 %	16487	4.04 %	33.5	7.53 %
Picnic-L5	160886	78.94 %	32776	8.04 %	98.5	22.13 %
Picnic-L5-sign	147228	72.24 %	30216	7.41 %	98.5	22.13 %
Picnic-L5-verify	131606	64.58 %	23826	5.85 %	62.5	14.04 %
Picnic-L5-lf	165807	81.36 %	33553	8.23 %	117	26.29 %
Picnic-L5-sign-lf	148999	73.11 %	29608	7.26 %	117	26.29 %
Picnic-L5-verify-lf	136862	67.16 %	24300	5.96 %	77.5	17.42 %

Table 6.4 shows the utilization of the completed coprocessors. Basically, the final coprocessors require as many resources, as the Picnic cores and the PCIe/DMA Subsystem combined. The Picnic-L5-FS coprocessors need more than $\sim \frac{4}{5}$ of all available LUTs making it very hard for the synthesizer to effectively place and route the design onto the FPGA resulting in longer synthesize times and bad slack times due to congestion.

Remark 8. *The coprocessors* Picnic-L5-sign, Picnic-L5-verify *and* Picnic-L5 *do not meet the timing requirements after synthesis for a clock frequency of* 125 *MHz. However, reducing the clock to* 100 *MHz would be enough to meet all the requirements. Since we were not able to configure the PCIe/DMA Subsystem to run at this frequency and the timing constraints all are worst-case constraints we tried to ignore the warnings and just run the coprocessors at* 125 *MHz. This is equivalent to overclocking the design, and in our benchmarks, we were able to produce valid signatures and correct verification without any problems. However, we also wanted to design coprocessors which meet the timing requirements. This is the main reason why we also implemented the low-frequency coprocessors, where we included the optimizations to reduce the clock cycles for signing and verification. These designs meet the timing requirements and work without overclocking the coprocessor.*

6.2 Benchmarks

In this Section, we compare the runtime of the developed coprocessors to state-of-the-art software implementations of Picnic.

Table 6.4: Hardware Utilization for the complete coprocessors.

Design Part	LUTs	%	FF	%	BRAM	%
Picnic-L1	104819	51.43 %	43370	10.64 %	85	19.10 %
Picnic-L1-sign	94231	46.24 %	41271	10.13 %	85	19.10 %
Picnic-L1-verify	81228	39.86 %	34484	8.46 %	63	14.16 %
Picnic-L5	180302	88.47 %	53514	13.13 %	131	29.44 %
Picnic-L5-sign	166894	81.89 %	51073	12.53 %	131	29.44 %
Picnic-L5-verify	147943	72.59 %	42126	10.34 %	92	20.67 %
Picnic-L5-lf	185711	91.12 %	54966	13.49 %	149.5	33.6 %
Picnic-L5-sign-lf	169096	82.97 %	51048	12.52 %	149.5	33.6 %
Picnic-L5-verify-lf	153647	75.39 %	42889	10.52 %	107	24.04 %

6.2.1 Benchmark Platforms

We compare the runtime of our implemented coprocessors to optimized state-of-the-art Picnic software implementations on different platforms which can be seen in Table 6.5.

Table 6.5: Different Benchmark Platforms.

Platform	CPU	RAM	OS	GCC
A	Intel i7-960, 3.2 GHz	16 GB	Debian 9 stable	6.3.0
B	Intel i7-4790, 3.6 GHz	16 GB	Ubuntu 18.04.1	7.3.0
C	Intel E31230, 3.2 GHz	8 GB	Ubuntu 18.04.2 LTS	7.3.0

We used platform A to test our coprocessors, platforms B and C were used in the official Picnic design document [14] to test their highly optimized software implementations.

6.2.2 Timing Benchmarks

Table 6.6 shows the average runtime of the developed coprocessors for signing and verification. The column *FPGA runtime* is the calculated time resulting from the clock frequency and the number of clock cycles (including 1 cycle per 128 bit of data transmission) and therefore is the actual runtime of the FPGA. The column *C-Access runtime* is the measured runtime using our developed C-Library on platform A.

