Graz University of Technology

Stefan Steinegger, BSc

Side-Channel Attacks and Countermeasures for Lattice-Based Signatures

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Stefan Mangard

Institute for Applied Information

Processing and Communications (IAIK)

Graz, May 2018

AFFIDAVIT

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

Date	Signature

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

Acknowledgements

First, I would like to express my gratitude to my thesis advisors Peter Pessl and Stefan Mangard of the Institute of Applied Information Processing and Communications (IAIK) at Graz University of Technology. Whenever I had any questions, they offered their valuable assistance and pointed me in the right direction, while still allowing the thesis to be my own. Their continuous support, knowledge and guidance helped me to accomplish this thesis.
I would also like to thank Robert for his advice and comments on my work and the insightful discussions.

I would like to thank my parents Elisabeth and Richard for their unfailing support throughout the years of my studies and throughout my life.
Further, I would like to express my profound gratitude to Sabine for her unlimited support, motivation and encouragement throughout this thesis and my studies.
I also owe a special thanks to my grandparents Gertraud and Hermann for always encouraging my curiosity, their endless support and always believing in me.

Thank you.

Abstract

Over two decades ago Peter Shor presented an algorithm that allows conceptual quantum computers to factorise integers and to solve the discrete logarithm problem in polynomial time. Years later quantum computers were rendered possible. Since then they are becoming more and more powerful and an increasing problem for today's cryptographic systems.

While this might render conventional asymmetric cryptosystems obsolete in the future, it also is a chance for the development of new cryptographic primitives created with quantum computers in mind. An example is BLISS, a family of lattice-based signature schemes developed by Ducas et al.. Algorithms based on lattices are secure against quantum computing and offer a level of security and performance very similar to current state-of-the-art schemes based on RSA or ECC. They are therefore a promising replacement for conventional schemes. As a matter of fact, BLISS is already available in the strongSwan VPN solution. Additionally, Pöppelmann et al. demonstrated a hardware implementation of BLISS designed for the Spartan 6 FPGA family, further indicating the practicability of the algorithm.

However, due to lattice-based schemes being a very recent topic, little thought has been given on implementation security. Still, side-channel attacks targeting the polynomial multiplication or the sampler used in BLISS have already been proposed. They utilise power and cache side-channels and target software implementations. However, other parts of the algorithm and also the hardware implementation have not received much attention.

Consequently, this thesis aims at exploring the capabilities of side-channel attacks on the hardware implementation of the BLISS signature scheme. The sparse multiplication has been identified as an ideal target for the attack due to the operation representing a step directly involving the secret key during the signature creation. Using means of differential power analysis and simple power analysis, the process of recovering the secret key is evaluated. It is demonstrated that the multiplication shows a significant leakage when loading the secret key. Additionally, the leakage is time-invariant and allows to extract more information from each trace. Using this information, a clustering-based attack is built that reveals the secret key using 36 power traces. By also allowing for a number of wrongly detected key elements and additional post-processing, the attack can be improved to only require four traces for a successful key-recovery.

The success of the attack raises the need for a countermeasure. Several potential methods based on shuffling, hiding and masking are discussed. A countermeasure based on additive masking of the secret key is then selected as the most suitable method and added to the existing design. To show its effectiveness, the same attack is executed on the protected implementation. The results showed no leakage with the protected design, indicating the effectiveness of the countermeasure.

Keywords: BLISS Signature Scheme, Differential Power Analysis, DPA, Simple Power Analysis, Clustering-based Attack, Shuffling, Hiding, Masking, Side-Channel Attack, Post-Quantum Cryptography, Reconfigurable Hardware, FPGA

Kurzfassung

Vor mehr als zwei Jahrzehnten hat Peter Shor einen Algorithmus vorgestellt, der es konzeptuellen Quantencomputern erlaubt in polynomieller Zeit, Zahlen zu faktorisieren und das diskrete Logarithmus-Problem zu lösen. Jahre später wurden Quantencomputer ermöglicht. Seitdem werden sie immer leistungsfähiger und stellen ein zunehmendes Problem für heutige kryptographische Systeme dar.

Während das konventionelle asymmetrische Kryptographie in Zukunft womöglich obsolet macht, bietet es auch eine Chance für die Entwicklung von neuen kryptographischen Primitiven, die mit Blick auf Quantencomputer erstellt werden. Ein Beispiel ist BLISS, eine Familie von auf mathematischen Gittern basierenden Signaturschemen, die von Ducas et al. entwickelt wurden. Algorithmen die auf Gittern basieren sind sicher gegenüber Quantencomputern und bieten ein Niveau an Sicherheit und an Leistungsfähigkeit das sehr ähnlich zu modernen Systemen, basierend auf ECC und RSA, ist. Sie sind daher ein vielversprechender Ersatz für konventionelle Systeme. In der Tat ist BLISS bereits Bestandteil der strongSwan VPN Lösung. Zusätzlich haben Pöppelmann et al. eine Hardwareimplementierung von BLISS für die Spartan 6 FPGA Familie gezeigt, was einen weiteren Hinweis auf die Praktikabilität des Algorithmus gibt.

Da gitterbasierende Schemen ein sehr neues Thema sind, wurde der Implementierungssicherheit nur wenig Aufmerksamkeit geschenkt. Dennoch wurden Seitenkanalattacken auf die Polynommultiplikation und den Sampler in BLISS vorgeschlagen. Sie verwenden Leistungs- und Cache-Seitenkanäle und zielen auf Software-Implementierungen ab. Andere Teile des Algorithmus sowie die Hardware-Implementierung wurden noch nicht eingängig untersucht.

Folglich zielt diese Arbeit darauf ab, die Möglichkeiten von Seitenkanalattacken auf die Hardware-Implementierung des BLISS Signaturschemas zu erforschen. Die Multiplikation mit einem dünnbesetzten Vektor wurde als idealer Angriffspunkt bestimmt, da während der Signaturerstellung dieser Schritt den geheimen Schlüssel direkt involviert. Mit Mitteln der Differential Power Analysis und der Simple Power Analysis wird das Auslesen des geheimen Schlüssels evaluiert. Es wird gezeigt, dass das Laden des geheimen Schlüssels, während der Multiplikation, Informationen über diesen liefert. Außerdem ist diese Leistungsaufnahme zeitinvariant, was es ermöglicht mehr Information aus jeder Messung auszulesen. Mit Hilfe dieser Information wird ein Clustering-basierter-Angriff erstellt, der den geheimen Schlüssel mit 36 Messungen bestimmt. Durch das zusätzliche Erlauben einer gewissen Anzahl an falsch klassifizierten Elementen des geheimen Schlüssels und weiterer Nachbearbeitung, kann diese Attacke verbessert werden und benötigt nur mehr vier Messungen um den Schlüssel erfolgreich zu ermitteln.

Der Erfolg der Attacke zeigt den Bedarf für eine Gegenmaßnahme. Mehrere potentielle Methoden basierend auf Shuffling, Hiding und Maskierung werden diskutiert. Eine Gegenmaßnahme basierend auf additiver Maskierung des geheimen Schlüssels wird dann als die vielversprechendste Methode ausgewählt und zum bestehenden Design hinzugefügt. Um die Effektivität zu zeigen wird derselbe Angriff auf die geschützte Implementierung durchgeführt. Das Ergebnis zeigt, dass die Leistungsaufnahme der geschützten Implementierung keinen Aufschluss über den geheimen Schlüssel zulässt, was die Effektivität der Gegenmaßnahme zeigt.

Stichwörter: BLISS Signaturschema, Differential Power Analysis, DPA, Simple Power Analysis, Clustering Attacke, Shuffling, Hiding, Maskierung, Seitenkanalattacke, Post-Quanten-Kryptographie, Programmierbare Hardware, FPGA

Contents

List of Figures

Chapter 1

Introduction

State-of-the-art asymmetric cryptographic schemes such as RSA (Rivest – Shamir – Adleman) [45] and ECC (Elliptic-curve cryptography) [22, 32] rely on mathematical problems like the factorisation problem and the discrete logarithm problem, respectively. Although these primitives have been around for decades, in terms of widespread usage, they still gain popularity due to, among other things, browser vendors pushing for an encrypted internet [58, 49, 50]. The underlying factorisation and discrete logarithm problem of RSA and ECC, if used with integers sufficiently large, are infeasible to solve for today's conventional computers.

However, despite their popularity, these schemes become threatened by the increasing capabilities of quantum computers [12, 33]. As a matter of fact, in 1994 Peter Shor [52] presented a quantum algorithm capable of factoring integers and finding discrete logarithms. Later, in 1997, he even showed an algorithm solving these problems in polynomial time [53]. However, in both cases, a quantum computer is necessary to execute the algorithms.

With the progress made in building quantum computers [10, 12, 26], it is necessary to start finding viable alternatives that can be used in such a post-quantum scenario and ensure security in the long-term. In 2016 the National Institute of Standards and Technology (NIST) put out a call for proposals for post-quantum cryptography [55]. Additionally, the NSA is researching quantum computers as well as cryptography secure against quantum computers [44, 51], further indicating the need for new types of cryptosystems.

Cryptographic schemes based on lattices [2] pose a promising candidate for quantum secure cryptosystems. While the initial proposals were unsuited for real-life applications due to impractical key-sizes [31], these downsides were improved in the following years [29, 27] and lattice-based schemes became more practical. First tests of a lattice-based key exchange called *A New Hope* [3] in browsers in 2016 further showed the practicability of lattice-based cryptography [8].

Regarding digital signatures, Ducas et al. [15] developed a scheme called BLISS. It shows promising results and comparable performance to state-of-the-art implementations of asymmetric cryptography. An improved variant called BLISS-B is even used in StrongSwan's VPN solution [37].

While there already is some research on attacks and countermeasure for implementations of the BLISS signature scheme [17, 37, 47, 36], implementations for reconfigurable hardware [39] have stayed mostly unexplored. For this reason, this thesis explores the capabilities of side channel-attacks on an FPGA implementation of the BLISS signature scheme. After analysis of possible weak points of the BLISS algorithm, the sparse multiplication is selected as a suitable target to recover the secret key. Because the sparse

multiplication is also present in other lattice-based schemes, a successful attack on this operation also affects other schemes. For the analysis of the multiplication, the implementation of Pöppelmann et al. [39] is modified such that the FPGA can be controlled with a PC. The implementation can then be used as the target device to evaluate the power side-channel leakage. For this purpose, the BLISS algorithm on the FPGA is used to compute signatures while its power consumption is measured.