As Table 6.6 shows, the C-Library adds some overhead to the signing and verification process. For signing the overhead adds about ~ 0.1 ms to the runtime, for verification, the overhead is more significant. Especially for the Picnic-L5-FS the actually measured runtime is much bigger than the raw verification runtime of the coprocessor. We conclude that the driver for the PCIe/DMA Subsystem is slower for writing large amounts of data, like the Picnic-L5-FS signature, from the PC to the FPGA board.

Table 6.7 compares the runtime of the optimized C-implementation of Picnic to an optimized version which uses processor-specific compiler intrinsics on two different benchmark platforms as described in the official Picnic design document [14].

As Table 6.7 shows, the runtime of Picnic highly depends on the underlying hardware and if the CPU includes *single instruction, multiple data* (SIMD)

Table 6.6: Runtime comparison of the coprocessors on benchmark platform A.

Coprocessor	clock frequency	clock cycles	FPGA runtime	C-Access runtime
	MHz		ms	ms
Picnic-L1-sign	125	$\sim$ 31300	0.250	0.349
Picnic-L1-verify	125	$\sim$ 29600	0.237	0.395
Picnic-L5-sign	125	$\sim$ 154500	1.236	1.383
Picnic-L5-verify	125	$\sim$ 146600	1.173	2.128
Picnic-L5-sign-lf	62.5	$\sim$ 104200	1.667	1.798
Picnic-L5-verify-lf	62.5	$\sim$ 96300	1.541	2.423

Table 6.7: Runtime comparison of optimized software implementations [14].

Platform	Parameters	use SIMD	Sign	Verify
B	Picnic-L1-FS	✓	1.44 ms	1.15 ms
B	Picnic-L5-FS	✓	5.87 ms	4.92 ms
B	Picnic-L1-FS	×	2.82 ms	2.34 ms
B	Picnic-L5-FS	×	12.37 ms	10.59 ms
C	Picnic-L1-FS	✓	4.20 ms	3.40 ms
C	Picnic-L5-FS	✓	17.67 ms	14.67 ms
C	Picnic-L1-FS	×	4.41 ms	3.56 ms
C	Picnic-L5-FS	×	19.52 ms	16.81 ms

primitives, like SSE2 and AVX2, which would further improve execution time. However, in any case, our developed coprocessors are faster than the corresponding software counterparts. For Picnic-L1-FS signing is ~ 4 times faster then the fastest software implementation, verification is ~ 3 times faster. For Picnic-L5-FS the overclocked high-frequency implementations are ~ 4 times faster for signing and ~ 2.3 times faster for verification, the low-frequency coprocessors are ~ 3.2 times faster for signing and ~ 2 times faster for verification. For CPUs which do not include the performance improving SIMD primitives and for C-only implementations the speedup of our coprocessors is even more significant. As described before, the relatively bad verification performance of our coprocessors is most likely due to the driver for the PCIe/DMA Subsystem.

6.3 Comparison to other Signature Schemes

In this Section, we compare our Picnic coprocessors to implementations of other signature schemes.

Table 6.8 compares several different FPGA implementations of various signature schemes, the runtime for signing t is calculated from the clock frequency and the number of clock cycles and therefore does not take the overhead of any transmission of data via a C-program into account. Thus this value compares to the column *FPGA runtime* of Table 6.6.

As Table 6.6 and Table 6.8 show, our Picnic-L5-FS coprocessors, which have the same security level as a SPHINCS-256 [5] coprocessor, have a similar runtime for signing on the Kintex-7 (K7) FPGA. The implementations of the traditional signature schemes RSA [31] and ECDSA [4] on a Virtex-7 (V7) FPGA are way slower than our coprocessors. The implementation of BLISS-IV [29], another post-quantum signature scheme based on lattices, on a Spartan-6 (S6) FPGA is very efficient regarding area and runtime for signing; however, it has a lower security level, and its post-quantum properties are not well understood [5].

Contrary to the good runtime of our implemented coprocessor, the hardware utilization is very high in comparison to implementations of the other sig-

nature schemes. This is due to the fact that our Picnic cores include a high number of Keccak and LowMC primitives, where especially the LowMC instances have a high hardware utilization on their own. In comparison, the coprocessor of SPHINCS-256, a hash-based post-quantum signature scheme, only consists of one pipelined ChaCha12 [25] instance and one instance of BLAKE-256 [6] and therefore requires less hardware utilization [5].

Table 6.8: Comparison of different FPGA signature scheme implementations. Table modified from [5].

Scheme	Security Classic	PQ	FPGA	Area LUT/FF/BRAM	f MHz	*t* ms
SPHINCS-256 [5]	256	128	K7	19067/38132/36	525	1.53
ECDSA-256 [4]	128	0	V7	6816/4442/0	225	1.49
ECDSA-521 [4]	256	0	V7	8273/7689/0	161	5.02
RSA-2048 [31]	112	0	V7	3558 slices/0	399	5.68
BLISS-IV [29]	192	?	S6	6438/6198/7	135	0.35
Picnic-L1	128	64	K7	104819/43370/85	125	0.25
Picnic-L5	256	128	K7	180302/53514/131	125	1.24
Picnic-L5-lf	256	128	K7	185711/54966/149.5	62.5	1.67

7 Conclusion and Future Work

In this Chapter, we conclude the thesis and discuss some possible future work.

7.1 Conclusion

To summarize, we were able to implement different coprocessors for Picnic-L1-FS and Picnic-L5-FS, including standalone coprocessors for only signing and only verification. Our coprocessors create signatures at least 4 times faster and verify signatures at least 2 times faster than a software implementation. The exact speedup highly depends on the hardware of the platform running the software implementation of Picnic, for most CPUs the runtime improvement will be much higher.

Our coprocessors perform signing faster than the FPGA implementation of traditional signature schemes, like ECDSA, and have a similar performance to SPHINCS-256, another post-quantum signature scheme.

We also created an area optimized VHDL implementation of LowMC by considering recently published optimizations to LowMC's linear layer [16]. This area reduced LowMC implementation enabled us to implement the Picnic-L5-FS coprocessors in the first place.

However, the hardware utilization of the present Picnic implementation is very high, which is a direct result of having multiple LowMC instances in the design, even though these LowMC instances include all known optimizations to reduce the constants required for a LowMC encryption.

7.2 Future Work

During the development of this project, the Picnic specification was updated. Amongst other minor modifications, the new design includes a salt when deriving random key shares and tapes at the beginning of a run and during challenge creation [13]. However, the changes of this update are not yet implemented in the current design of the Picnic coprocessor. Therefore, the changes introduced in the specification update should be implemented in the near future.

An additional variant of Picnic, namely Picnic2 [14], was developed to reduce the size of the signature. However, this new variant uses a different proof system, where 64 players perform an MPC-calculation. An efficient VHDL implementation of Picnic2, therefore, requires at least 63 LowMC instances which would not fit on any FPGA. As a result, any implementation of Picnic2 would have to calculate the shares of the 64 players consecutively increasing the runtime of signing and signature verification drastically. A future work, therefore, could be to design an efficient Picnic2 coprocessor which has an acceptable runtime/hardware utilization tradeoff.

Another possible future work would be to try to optimize the design and therefore, reduce the high hardware utilization. This could also improve the timing of the synthesized designs and thus, enable the coprocessor to run with higher frequencies.

The current design also lacks an evaluation of possible side-channel attacks. One possible future work will be to analyze our coprocessor and to deduce if it is possible to extract any secret value over a side channel and if the design requires additional protection.