After some initial pre-processing it is first shown that the sparse multiplication shows a data-dependent power leakage. Using means of differential power analysis, it is then further shown that this leakage is time-invariant, which allows for extracting more information from each power trace. The observed leakage is then used to build a clustering-based attack capable of recovering the secret key. Moreover, it is explored how the number of traces can be reduced by allowing for a certain amount of wrongly classified key elements. This is achieved by combining the attack with a method proposed by Pessl et al. [37].

The success of the attack shows the need for an effective countermeasure. Therefore, countermeasures based on hiding, shuffling and masking are discussed regarding their effectiveness and how well they are suited for hardware implementations. A countermeasure based on masking is then used to protect the hardware implementation. To evaluate its effectiveness, the protected operation is evaluated using the same methods based on differential power analysis and a clustering-based attack. The results are discussed to establish the effectiveness of the countermeasure compared to the unprotected implementation.

Organisation

First, in Chapter 2 a general definition of lattices is given. Furthermore, the various types of lattice problems and lattice-related problems are explained as well as reasons for using rings when operating with lattices.

Next, an overview of a type of signature algorithms called BLISS is given in Chapter 3. Besides presenting the different configurations of the scheme, the algorithm to create the keys, the signature and also how to verify an existing signature are given. More detail is given on the sparse multiplication, which represents an essential step during the signing process. The advantages that come with it, its performance and the hardware implementation are discussed.

Chapter 4 gives an in-depth description of the fundamentals of side-channel attacks and ways to counteract them. Furthermore, this chapter also covers existing attacks and countermeasures regarding lattice-based cryptography found in the literature.

Starting with Chapter 5 the practical aspect of this thesis is described. First, the way the hardware implementation of the BLISS signature scheme by Pöppelmann et al. [39] is extended to enable the communication between a PC and the algorithm on an FPGA. Also, a description of the measurement setup to acquire power traces of the FPGA and the pre-processing of these traces is given. It is further shown why the sparse multiplication is a suitable target for the evaluation. Additionally, the evaluation of the power side-channel is discussed and a description of the attack to recover the secret key of the scheme is given.

In Chapter 6 the practical part is continued albeit with the motivation to protect the secret key instead of revealing it. Thus numerous means to protect the design are discussed and evaluated regarding their use in the hardware design. The most fitting countermeasure is picked. Its integration in the existing design and the effect on metrics like area and

throughput are then discussed. To elaborate the quality of the method, another attempt to exploit the power side-channel is made.

The results of the experiments are then summarised in Chapter 7. Additionally, ideas for future works and for related attacks on the BLISS scheme are given.

Chapter 2

Lattice-based Cryptography

This chapter discusses lattices and some of their underlying problems utilised in modern post-quantum cryptography. With early references to lattices being dated back to the 18th century, the mathematical problems have become increasingly popular over the last years due to the efforts of finding post-quantum secure cryptographic schemes. Section 2.1 defines fundamental properties of lattices and lattice-based problems. This is then elaborated further by discussing the mathematical problems defined on lattices, like the *Short Integer Solution problem* in Section 2.2 or the *Ring-Short Integer Solution problem* in Section 2.3. Both are used to base more modern schemes on. Characteristics making a lattice an ideal lattice are described in Section 2.3.1 and advantages of basing lattices on a ring to perform more efficient computations are given in Section 2.3.2.

2.1 Lattices

In general, the term lattice refers to a set of regular discrete points in an n-dimensional space. This leads to the definition:

Definition 1. *Given an ordered basis* $\mathbf{B} = (\mathbf{b}_1, ..., \mathbf{b}_n)$ *with* n*-linearly independent column vectors* $\mathbf{b}_i \in \mathbb{R}^n$*. The set* $\mathbf{B}$ *is referred to as the basis for the lattice [30]. Any point on the lattice can be expressed by an integer combination of the vectors of the basis. A lattice* $\mathcal{L}$ *is then defined as*

$$\mathcal{L} = \mathcal{L}(\mathbf{B}) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{Z} \right\} \tag{2.1}$$

A particular type of lattices are q-ary lattices. It is a commonly used type of lattices for lattice-based cryptography. For a q-ary lattice a vector $\mathbf{x}$ is only part of the given lattice iff the vector $\mathbf{x} \bmod q$ is also part of the same lattice, thus satisfying $q\mathbb{Z}^n \subseteq \mathcal{L} \subseteq \mathbb{Z}^n$ for some integer q [30].
Furthermore, any lattice can be represented by infinitely many different bases. A simple example of a two-dimensional lattice with two different bases can be seen in Figure 2.1.

2.1.1 Lattice-based Problems

Cryptographic schemes based on lattices build on hard to solve mathematical problems defined on lattices. Problems that are related to lattices, but are not directly based on them, are traced back to these as well to show that they are at least as difficult to solve. Next, two such lattice problems are stated.

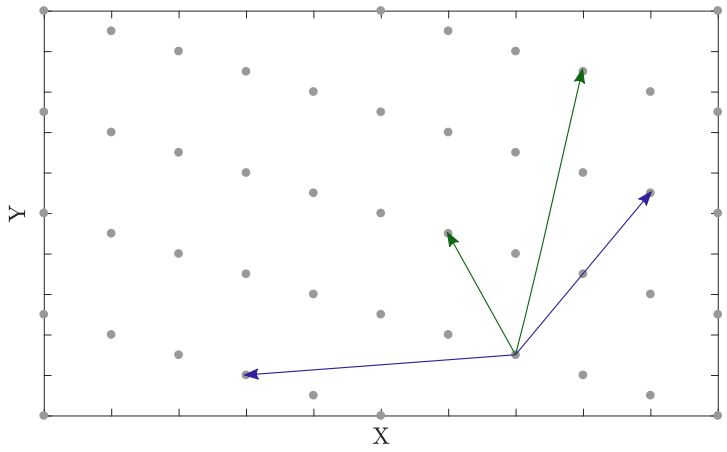


Figure 2.1: Example of a two-dimensional lattice, with two different bases given in blue and green.

Shortest Vector Problem

The Shortest Vector Problem (SVP) is one of the most commonly used lattice problems in cryptography and more than 250 years old [1].

With $|| \cdot ||$ denoting the Euclidean norm, the basic task is to find a shortest vector or its length in a given lattice. In other words: Given a lattice $\mathcal{L}$ defined over its basis $\mathbf{B}$ the task is to find the shortest, but non-zero, vector $\mathbf{x}$ based on an integer combination of the basis vectors $\mathbf{b}_i$. This minimum distance within the lattice is defined as λ [35]:

$$\lambda(\mathcal{L}) := \min_{\mathbf{x} \in \mathcal{L} \setminus \{\mathbf{0}\}} ||\mathbf{x}|| \tag{2.2}$$

The SVP has an exact and an approximate form. The exact SVP requires to find a vector $\mathbf{x} \in \mathcal{L}(\mathbf{B})$ to which applies:

$$||\mathbf{x}|| = \lambda(\mathcal{L}) \tag{2.3}$$

A variant of this problem is the approximate or γ-approximate SVP. In contrast to the exact version it aims at finding a short, non-zero, vector $\mathbf{x}$ which is smaller than $\lambda(\mathcal{L})$ multiplied with a specified factor $\gamma \geq 1$:

$$||\mathbf{x}|| \leq \gamma\lambda(\mathcal{L}) \tag{2.4}$$

A further variation of this problem is the so-called GapSVP. While the previous problems required to determine the shortest vector of the lattice, the GapSVP is a (binary) decision-based approximation of the underlying issue. Given the lattice $\mathcal{L}$ the goal is to find if the shortest vector $\mathbf{x}$ is shorter than one ($\lambda(\mathcal{L}) \leq 1$) or longer than a given approximation factor γ ($\lambda(\mathcal{L}) \geq \gamma$).

Approximate Shortest Independent Vectors Problem

The Approximate Shortest Independent Vectors Problem (SIVP) is very similar to the approximate SVP. While the latter requires finding a single vector $\mathbf{x}$ that is within a

margin γ, the approximate SIVP aims a finding a set of n linearly independent vectors $||\mathbf{x}_i|| \leq \gamma \cdot \lambda(\mathcal{L})$ for $i \in \{1, 2, ..., n\}$.

2.2 Short Integer Solution Problem

The Short Integer Solution (SIS) problem is commonly used for cryptographic schemes. There exist hash functions, identification- and signature schemes based on the SIS problem [35]. It was first described by Ajtai [1] in 1996. Unlike the lattice-based problems described in Section 2.1.1, the SIS problem is not directly based on a lattice problem but can be reduced to the GapSVP or the approximate SIVP problem [35].

Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ with column vectors $(\mathbf{a}_1, ..., \mathbf{a}_m)$ the SIS problem asks to find a *non-zero* integer vector $\mathbf{x} \in \mathbb{Z}^m$ with an Euclidean norm $||\mathbf{x}|| \leq \beta < q$, that satisfies:

$$\mathbf{Ax} = \sum_{i=1}^{m} \mathbf{a}_i \cdot x_i = 0 \in \mathbb{Z}_q^n \tag{2.5}$$

Without the further constraint of limiting the Euclidean norm of $\mathbf{x}$ to be smaller than β, it would be easy to solve this problem using Gaussian elimination. Similarly, the zero-vector is excluded from this problem as it would always be a solution to the equation. Furthermore, it is essential to require $\beta < q$, because otherwise, it allows for the solution $\mathbf{x} = (q, 0, 0, ..., 0) \in \mathbb{Z}^m$. This problem can be shown to be as hard as the approximate GapSVP or the approximate SIVP if the parameters are chosen reasonably [35].

2.3 Ring-Short Integer Solution Problem

This section describes the ring-version of SIS ($\mathcal{R}$-SIS) and how it can be used to improve the efficiency of cryptographic algorithms.

The $\mathcal{R}$-SIS problem is defined over a polynomial ring. A very commonly used ring is $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$. The $\mathcal{R}$-SIS problem is then defined as follows: Given the uniformly random vector $\mathbf{a} \in \mathcal{R}_q^m$ with elements $(a_1, ..., a_m) \in \mathcal{R}_q$ the $\mathcal{R}$-SIS problem asks to find a non-zero vector $\mathbf{x} \in \mathcal{R}_q^m$, with $||\mathbf{x}|| \leq \beta$ and the same number of elements as $\mathbf{a}$ fulfilling the equation

$$\sum_{i=1}^{m} a_i \cdot x_i = 0 \in \mathcal{R}_q \tag{2.6}$$

Overall the $\mathcal{R}$-SIS is similar to the SIS problem with the difference being the defined space of the occurring vectors. While the SIS solution utilises elements from $\mathbb{Z}_q$, the $\mathcal{R}$-SIS solution uses elements from the ring $\mathcal{R}_q$. Due to the $\mathcal{R}$-SIS utilising a ring, it is only necessary to store the coefficients of the polynomial as a vector, reducing the memory requirement. Furthermore, mathematical operations over a ring implemented in hardware require less area on the chip and tend to compute the result faster.

While the use of rings already offers several advantages and improvements in practicability, the usage of rings allows for further improvements by using ideal lattices.

2.3.1 Ideal Lattices

In case of the SIS problem, the basis $\mathbf{B}$ can be written as a matrix, requiring $\mathcal{O}(n^2)$ memory. The quadratic growth and an increasingly large security parameter n make the

8	1	9	3
5	2	7	5
3	5	4	8
2	6	8	3

6	-1	-4	-3
3	6	-1	-4
4	3	6	-1
1	4	3	6

Figure 2.2: An example of a regular lattice basis on the left and an ideal lattice basis based on a polynomial ring on the right. The second to fourth column are basically the first column rotated with a flipped sign.

scheme less suited for practical usage. This issue can be tackled by using ideal lattices, a more generalised form of cyclic lattices and $\mathcal{R}$-SIS as the underlying problem. An ideal lattice is defined over some ring $\mathbb{Z}[x]/(f)$ with f being an irreducible polynomial of degree n. An example of a commonly used polynomial f is $f(x) = x^n + 1$, with n being a power of two and a prime q such that $q \equiv 1 \bmod 2n$, resulting in the ring $\mathcal{R}_q = \mathbb{Z}_q/(x^n + 1)$ [41].

In practice, this removes the need to store the full sized $n \times n$ matrix, and instead, it suffices to only save the first column. The other columns can then be computed on the fly based on rotations of the first one, fully describing the matrix as seen in Figure 2.2 [30]. This reduces the memory consumption to $\mathcal{O}(n)$, i.e., growing only linearly.

2.3.2 Number Theoretic Transform

While additions in the ring $\mathcal{R}_q = \mathbb{Z}_q/(x^n + 1)$ can already be performed efficiently, coefficient-wise with a complexity of $\mathcal{O}(n)$, multiplications in this ring still have a complexity of $\mathcal{O}(n^2)$ when using the *schoolbook method*. The computation of the product of the two polynomials $\mathbf{a} \cdot \mathbf{b}$ over $\mathcal{R}_q$ is then performed by computing:

$$\mathbf{a} \cdot \mathbf{b} = \left[\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j} \right] \bmod (x^n + 1) \tag{2.7}$$

This approach requires n^2 multiplications and $(n-1)^2$ additions or subtractions. Algorithms like the Karatsuba method [21] reduce this complexity to $\mathcal{O}(n^{\log(3)})$, but due to their recursive structure, they are not suitable for hardware implementations [41].

However, the multiplication in $\mathcal{R}_q$ can be even more efficiently implemented using a Number Theoretic Transform (NTT) which is very similar to a Fast Fourier Transform (FFT) defined over a finite field or ring [41]. While the FFT would require floating point operations and complex arithmetic, the NTT is defined over $\mathbb{Z}_q$, eliminating the need for this kind of operations.

Performing the NTT on a given polynomial requires the primitive n-th root of unity ω to exist. An integer qualifies as the primitive n-th root of unity ω modulo q iff $\omega^n \equiv 1 \bmod q$ and $\omega^{n/p} - 1 \not\equiv 0 \bmod q$ for *any* prime divisor p of n [4]. Choosing the ring $\mathcal{R}_q$ with $q \equiv 1 \bmod 2n$ and n a power of two ensures that ω exists.

The parameter ω can then be used to calculate the NTT of a polynomial $\mathbf{a}$:

$$\hat{\mathbf{a}} = \mathrm{NTT}(\mathbf{a}) := \hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \bmod q, i = \{0, 1, ..., n-1\} \tag{2.8}$$

To inverse the calculation using the iNTT and receive the polynomial $\mathbf{a}$:

$$\mathbf{a} = \text{iNTT}(\hat{\mathbf{a}}) := a_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{a}_j \omega^{-ij} \bmod q, i = \{0, 1, ..., n-1\} \tag{2.9}$$

To perform the multiplication $\mathbf{c} = \mathbf{a} \cdot \mathbf{b} \in \mathcal{R}_q$, both factors need to be transformed using the NTT, then component-wise multiplied (indicated by $\odot$), before using the iNTT to get the result:

$$\mathbf{c} = \mathbf{a} \cdot \mathbf{b} = \text{iNTT}(\text{NTT}(\mathbf{a}) \odot \text{NTT}(\mathbf{b})) \tag{2.10}$$

Algorithm 2.1, from [41], shows an iterative approach to perform the NTT operation. It takes a polynomial $\mathbf{g} \in \mathcal{R}_q$ with a degree smaller than n and with a primitive n-th root of unity ω. The `Bit-Reverse` function refers to a mathematical bit-reversal permutation. It uses the bit representation of each index and reverses its binary representation to determine the new location of the value, effectively reordering all values. The algorithm has a complexity of $\mathcal{O}(n \log(n))$ and is suited for hardware implementations. The algorithm uses a butterfly network similar to the FFT, which can be seen in line 13f. of Algorithm 2.1

Algorithm 2.1 Iterative algorithm computing the NTT, based on the Cooley and Turkey radix-2 decimation according to Pöppelmann and Güneysu [41]

1: **procedure** NTT(Polynomial $\mathbf{g} \in \mathcal{R}_q$)
2: $A \leftarrow$ Bit-Reverse($\mathbf{g}$)
3: $m \leftarrow 2$
4: **while** $m \leq N$ **do**
5: $s \leftarrow 0$
6: **while** $s < N$ **do**
7: **for all** i **to** $m/2 - 1$ **do**
8: $N \leftarrow i \cdot n/m$
9: $a \leftarrow s + i$
10: $b \leftarrow s + i + m/2$
11: $c \leftarrow A[a]$
12: $d \leftarrow A[b]$
13: $A[a] \leftarrow c + \omega^{N \bmod n} d \bmod q$
14: $A[b] \leftarrow c - \omega^{N \bmod n} d \bmod q$
15: $s \leftarrow s + m$
16: $m \leftarrow m \cdot 2$
17: **return** A

Chapter 3

BLISS Signature Family

This chapter describes a family of lattice-based signature schemes called BLISS (**B**imodal **L**att**i**ce **S**ignature **S**cheme) proposed by Ducas et al. [15], which is based on the $\mathcal{R}$-SIS problem and utilises a bimodal Gaussian distribution for rejection sampling. Section 3.1 details different possible parameter sets of the BLISS algorithm. Section 3.2.1 explains how a key is generated. Then, Section 3.2.2 shows the algorithm that uses the key to create a signature. In Section 3.2.3 the verification process is illustrated. Section 3.3 goes into detail with the sparse multiplication, an essential step during the signature process, directly interacting with the secret key. Furthermore, the performance of algorithms of the BLISS-family implemented in hard- and software is compared to other state-of-the-art cryptographic primitives in Section 3.4. Finally, in Section 3.5 details on the hardware implementations are discussed, with a focus on the BLISS signing algorithm.

3.1 Parameter Sets

The BLISS signature family allows for changing its parameters, tuning the algorithm in terms of security level, signature size and speed. Ducas et al. [15] propose five parameter sets named BLISS-0 to BLISS-IV shown in Table 3.1. BLISS-0 is considered a toy example and challenge to improve attacks on lattice-based cryptography. BLISS-I and BLISS-II use similar parameter sets and offer the same 128 bits level of security. BLISS-I is optimised for speed, with fewer repetitions per signature, whereas BLISS-II is optimised for size, resulting in smaller signatures. BLISS-III and BLISS-IV each represent more secure parameter sets, with 160 bits and 192 bits of security respectively at the cost of larger signatures.
Throughout this work, the BLISS-I parametrisation will be used.

3.2 Algorithms

In this section, the algorithms proposed by Ducas et al. [15] for key generation, creating and verifying signatures are discussed.

3.2.1 Key Generation

Key generation for BLISS is done by generating two random polynomials $\mathbf{f}, \mathbf{g}$ based on the secret key densities δ_1, δ_2. Each polynomial has $d_1 = \lceil \delta_1 n \rceil$ coefficients from the set $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$ coefficients from the set $\{\pm 2\}$. The remaining coefficients are set to 0; this

Name	BLISS-0	BLISS-I	BLISS-II	BLISS-III	BLISS-IV
Security level	$\leq$ 60 bits	128 bits	128 bits	160 bits	192 bits
Optimisation	Fun	Speed	Size	Security	Security
n	256	512	512	512	512
Modulus q	7681	12289	12289	12289	12289
Secret key densities δ_1, δ_2	0.55, 0.15	0.3, 0	0.3, 0	0.42, 0.03	0.45, 0.06
Gaussian standard deviaten σ	100	215	107	250	271
κ	12	23	23	30	30
N_k-Threshold C	1.5	1.62	1.62	1.75	1.88
d dropped Bits in $\mathbf{z}_2$	5	10	10	9	8
Verification thresholds B_2, B_∞	2492,530	12872,2100	11074,1563	10206,1760	9901,1613
Repetition rate	7.4	1.6	7.4	2.8	5.2
Signature size	3.3kb	5.6kb	5kb	6kb	6.5kb
Secret key size	1.5kb	2kb	2kb	3kb	3kb
Public key size	3.3kb	7kb	7kb	7kb	7kb

Table 3.1: Different parameter sets as published in [15]. The implementation used later in this thesis uses the parameters for BLISS-I, highlighted in grey.

is repeated until $\mathbf{f}$ is invertible. The secret key $\mathbf{S}$ is then calculated as

$$\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t = (\mathbf{f}, 2\mathbf{g} + 1)^t \tag{3.1}$$

Each set of parameters defines a value C that is used to compute a threshold setting the upper limit for the function $N_\kappa(\mathbf{S})$. The key is rejected if $N_\kappa(\mathbf{S}) \geq C^2 \cdot 5\ (d_1 + 4d_2) \cdot \kappa$, whereas $N_\kappa(\mathbf{S})$ is calculated by

$$N_\kappa(\mathbf{S}) = \max_{\substack{I \subset \{1,\ldots,n\} \\ \#I = \kappa}} \sum_{i \subset I} \left(\max_{\substack{J \subset \{1,\ldots,n\} \\ \#J = \kappa}} \sum_{j \subset J} T_{i,j} \right) \text{ with } \mathbf{T} = \mathbf{S}^t \cdot \mathbf{S} \subset \mathcal{R}^{n \times n} \tag{3.2}$$

There are further optimisations to Equation (3.2) described by Ducas et al. [15] to improve the speed of the calculation of the two nested sums. In practice the calculation of the $N_\kappa(\mathbf{S})$ value results in about 25% rejected keys, reducing the overall security by 2 bits. Additionally, the secret key is rejected if the polynomial $\mathbf{f}$ is not invertible, as it would prevent the calculation of $\mathbf{a}_q$ needed for the public key.