Bibliography

[1] Gorjan Alagic et al. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*. 2019. DOI: 10.6028/NIST.IR.8240. URL: https://doi.org/10.6028/NIST.IR.8240 (cit. on pp. 2, 3, 45).

[2] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. "Ciphers for MPC and FHE." In: *EUROCRYPT (1)*. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 430–454 (cit. on pp. 10–12).

[3] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Patrick Longa, and Jefferson E. Ricardini. "The Lattice-Based Digital Signature Scheme qTESLA." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 85 (cit. on p. 2).

[4] Dorian Amiet, Andreas Curiger, and Paul Zbinden. "Flexible FPGA-Based Architectures for Curve Point Multiplication over GF(p)." In: *DSD*. IEEE Computer Society, 2016, pp. 107–114 (cit. on pp. 64, 65).

[5] Dorian Amiet, Andreas Curiger, and Paul Zbinden. "FPGA-based Accelerator for Post-Quantum Signature Scheme SPHINCS-256." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.1 (2018), pp. 18–39 (cit. on pp. 3, 64, 65).

[6] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. "BLAKE2: Simpler, Smaller, Fast as MD5." In: *ACNS*. Vol. 7954. Lecture Notes in Computer Science. Springer, 2013, pp. 119–135 (cit. on p. 65).

[7] Daniel J. Bernstein et al. "SPHINCS: Practical Stateless Hash-Based Signatures." In: *EUROCRYPT (1)*. Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 368–397 (cit. on p. 2).

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. *The Keccak reference*. Submission to NIST (Round 3). 2011. URL: http://keccak.noekeon.org/Keccak-reference-3.0.pdf (cit. on pp. 12, 13).

[9] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. *The sponge construction of Keccak*. 2008. URL: https://keccak.team/sponge_duplex.html (visited on 09/28/2017) (cit. on p. 13).

[10] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. "Random Oracles in a Quantum World." In: *ASIACRYPT*. Vol. 7073. Lecture Notes in Computer Science. Springer, 2011, pp. 41–69 (cit. on p. 14).

[11] 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: *ACM Conference on Computer and Communications Security*. ACM, pp. 1825–1842 (cit. on pp. 2, 8–10).

[12] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. *The Picnic Signature Algorithm Specification (2017)*. URL: https://github.com/microsoft/Picnic/blob/master/spec/spec-v1.0.pdf (cit. on pp. 14, 16, 18–22, 41).

[13] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. *The Picnic Signature Algorithm Specification (2018)*. URL: https://github.com/microsoft/Picnic/blob/master/spec/spec-v2.0.pdf (cit. on p. 67).

[14] Melissa Chase et al. *The Picnic Signature Scheme Design Document (2019)*. URL: https://github.com/microsoft/Picnic/blob/master/spec/design-v2.0.pdf (cit. on pp. 62, 63, 67).

[15] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. "From 5-Pass *MQ* -Based Identification to *MQ* -Based Signatures." In: *ASIACRYPT (2)*. Vol. 10032. Lecture Notes in Computer Science. 2016, pp. 135–165 (cit. on p. 2).

[16] Itai Dinur, Daniel Kales, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. "Linear Equivalence of Block Ciphers with Partial Non-Linear Layers: Application to LowMC." In: *EUROCRYPT (1)*. Vol. 11476. Lecture Notes in Computer Science. Springer, 2019, pp. 343–372 (cit. on pp. 23–27, 66).

[17] Itai Dinur, Yunwen Liu, Willi Meier, and Qingju Wang. "Optimized Interpolation Attacks on LowMC." In: *ASIACRYPT (2)*. Vol. 9453. Lecture Notes in Computer Science. Springer, 2015, pp. 535–560 (cit. on p. 10).

[18] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. "Higher-Order Cryptanalysis of LowMC." In: *ICISC*. Vol. 9558. Lecture Notes in Computer Science. Springer, 2015, pp. 87–101 (cit. on p. 10).