Given the private key $\mathbf{S}$, the public key $\mathbf{A}$ is then defined over the quotient ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ such that $\mathbf{AS} = q\mathbf{I}_n (\text{mod } 2q)$ with $\mathbf{I}$ being the identity matrix resulting in:

$$\mathbf{A} = (2\mathbf{a}_q, q - 2) = (2 \cdot {}^{(2\mathbf{g}+1)}/_{\mathbf{f}}, q - 2) \in \mathcal{R}_{2q}^{1 \times 2} \tag{3.3}$$

3.2.2 Signing

Signing a message μ is done by first sampling two random vectors $\mathbf{y}_1$, $\mathbf{y}_2$ from a discrete Gaussian distribution D with the specified standard deviation σ. The value for σ is given in Table 3.1.

Algorithm 3.1 BLISS key generation according to [15]

1: **procedure** KeyGen
2: Choose uniform polynomials $\mathbf{f}, \mathbf{g}$ with d_1 entries in $\{\pm 1\}$ and d_2 entries in $\{\pm 2\}$
3: $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t \leftarrow (\mathbf{f}, 2\mathbf{g} + 1)^t$
4: **if** $N_\kappa(\mathbf{S}) \geq C^2 \cdot 5 \cdot (\lceil \delta_2 n \rceil + 4\lceil \delta_2 n \rceil) \cdot \kappa$ **then restart**
5: **if** $\mathbf{f}$ not invertible **then restart**
6: $\mathbf{a}_q = (2\mathbf{g} + 1)/\mathbf{f} \bmod q$
7: **return**$(\mathbf{A} = (2\mathbf{a}_q, q - 2) \bmod 2q, \mathbf{S})$

Next, a polynomial multiplication of $\zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$ is performed. Due to the key-generation, the polynomial $\mathbf{a}_1$ is created over the ring $\mathbb{Z}_{2q}[x]/(x^n + 1)$. At first glance, this makes it harder to compute the product $\mathbf{a}_1\mathbf{y}_1$, because the multiplication using the NTT from Section 2.3.2 is defined over the ring $\mathbb{Z}_q[x]/(x^n + 1)$. However by setting $\mathbf{a}_1 = 2 \cdot \mathbf{a}_1'$ it is possible to compute $\mathbf{a}_1' \cdot \mathbf{y}_1$ over $\mathbb{Z}_q[x]/(x^n + 1)$ and afterwards multiplying the coefficients of the result by 2 to get the result over the ring $\mathbb{Z}_{2q}[x]/(x^n + 1)$. By carrying out this additional step, the multiplication can be efficiently performed using the NTT.

The upper d bits of the result u are then used together with the message μ as input for the hash function H. The hash function needs to output a uniform vector $\mathbf{c}$ in $\mathbb{B}_\kappa^n$, i.e., a binary vector with κ elements being 1 and a total length of n, making it a sparse vector [15].

Next, a single random bit is used to decide if the product of the multiplication $\mathbf{s}_{1,2}\mathbf{c}$ is subtracted from or added to the previously generated vector $\mathbf{y}_{1,2}$.

In line 8 a rejection sampling is performed. This ensures that the distribution of the result of $\mathbf{z}$ follows the Gaussian distribution, prohibiting leakage on the secret key [39]. However, it should be noted that due to the rejection sampling, the signature creation is not performed in constant time.

Finally, a signature compression is carried out, effectively reducing the final signature size by dropping most of the low-order bits from $\mathbf{z}_2$ and in conjunction with $\mathbf{u}$ calculating the output $\mathbf{z}_2^\dagger$.

Algorithm 3.2 BLISS signature creation according to [15]

1: **procedure** Sign(Message μ, public key $\mathbf{A} = (\mathbf{a}_1, q - 2)$, secret key $\mathbf{S}$)
2: $\mathbf{y}_1, \mathbf{y}_2 \leftarrow D_{\mathbb{Z}^n, \sigma}$
3: $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
4: $\mathbf{c} \leftarrow H(\lfloor \mathbf{u} \rceil_d \bmod p, \mu)$
5: Choose random bit b
6: $\mathbf{z}_1 \leftarrow \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \mathbf{c}$
7: $\mathbf{z}_2 \leftarrow \mathbf{y}_2 + (-1)^b \mathbf{s}_2 \mathbf{c}$
8: **Continue** with probability $1/(M \exp(-\frac{\|\mathbf{Sc}\|^2}{2\sigma^2}) \cosh(\frac{<\mathbf{z}, \mathbf{Sc}>}{\sigma^2}))$ **Else** restart
9: $\mathbf{z}_2^\dagger \leftarrow (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
10: **return**$(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$

3.2.3 Verifying

To verify a signature, the norms of the signature are calculated and compared to the specified verification thresholds B_2, B_∞ from Table 3.1. Furthermore, the input of the

hash function H is reconstructed and the output compared to the vector $\mathbf{c}$. If $\mathbf{c}$ equals the result of the hash function the signature is accepted.

Algorithm 3.3 BLISS signature verification according to [15]

1: **procedure** VERIFY(Message μ, public key $\mathbf{A}$, signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$)
2: **if** $\|(\mathbf{z}_1|2^d \cdot \mathbf{z}_2^\dagger)\|_2 > B_2$ **then** Reject
3: **if** $\|(\mathbf{z}_1|2^d \cdot \mathbf{z}_2^\dagger)\|_i nfty > B_\infty$ **then** Reject
4: **if** $\mathbf{c} == H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$ **then**
5: Accept
6: **else**
7: Reject

3.3 Sparse Multiplication

In Algorithm 3.2 on line 6f., the secret key $\mathbf{S} = (\mathbf{s_1}, \mathbf{s_2})$ is multiplied by a vector $\mathbf{c}$ over the ring $\mathbb{Z}_{2q}[x]/(x^n + 1)$. The vector $\mathbf{c}$ has κ out of n entries set to one. The value for κ depends on the parameter set from Table 3.1 and ranges from 12 for BLISS-0 to 39 for BLISS-IV. With most of its entries being zero, $\mathbf{c}$ is a *sparse binary vector*. Furthermore, the number of non-zero elements in the key $\mathbf{s}_i$ is limited by the density parameters δ_1 and δ_2 during the key generation.

The multiplication $\mathbf{s}_i\mathbf{c}$ could be performed using the NTT. However, as $\mathbf{c}$ is a sparse binary vector and $\mathbf{s}_i$ is small it is much more efficient to perform a sparse multiplication and compute the result by additions over $\mathbb{Z}$ [39].

A further advantage is that due to the sparse vector $\mathbf{c}$ having a Hamming weight of κ it is sufficient to store only the κ indices of elements being 1.

3.4 Performance

A performance comparison between state-of-the-art signature schemes and algorithms of the BLISS-family, implemented in hard- and software, can be seen in Table 3.2. The metrics of the software implementation are based on an Intel Core i7 desktop computer, with RSA and ECC using OpenSSL 1.0.1c [15]. In terms of the hardware implementation, a Spartan 6 LX25-3 has been used for 1024 bit messages [39]. The hardware implementation allows for varying levels of parallelisation because for example the sparse multiplication can be calculated parallelised. On that matter, the hardware implementation tries to use a trade-off between speed and resource consumption by using 8 cores for the multiplication [39]. Thus 8 output coefficients of $\mathbf{s}_{1,2}$ are computed simultaneously. Furthermore, it should be noted, that the software implementation uses SHA-512 as the hash function, while the hardware implementation uses Keccak for this purpose. The number of sign- and verification operations per second in Table 3.2 of the hard- and software implementations should not be compared to each other in terms of their absolute values because they are optimised for different constraints. The hardware represents an efficient implementation on a low-cost FPGA [39] with very similar performance to a software implementation running on a state-of-the-art PC.

When comparing the implementations and schemes, the resulting signature sizes for a BLISS-signature are somewhat larger than RSA and much larger than ECDSA signatures.

Secret- and public key sizes are significantly larger than ECDSA keys at similar security levels as well. Comparing the key sizes to RSA at the same level, shows smaller secret key sizes favouring BLISS despite the larger public keys. In terms of speed, BLISS, with about 8-9k signatures per second for BLISS-I, is not performing quite as well as ECDSA with 9.5k signatures per seconds, but clearly outperforming RSA 2048 at 0.8k signatures per second. However, in terms of verification speed, the software implementation of BLISS-I manages to verify about 13 times as many signatures as the ECDSA implementation at an equivalent security level. Comparing BLISS-I with 33k verifications per second to RSA 2048 with 27k verifications per second still shows a significant difference.

Implementation	Security level bits	Signature size kb	Secret key size kb	Public key size kb	Sign/s	Verify/s
BLISS-0 (SW)	≤ 60	3.3	1.5	3.3	4k	59k
BLISS-I (SW)	128	5.6	2	7	8k	33k
BLISS-I (HW)					8.8k	17.1k
BLISS-II (SW)	128	5	2	7	2k	33k
BLISS-III (SW)	160	6	3	7	5k	32k
BLISS-III (HW)					4.8k	17.8k
BLISS-IV (SW)	192	6.5	3	7	2.5k	31k
BLISS-IV (HW)					2.8k	17.8k
RSA 2048 (SW)	103-112	2	2	2	0.8k	27k
RSA 4096 (SW)	≥ 128	4	4	4	0.1k	7.5k
ECDSA 160 (SW)	80	0.32	0.16	0.16	17k	5k
ECDSA 256 (SW)	128	0.5	0.25	0.25	9.5k	2.5k
ECDSA 384 (SW)	192	0.75	0.37	0.37	5k	1k

Table 3.2: Performance comparison of the BLISS signature family to state of the art signature algorithms. SW indicates a software implementations, HW indicates an FPGA hardware implementation on a Spartan 6 for 1024 bit messages and $C = 8$ cores for the sparse multiplication [39, p. 16f.]. Openssl 1.0.1c was used for RSA and ECSDA [15, p.2]

3.5 BLISS on Hardware

A BLISS hardware implementation was published by Pöppelmann et al. [39]. It targets the Xilinx Spartan-6 FPGA, uses the KECCAK-f[1600] hash function and offers modules for signing and verifying the signature. The algorithm can be configured for various parameter-sets and allows for tweaking the time it takes to perform the sparse multiplication by configuring the number of cores performing the operation in parallel. This allows balancing the speed of the implementation and the area required on actual hardware.

Both the signer and verifier access common dual port BRAMs for the signature triplet consisting of $\mathbf{z}_1$, $\mathbf{z}_2$, $\mathbf{c}$. Next, the implementations of the signer and verifier are discussed.

3.5.1 BLISS Signer

The *BLISS Signer* module features a modular and configurable design as outlined in Figure 3.1. The *Polynomial Multiplication* module is based on a microcode engine by Pöppelmann and Güneysu [42]. The version used in this BLISS implementation has been scaled

down by removing polynomial operations not used in this signature scheme and statically using parameters specified by the BLISS design. Besides calculating the $\mathbf{a}_1 \cdot \mathbf{y}_1$ product using the NTT, the module is tasked with sampling the random vectors $\mathbf{y}_1$, $\mathbf{y}_2$. Based on the configuration either a Bernoulli- or a CDT Gaussian sampler is used for this task [39].

The product $\mathbf{a}_1 \cdot \mathbf{y}_1$ of the polynomial multiplication is used to compute $\mathbf{u} = \zeta \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$. The result $\mathbf{u}$, as well as the randomly sampled vectors $\mathbf{y}_1$, $\mathbf{y}_2$ are then saved to the respective block memories `BRAM-U`, `BRAM-Z1` and `BRAM-Z2`. Based on the configuration of the number of dropped bits d from Table 3.1 the lower order part of $\lfloor \mathbf{u} \rceil_d \bmod p$ is stored in `RAM-U`. For the hash function used to generate the vector $\mathbf{c}$, Keccak processing 16 slices in parallel, configured with a rate of 1024 bits and a capacity of 576 bits is used. In its current form, the implementation can only handle messages with a length being a multiple of 1024 bits as no padding scheme has been implemented. Furthermore, it is limited to a total size of 4096 bits due to the size of the internal message buffer `RAM-M`, which needs to hold the message in case the signature gets rejected. Due to the rejection sampling at the end of the BLISS-algorithm, the hash function has to be able to rehash the concatenation of the original message μ and a new different $\mathbf{u}$. The issue could be solved by saving the internal state of Keccak after hashing the blocks related to the message, or by rehashing the full message again. This implementation uses the second approach and the full message has to be rehashed, limiting the maximum message length to the size of `RAM-M`. This can be considered a limitation of this particular design. After κ unique positions have been extracted from the hash function H, they are saved to `RAM-Pos` to be used for the sparse multiplication. It should be noted that only the *indices* of the elements of the sparse vector $\mathbf{c}$ that are equal to 1 are stored in `RAM-Pos` and not the vector as such.

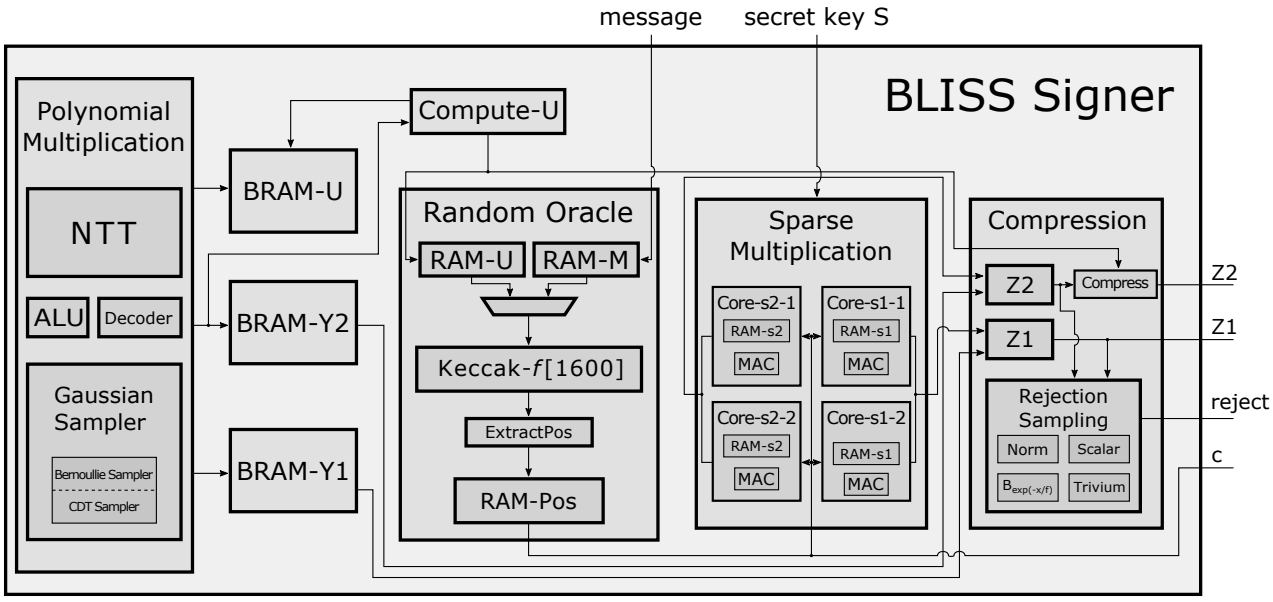


Figure 3.1: Block diagram of the FPGA implementation of the BLISS-I signature algorithm on reconfigurable hardware. The illustration shows the algorithm for $C = 2$ sparse multiplication cores.

The content of `RAM-Pos` is then used for the sparse multiplication in the `SparseMul` module. This module contains a specified number of cores C for each secret key $\mathbf{s}_i$ with $i \in \{1, 2\}$ and calculates the product of $\mathbf{s}_i\mathbf{c}$. The number of cores can be configured and impacts the size and runtime of the scheme. As each core is dedicated to one part of the secret key, each core holds a full copy of either $\mathbf{s}_1$ or $\mathbf{s}_2$. The internals of the sparse multiplication are discussed in Section 3.5.2.

The partial results of the vector multiplication are then used, in parallel to the sparse multiplication, together with $\mathbf{z}_1$, $\mathbf{z}_2$ to calculate the norms and scalar products, needed for the rejection sampling of the signature algorithm.

3.5.2 Sparse Multiplication in Hardware

A critical part of the signature generation is the multiplication of the secret key $\mathbf{S}$ with the sparse vector $\mathbf{c}$ as described in Section 3.3. The hardware implementation is capable of doing a polynomial multiplication with the help of the NTT as described in Section 2.3.2 because computing the vector $\mathbf{u}$ can be efficiently done by utilising this transformation. However, due to all except κ coefficients of $\mathbf{c}$ being 0, the product can be calculated more efficiently by performing a sparse multiplication.
The implemented algorithm on the FPGA performs the sparse multiplication using the schoolbook method as seen in Algorithm 3.4. The algorithm performs a column-wise multiplication, consequently allowing for an on the fly computation of the norm and inner product for the rejection sampling afterwards [39].

Algorithm 3.4 The algorithm to compute the product of the sparse multiplication using a column-wise schoolbook method according to [39]

1: **procedure** SparseMultiplication(Secret Key $\mathbf{s}$, Index Vector $\mathbf{c}$, Result coefficient Index j, n)
2: $result \leftarrow 0$
3: **for** $i \leftarrow 1, \kappa$ **do**
4: $position \leftarrow n - c_i + j$
5: $k \leftarrow position \bmod n$
6: **if** position $<$ n **then**
7: $result \leftarrow result - s_k$
8: **else**
9: $result \leftarrow result + s_k$
10: **return** $result$

The waveform of a reduced example of the sparse multiplication can be seen in Figure 3.2, with κ being 8 and only two cores performing the multiplication.

The `c_index` acts as a counter to feed the saved `c_data` into all cores simultaneously. The `c_data` signal is the index of those elements of the sparse vector c that are set to 1. If the `sc_valid` signal is high, the signal `sc_out` holds the results of the multiplication for the indicated location of `sc_address`.

Each core computes its part of the result independently. Core 1 will calculate the results of the `sc_address` in steps of two, starting at zero, and Core 2 will start at 1. Knowing the output position, each core will then compute the actual address `s_address` of the involved elements of the key $\mathbf{s}_i$ as also indicated on line 4 of Algorithm 3.4. The content of the key $\mathbf{s}_i$ is then available through `s_data` and depending on the calculated

key address, the key is added or subtracted from the content of the temporary register `result_reg` as seen on line 6 of Algorithm 3.4.

After adding or subtracting all involved keys, the result of each `result_reg` is written to `result_out` in parallel and then element-wise written to `sc_out` by the top module of the sparse multiplication.