[19] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. "Security of the Fiat-Shamir Transformation in the Quantum Random-Oracle Model." In: *CoRR* abs/1902.07556 (2019) (cit. on p. 14).

[20] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.1 (2018), pp. 238–268 (cit. on p. 2).

[21] EU. *Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC*. 2014. URL: `https://eur-lex.europa.eu/eli/reg/2014/910/oj` (visited on 05/29/2019) (cit. on p. 1).

[22] A Ferozpuri, F Farahmand, MU Sharif, JP Kaps, and K Gaj. *Hardware API for Post-Quantum Public Key Cryptosystems*. 2016. URL: `https://cryptography.gmu.edu/athena/PQC/PQC_HW_API.pdf` (cit. on p. 45).

[23] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems." In: *CRYPTO*. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194 (cit. on p. 6).

[24] 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. 8).

[25] Daniel J Bernstein. "ChaCha, a variant of Salsa20." In: (Jan. 2008) (cit. on p. 65).

[26] LowMC. *Official LowMC Github Repository*. URL: https://github.com/LowMC/lowmc (visited on 08/27/2018) (cit. on p. 10).

[27] NIST. *NIST Post-Quantum Cryptography*. 2008. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography (visited on 05/21/2019) (cit. on p. 2).

[28] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS Publication 202. National Institute of Standards and Technology, U.S. Department of Commerce, Aug. 2015. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf (cit. on p. 12).

[29] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. "Enhanced Lattice-Based Signatures on Reconfigurable Hardware." In: *CHES*. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 353–370 (cit. on pp. 64, 65).

[30] Christian Rechberger, Hadi Soleimany, and Tyge Tiessen. "Cryptanalysis of Low-Data Instances of Full LowMCv2." In: *IACR Trans. Symmetric Cryptol.* 2018.3 (2018), pp. 163–181 (cit. on p. 10).

[31] Ismail San and Nuray At. "Improving the computational efficiency of modular operations for embedded systems." In: *Journal of Systems Architecture - Embedded Systems Design* 60.5 (2014), pp. 440–451 (cit. on pp. 64, 65).

[32] Claus-Peter Schnorr. "Efficient Identification and Signatures for Smart Cards (Abstract)." In: *EUROCRYPT*. Vol. 434. Lecture Notes in Computer Science. Springer, 1989, pp. 688–689 (cit. on p. 7).

[33] Claus-Peter Schnorr. "Efficient Signature Generation by Smart Cards." In: *J. Cryptology* 4.3 (1991), pp. 161–174 (cit. on p. 7).

[34] Yannick Seurin. "On the Exact Security of Schnorr-Type Signatures in the Random Oracle Model." In: *EUROCRYPT*. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 554–571 (cit. on p. 7).

[35] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." In: *FOCS*. IEEE Computer Society, 1994, pp. 124–134 (cit. on p. 2).

[36] Dominique Unruh. "Quantum Proofs of Knowledge." In: *EUROCRYPT*. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 135–152 (cit. on p. 14).

[37] Roman Walch. "Design and Implementation of a LowMC coprocessor." Master Project at Graz Univerity of Technology. 2018 (cit. on pp. 10, 23, 45).

[38] Xilinx. *DMA Subsystem for PCI Express v2.0 Product Guide*. URL: https://www.xilinx.com/support/documentation/ip_documentation/xdma/v2_0/pg195-pcie-dma.pdf (visited on 08/27/2018) (cit. on p. 51).

[39] Xilinx. *Xilinx Kintex-7 FPGA KC705 Evaluation Kit*. URL: https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html (visited on 08/17/2018) (cit. on p. 50).

[40] Xilinx. *Xilinx PCI Express DMA Driver*. URL: https://www.xilinx.com/support/answers/65444.html (visited on 08/27/2018) (cit. on p. 54).

[41] Xilinx. *Xilinx Vivado*. URL: https://www.xilinx.com/products/design-tools/vivado.html (visited on 08/17/2018) (cit. on p. 50).