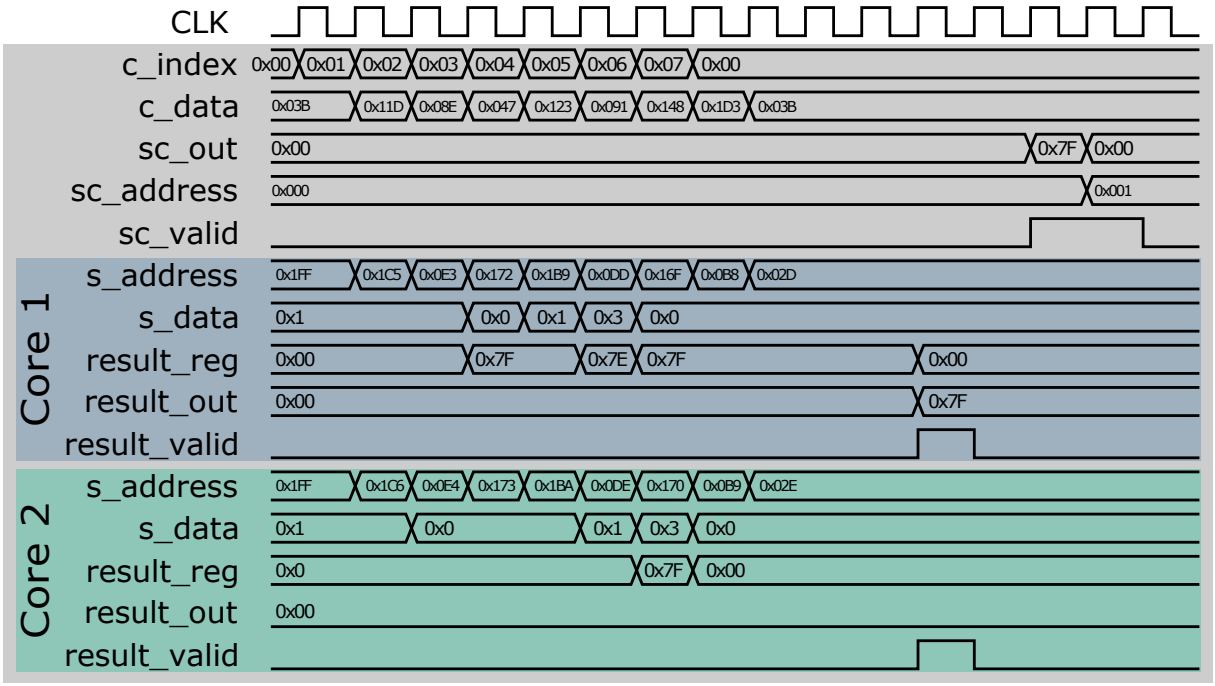


Figure 3.2: Waveform of the sparse multiplication involving the key $\mathbf{s}_1$ using two multiplication cores and $\kappa = 8$, computing subsequent results.

Chapter 4

Side-Channel Attacks

This chapter gives an overview of side channel attacks. While this kind of attack applies to various types of systems and protocols (e.g., the Lucky Thirteen attack on TLS described in [18]), this chapter focuses on side-channel attacks on cryptographic hardware. Some of the concepts, however, can be used for other fields as well.

First, an overview is given to give a better understanding of different types of attacks and how they can be classified in Section 4.1. Continuing with Section 4.2 the basic idea of the principles and parasitic effects of CMOS, the most used technology for microchips, is given. Moreover, Section 4.3 gets into more detail on how this knowledge on CMOS can be used to elaborate data dependent power consumptions described by power models.

Power models are essential for attacks utilising the specific power consumption of a device as exploited in a simple power analysis attack in Section 4.4 or a differential power analysis attack as elaborated in Section 4.5.

Furthermore, ways to mitigate these attacks are given in Section 4.6, focusing on the two most common methods known as hiding and masking. Finally, in Section 4.7 proposed and published ways to perform side-channel attacks on lattice-based signatures are discussed, as well as proposed countermeasures to mitigate these attacks.

4.1 Overview

In general, with cryptanalysis reviewing the algorithm and the mathematical model a cryptographic scheme is based on, side-channel attacks (SCA) specifically target the implementation of a cryptographic scheme. This implementation can range from a physical device running a single cryptographic primitive to an application on a web server performing a cryptographic protocol. While SCA exists in many contexts, the scope of this thesis lies primarily on SCA on cryptographic microchips.

The goal of an SCA is to gain knowledge of internal information of the algorithm in use, like states, conditional branches, internal errors, cryptographic secret keys or even to alter the control flow of the algorithm to make the device reveal internal information.

There is a broad range of methods to perform such an attack, each with a varying amount of cost, tools, time and expertise involved. There are different ways to distinguish between various kinds of attacks. A basic separation is to first categorise into passive- and active attacks and then further distinguish in terms of invasiveness ranging from non-invasive-, over semi-invasive- to invasive attacks. This and the following descriptions given in this chapter are based on Mangard et al. [28].

Active Attacks

To classify as an active attack, the device needs to be tempered with in some way. One way to perform an active attack is to run the device outside of its specification. This can be achieved by, e.g., changing the environment and operating it above the maximum temperature it was designed for or operating it above or below the specified voltage. Another way of performing an active attack is to temper with single input pins of the device.

Passive Attacks

While the active attack requires manipulation of the device under attack, the passive attack aims at exposing confidential internal information. This is achieved by observing the emitted physical properties of the device under normal operation, i.e., fully or at least mainly within the specification. For example, this can be done by, but is not limited to, measuring the power consumption, the emitted electromagnetic interference (EMI), sound or the execution time.

Non-Invasive Attacks

Non-invasive attacks only use the available inputs on the device and do not require to open up the package of the microchips or adjusting parameters of the environment. These properties drive down the cost of non-invasive attacks, increasing the risk for practical attacks. To perform the attack, the device gets known, sometimes specially crafted, data to process. An external measurement device is then used to obtain data on the appliance, like power consumption, execution time or EMI. Examples of passive non-invasive attacks are timing attacks as described by Kocher [23] and Dhem et al. [13], and simple- or differential power analysis attacks as described by Kocher et al. [24]. It should be noted that this kind of attack does not alter the device permanently, making it difficult to detect if the appliance has been tampered with in the past or at runtime.

Active non-invasive attacks include changes in the environment that interfere with the normal operation of the device, but do not require depackaging of the chip. This type of attack includes changes in the environmental temperature [20], introducing clock glitches and changes in the supplied power to induce wrong calculations [7, 5]. Because this requires changes to some domain parameters, it is easier detectable than the passive non-invasive attack, e.g., by using temperature sensors to detect drastic changes in temperature.

Semi-Invasive Attacks

In contrast to non-invasive attacks, semi-invasive attacks include depackaging of the chip down to the passivation layer using mechanical or chemical means. The semi-invasive attack separates itself from the invasive attack by not making electrical contact with the surface of the chip. The depackaging is done to make the chip better accessible for probes or fault inducers. This makes semi-invasive attacks more expensive than non-invasive attacks, and also increases the effort and required know-how for depacking a chip and positioning the probes or tools at the correct location.

In case of a passive semi-invasive attack, the content of the memory is obtained by means other than using a specially crafted readout circuit or probing existing read-out circuits already existing on the chip. As an example, Samyde et al. [48] demonstrated

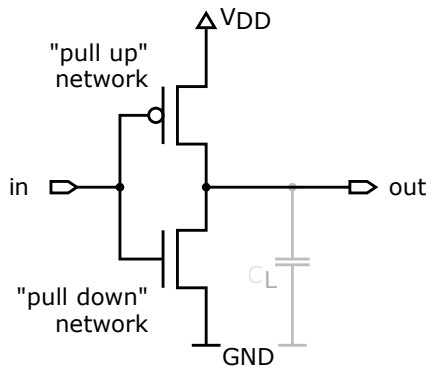


Figure 4.1: Structure of a CMOS inverter logic cell including the pull-up-, pull-down network and the parasitic capacitive load.

how to read the content of memory cells by using lasers without changing the information stored in the cell.

Active semi-invasive attacks on the other hand aim at inducing faults in the device. The depackaging of the chip makes it easier to use X-rays, electromagnetic fields or light to target the chip. In case of light, when photons hit the p- or n-channel area, it induces free carriers, reducing the resistance of the channel and thus making it possible to cause bit-flips in specific parts of the chip as demonstrated by Skorobogatov and Anderson [54].

Invasive Attacks

Invasive attacks represent the most powerful group of attacks in terms of effectiveness but are also the most expensive. They require depackaging the chip, costly tools and a considerable amount of knowledge of the underlying circuit.

In addition to the depackaging, as performed at the semi-invasive attack, the passivation layer also gets removed to allow a probing station to be connected to the microchip. To perform a passive invasive attack, the probing station is then used to log the data transmitted over a bus or at specific pins. This also allows for alterations of the transmitted data using the probing station or lasers, resulting in an effective active attack.

Although the costly nature of this type of attack reduces its practical threat, it is something that should be considered for high-risk applications. A countermeasure based on exploiting imperfections of capacitors in the fabrication process has been proposed by Wan et al. [59].

4.2 CMOS

The most commonly used process technology modern microchips are based on is **c**omplementary **m**etal-**o**xide-**s**emiconductor (CMOS), which also represents the most common way to implement logic cells. In CMOS each logic cell contains a pull-up and a pull-down network as seen in Figure 4.1 for an inverter. To give a basic explanation of the functionality: The pull-up network implements the logic for the output to be a logic 1, whereas the pull-down network covers the complementary cases for the output to be a logic 0.

Due to this structure, the output of CMOS always results in a logic one or zero. Based on Mangard et al. [28] the power consumption of CMOS circuit can be traced back to two main groups, as explained below.

4.2.1 Static Power Consumption

The significance of the static power consumption varies with the structure size of the used process technology. For smaller structure sizes this type of leakage tends to contribute more to the overall power consumption. With CMOS only one of the pull-up or pull-down networks is active at the same time. However, there is still a small parasitic leakage between the supply voltage and the ground connection. This leakage causes the static power consumption and is independent of the processed data. The static power consumption can be calculated based on the leakage Current I_{leak} and the supply voltage V_{DD} of the logic circuit:

$$P_{stat} = I_{leak} \cdot V_{DD} \tag{4.1}$$

4.2.2 Dynamic Power Consumption

The dynamic power consumption is typically higher than the static power consumption. It is caused by switching of the CMOS and depends on the data processed. While the internal switching of a cell is also affected, it contributes only a small part compared to changes in output bits. Therefore, only changes in output signals are considered in the following. Each output bit of a cell can either change to the opposite- or stay at its current value, resulting in a total of four possible transitions. In case the bit does not flip, the cell only consumes static power, whereas a change in its value causes static and dynamic power consumption. Depending on the exact process used each transition may consume a different amount of power. The cause for dynamic power consumption is twofold:

Charging Current

The output as seen in Figure 4.1 is connected to another part of the chip. This can be modelled as a parasitic capacitance. Every time the output changes its value, either the pull-down or the pull-up network has to charge this output capacitance, causing a parasitic current.

Given a clock frequency f, the output capacitance C_L, the supply voltage V_{DD} and an activity factor α, the power consumption causing the charging current is given as:

$$P_{charge} = \alpha \cdot f \cdot C_L \cdot V_{DD}^2 \tag{4.2}$$

The activity factor α denotes the average number of switches between the two output states. A value of $\alpha = 1$ notes that the output switches with every clock cycle. It should be noted that while the frequency f and the capacitance C_L only contribute in a linear way, the supply voltage has a quadratic impact on the power consumption caused by the charging current of output capacitances. Therefore, reducing the voltage V_{DD} as far as possible is a good way to reduce the dynamic charging power consumption.

Short Circuit Current

A change in the input of a CMOS circuit, that also changes the output will affect both the pull-down and the pull-up network. Both the pull-down and the pull-up network will

change in a way that a short-circuit exists between the supply voltage and the ground for a short period of time. The maximum current during this transition is denoted as I_{peak}. This results in a peak in power consumption during the transition of the output signal. A commonly used approximation for the power consumption of this event is given as:

$$P_{sc} = \alpha \cdot f \cdot V_{DD} \cdot I_{peak} \cdot t_{sc} \tag{4.3}$$

With α being the switching activity of the cell compared to the clock frequency f and V_{DD} being the supply voltage used.

4.3 Power Models

Power analysis attacks often require assumptions on the power consumed by the device and the data that has been processed. Such assumptions are referred to as *power models*. Power models can be arbitrarily sophisticated, depending on the amount of information available on the device under attack. However, in general, an attacker does not know the details of the implementation, resulting in simple power models like Hamming-distance or Hamming-weight which offer broader applicability.

With Hamming distance and Hamming weight being the most generic power models, more specific power models implementing specific characteristics of the device can be created. For instance, single bits can be weighted differently (e.g., in the Hamming distance or Hamming weight power model), or assumptions for a larger part of the chip can be made [28].

4.3.1 Hamming Weight

The Hamming weight (HW) power model assumes the power consumption to be proportional to the number of logical ones in the processed data. Given $\mathbf{b}_n$, a binary vector $\mathbf{b}$ of length n:

$$\mathrm{HW}(\mathbf{b_n}) = \sum_{i=0}^{n-1} b_i, \qquad b_i \in \{0, 1\} \tag{4.4}$$

The position of the ones and zeros has no effect on the outcome of the Hamming weight. This model does not attribute to past values on a bus or register and is therefore mainly used in situations where the attacker has no or only very limited information about previously stored data. Due to the dynamic power consumption in CMOS (Section 4.2.2), transitions in the circuit lead to a change in the power consumption. Based on these properties the Hamming weight model is not very well suited for CMOS.

4.3.2 Hamming Distance

With the Hamming weight model based on a specific state of a variable and its related power consumption, the Hamming distance (HD) model emphasis on the transition between states. More precisely it counts the number of bits changing their value from one state to another. For this matter, the direction of the transition is not accounted for. It should be noted that in actual CMOS hardware the power consumption depends on the direction because pull-down and pull-up networks have different power characteristics. Furthermore, the Hamming distance model does neither account for parasitic capacitances nor the static

power consumption. Therefore, a transition from a logic zero to a logic one contributes the same way to the overall power consumption as a transition from one to zero.

With two states A and B, represented as binary strings of the same length, and $\oplus$ being a logical XOR, the Hamming distance of the transition between the two can be seen as the Hamming weight of the XOR of the two states:

$$\mathrm{HD}(A, B) = \mathrm{HW}(A \oplus B) \tag{4.5}$$

The Hamming distance model is well suited for parts of the device with a high number of capacitive loads or long data lines, like a bus, which is connected to many components. Registers are driven by a clock and therefore only change their value once per clock cycle, making them suited for the Hamming distance power model as well. In contrast to parts like buses and registers, the output of combinational logic cells glitches between its possible values, making it unsuited for the Hamming distance model [28].

To take a closer look at the Hamming weight and the Hamming Distance model, based on Mangard et al. [28], three cases of transitions between a state A to a state B are compared:

Primary state equal and constant

Given a number of transitions from state A to B, the Hamming-weight model equals the Hamming-distance model, if A always has the same value with all its bits set the same logical value. Therefore, with $\oplus$ indicating a binary exclusive-or operation, it holds that $\mathrm{HD}(A, B) = \mathrm{HW}(A \oplus B) = \mathrm{HW}(B)$. As power analysis tries to correlate the power consumption to the processed data, it is indifferent if it is directly or inversely proportional to the processed data, hence it makes no difference if state A consists of logic ones or zeros.

Primary state constant

In this case, the bits of state A are constant and thus the same for each transition from A to B. However, while the overall value of A is the same before each transition, the value itself consists of a mix of logical ones and zeros and is not known to the attacker. Therefore, the Hamming weight model is different from the Hamming distance model in this case, making it a poor choice for this kind of transitions. However, it is possible to base the model on a single bit making both models equivalent in terms of power analysis attacks. Furthermore, it should be noted that the accuracy of the Hamming weight power model increases the more bits of state A share the same value.

Independent uniformly distributed transition

In this case, the bits of state A are random for each transition A to B, following a uniform distribution. The properties of A clearly make the Hamming weight model unsuited for this type of transition, as $\mathrm{HW}(B)$ is different from the Hamming distance $\mathrm{HD}(A \oplus B)$ with A being a random and independent state from B.

4.4 Simple Power Analysis

A simple power analysis (SPA) typically involves a small number of power traces, that were captured during the operation of the device and the attacker tries to interpret them

directly [24]. This is useful in scenarios in which the attacker has only a limited number of occasions to measure the power consumption of the device, either because the target is only using it a limited time, but also in cases in which the device locks up after a certain number of tries. In situations like these, there might be only a single trace to recover the embedded key, indicating the difficulty of this attack in certain settings. Based on Mangard et al. [28] this section discusses two ways of performing an SPA attack.

4.4.1 Attack by Visual Inspection

An attack based on visual inspection of the traces is the most basic form of an SPA attack. Integrated circuits usually consist of multiple components working together. The algorithm executed on the device makes use of several of these components. Especially in case of software run on microcontrollers, the executed instructions show a distinctive pattern in the power consumption, caused by arithmetic instructions, logic instructions, branching or data transfer instructions. Additionally, microcontrollers are designed for general purpose applications and therefore less tightly integrated in regard to the active components in parallel.

However, this attack requires detailed knowledge of the algorithm. After normalisation of the measured power traces, this knowledge can be used to label the segments of the executed algorithm in the trace. In a badly designed implementation, the labels not only show the executed instructions but also give information on the key, because the instructions used are directly related to the key. A primary example for this would be a branching based on a small part of the key being zero during a multiplication.

4.4.2 Template Attacks

Template attacks represent a more advanced form of SCA attacks. The assumption for this attack is that the data processed by the device influences the power consumption. Essentially, the power consumption of the device is characterised using a multivariate normal distribution and described by a pair of its mean $\mathbf{m}$ and its covariance matrix $\mathbf{C}$. The pair consisting of $(\mathbf{m}, \mathbf{C})$ is referred to as a template, denoted as h.

A template attack consists of a template building- and a template matching phase. The template building should be run on a device as similar as possible to the device under attack but allowing for arbitrary data and keys. Afterwards, the obtained data can be used for the template matching phase to recover the key on the target device.

Template Building

The template building phase aims at characterising the device by measuring the power consumption $\mathbf{t}$ for all combinations of data inputs d_i and keys k_j. It should be noted that the number of combinations of the key and data only cover the possibilities necessary for the affected operation and not the entire key-space. The template can be built for a single- or a sequence of instructions. The power traces referring to the same data are then consolidated to estimate the mean $\mathbf{m}$ and the covariance matrix $\mathbf{C}$ resulting in the templates $h_{d_i,k_j} = (\mathbf{m}, \mathbf{C})_{d_i,k_j}$ for each tuple of data value and key. In contrast to the mean, the size of the covariance matrix is increasing quadratically with the number of points in the trace. Therefore, it is necessary to limit the power traces to a set of points with a power consumption showing a high data dependency.

Depending on the algorithm and properties of the device, different strategies can be used to build templates:

One way, as already hinted, is based on points in the algorithm that directly rely on pairs consisting of data value and key (d_i, k_j). The power consumption is proportional to the values of the tuple.

Alternatively, the templates can also be created for intermediate values based on some function $f(d_i, k_j)$. The function f can be any part of the algorithm that incorporates the data value and the key and maps it to a known output.

In some cases, it may be necessary to make use of knowledge about the power model describing the device to reduce the number of templates drastically. For instance, for a device leaking the Hamming weight at a specific operation, the number of templates for an 8-bit value can be reduced from 256 to nine by grouping templates causing the same Hamming weight.

Template Matching

Once the template building is completed, the data can be processed and matched against the power trace(s) of the device under attack. The templates h_{d_i,k_j} are then used with the power trace t to calculate the probability p of them matching each other:

$$p(\mathbf{t}; h = (\mathbf{m}, \mathbf{C})_{d_i,k_j}) = \frac{\exp\left(-\frac{1}{2} \cdot (\mathbf{t} - \mathbf{m})' \cdot \mathbf{C}^{-1} \cdot (\mathbf{t} - \mathbf{m})\right)}{\sqrt{(2 \cdot \pi)^T \cdot det(\mathbf{C})}} \tag{4.6}$$

This results in a probability for each of the templates matching the trace. The template with the highest probability, based on the maximum-likelihood decision rule, is then the most likely to fit the trace:

$$p(\mathbf{t}, h_{d_i,k_j}) > p(\mathbf{t}, h_{d_i,k_l}) \forall l \neq j \tag{4.7}$$

In practice, however, as seen in Equation 4.6, it is necessary to invert the covariance matrix $\mathbf{C}$, which may not always be mathematically possible due to the measured power traces the matrix is based on. In such cases, the matrix can be substituted by the identity matrix of the same size. This effectively just ignores the covariance between the points of the trace. The resulting template is then called a *reduced template*.

To more efficiently compute the likelihood the logarithm can be used to simplify the Equation 4.6:

$$\ln p(\mathbf{t}; h = (\mathbf{m}, \mathbf{C})) = -\frac{1}{2}(\ln((2 \cdot \pi)^T \cdot det(\mathbf{C})) + (\mathbf{t} - \mathbf{m})' \cdot \mathbf{C^{-1}} \cdot (\mathbf{t} - \mathbf{m})) \tag{4.8}$$

The maximum likelihood then becomes

$$|\ln p(\mathbf{t}, h_{d_i,k_j})| < |\ln p(\mathbf{t}, h_{d_i,k_l})| \forall l \neq j \tag{4.9}$$

4.5 Differential Power Analysis

The differential power analysis (DPA) [24] is one of the most universally applicable power analysis attacks because it does not require detailed knowledge of the internals of the device under attack. Furthermore, this kind of attack is hardly affected by noise generated

by other areas of the device. For an attack, it is basically sufficient to know the used cryptographic algorithm the device is executing. Compared to the simple power analysis, the DPA requires more power traces to succeed, making it usually a requirement to get into possession of the attacked device. The attack itself focusses on the relative power consumption at a specific moment in time, as it assumes the power consumption to depend on the processed data. Therefore, it is critical to properly align the traces to be able to perform the attack at all. A DPA attack can generally be split into five steps [28]:

First, it is essential to identify the intermediate results of the algorithm that are going to be the target of the attack. This intermediate value $f(d, k)$ has to depend on some part of the key k as well as a known, non-constant value d that can be altered by the attacker, like a message to be signed or a ciphertext.

Second, the power consumption of the device needs to be measured while performing its operation on the intermediate value. The attacker needs to save the intermediate value d used in the respective run and store it in connection with the measured power trace t. Each power trace has a length T. The known values d are stored in the vector $\mathbf{d}$.

Next, a vector $\mathbf{k}$ of length K containing all possible value of the key k is created. The value k is not the full key, but only a much smaller part. The vector is then used, together with the known vector $\mathbf{d}$ to compute the output of the function $f(d, k)$ for all possible keys in $\mathbf{k}$. The result is a matrix with the number of elements in $\mathbf{k}$ times the elements in $\mathbf{d}$. Because the matrix is built from the data that has been used and all possible key values, the DPA aims to find the column that has actually been processed by the device. This step is referred to as building a hypothetical intermediate value.

Afterwards, each of the hypothetical intermediate values is mapped to a hypothetical power consumption. The power consumption is calculated by applying a suitable power model, like the Hamming distance model, on the intermediate values generated in the previous step. It is essential to apply a power model that best describes the attacked device for the DPA to work properly.

Finally, the hypothetical power consumption is mapped to each position of the actual power traces. This is then compared using statistical methods to determine which of the hypothetical intermediate values fits best to the observed power traces. One way to determine the likeliness of the hypothetical model being represented in the power traces is the *correlation coefficient*. The matrix $\mathbf{H}$ is representing the hypothetical power model and $\mathbf{T}$ consists of the measured and aligned power traces with $\mathbf{h}_i$ and $\mathbf{t}_j$ being column vectors of length D of the respective matrices with $i \in \{1, 2, ..., K\}$ and $j \in \{1, 2, ..., T\}$. The variables $\bar{h}_i$ and $\bar{t}_j$ represent the mean values of the respective vectors $\mathbf{h}_i$ and $\mathbf{t}_j$. The correlation coefficient $r_{i,j}$ for each power trace and hypothetical power model can then be calculated with:

$$\mathbf{R} = r_{i,j} = \frac{\sum_{d=1}^{D}((h_{d,i} - \bar{h}_i) \cdot (t_{d,j} - \bar{t}_j))}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i)^2 \cdot \sum_{d=1}^{D}(t_{d,j} - \bar{t}_j)^2}} \tag{4.10}$$

If there's a correlation between the power hypothesis and the power traces, the location of the highest values in the matrix $\mathbf{R}$ indicate the secret key used by the cryptographic device.

4.6 Countermeasures

The ability to extract the full or at least parts of the secret key embedded in a cryptographic device raises the need to counteract these attacks and protect the secret asset or at least raise the difficulty to perform the attacks to a certain level. This can be achieved by using hiding, masking or even a combination of both [28]. It should be noted that in practice it is not always possible to classify a countermeasure into a specific category because a combination of different techniques may be used and slightly different definitions for the terms exist.

4.6.1 Hiding

Hiding is a countermeasure against power analysis attacks that tries to make the power consumption independent of the processed intermediate data values and the performed operation on the data. This can be achieved by either making the device consuming a random or the same amount of power in each clock cycle throughout its operations. However, both these properties cannot be entirely fulfilled in practice. Methods of hiding the power consumption focus on either the time dimension or the amplitude dimension and can be implemented in software or hardware.

Time Dimension

One requirement for a side-channel attack like the differential power analysis (Section 4.5) is to align the traces properly such that the same operation is executed at the same time over all traces. This property can be exploited to protect the implementation. Hiding the point in time an operation has been executed significantly increases the number of traces needed for a successful attack.

To hide the operation in the time dimension either the order in which the necessary operations are executed is shuffled or dummy operations are inserted randomly.

To protect an implementation by *randomly inserting dummy operations*, they have to be placed before, during and after executing the cryptographic primitive. The number of operations inserted overall has to be equal, to prevent the attacker from gaining any information on the number of totally inserted dummies. The split between the locations must be determined using random numbers. By varying the number of operations at each location, an attacker cannot align the traces easily as needed for a DPA. However, the downside to this approach is that while the number of distributed dummy operations determines the security gained with this method, more dummy operations slow down the algorithm. In practice, a trade-off between the number of operations and overall speed has to be found.

Another way of hiding the executed operations is to shuffle the order in which the operations of the cryptographic algorithm are executed. The applicability is highly dependant on the algorithm itself, as it is not possible for all operations. Operations like entry-wise table lookups, or matrix multiplications are examples of operations that can be shuffled easily. In general, shuffling has a smaller overhead in performance than the insertion of random operations (which effectively slows down the implementation) but is not as universally applicable.

Amplitude Dimension

The alternative to hiding in the time dimension is hiding in the amplitude dimension. This countermeasure aims at equalising the signal-to-noise ratio (SNR) of the cryptographic operations by either increasing the noise or by reducing the leakage of the signal. Both methods aim at reducing the SNR to 0, i.e., the signal is indistinguishable from noise, though this is not entirely possible in practice.

To increase the noise of the signal, the simplest way is to perform operations in parallel, thus increasing the effective width of the data path and also the noise introduced. Dedicated noise generators represent an alternative to this. They add random switching activity to the device and thus increase the noise. A downside of this is the increased overall power consumption introduced by the switching of the noise generator.

Another option is to reduce the signal of the cryptographic operations. However, the DPA requires the attacker to capture a large amount of traces, thus making even small differences in the traces exploitable. The most common way to effectively reduce the signal is to employ a dedicated logic style. Because the total power consumption is based on the sum of the consumed power of the cells, if each logic cell consumes the same amount of power for any given input the overall power is constant as well. On the hardware level, this can be achieved by using dual-rail logic, precharge logic or a combination of both.

4.6.2 Masking

Masking is another way of protecting an algorithm against power analysis attacks. In contrast to hiding, masking does not aim at deceiving about the power consumption connected to a certain operation or processed value. Instead, it aims at breaking the connection between the intermediate value and the processed data. This also means that the power consumption will still depend on the processed data, but the processed data will not allow drawing conclusions on the intermediate value.

In case of a DPA (Section 4.5), given an operation $f(d, k)$, the goal is to find the key k by using the power consumption of f and the known value d. To break this link using masking a random mask m is generated and applied to either d or k before performing the operation. As a result, because the mask m is not known to the attacker, the operation computing $f()$ works on an unknown d or a randomised k, thus not meeting the requirements for a DPA.

Masking itself can be done by boolean or arithmetic means. For the former, the intermediate value of the computation is XORed with the mask. In case of the known intermediate, this leads to a concealed intermediate $d_m = d \oplus m$. This type of masking can be applied easily with only a minimal performance impact. However, to use this method, the performed operation has to be linear in regard to the XOR operator, i.e. $f(d \oplus m) = f(d) \oplus f(m)$.

Arithmetic masking acts as an alternative and may be applicable in different scenarios. To perform arithmetic masking, the masked value is combined with the intermediate value through an arithmetic operation like addition or multiplication, often with the values defined over a ring. In regard to multiplicative masks their weakness is towards the intermediate value being 0, therefore not concealed by any multiplicative operation.

In the context of asymmetric cryptographic schemes, masking schemes are commonly referred to as blinding, involving additive as well as multiplicative operations. In some cases, the blinding schemes do not require to be unmasked explicitly because the mathematical characteristics are exploited that inherently cancel the mask towards the end.

4.7 Side-Channel Attacks on Lattices

This section gives an overview of published theoretical and practical side-channel attacks on lattice-based schemes and mitigation techniques, some of which can also be applied to the BLISS signature scheme. However, attacks on a software implementation might not be directly transferable to an implementation on reconfigurable hardware.

4.7.1 Gaussian Sampling

Attacks have been proposed that target the Gaussian sampler used in the signature creation to recover the secret key. The attack proposed by Espitau et al. [17] pursues a branch tracing attack. A first proof-of-concept cache-attack on the sampler of BLISS was shown by Bruinderink et al. [9]. Later Pessl et al. [37] proposed an improved cache-attack in a more realistic setting on the sampler of BLISS-B.

Branch Tracing Attack

A weak point of the BLISS signature scheme is the Gaussian sampling of the noise vectors $\mathbf{y_1}$ and $\mathbf{y_2}$. The noise vectors are used to compute the output of the signatures $\mathbf{z}_i = \mathbf{y}_i + (-1)^b \mathbf{s}_i \mathbf{c}$ with $i \in \{1, 2\}$. Therefore, knowing the noise vector paired with an invertible vector $\mathbf{c}$ allows to directly compute the secret key involved in creating the signature. The sparse vector $\mathbf{c}$ is invertible in about 95% of the cases. To obtain the information on the involved noise vectors, Espitau et al. [17] utilise the time variability of the sampler as proposed by Pöppelmann et al. [39]. This sampler follows an iterative approach and requires a variable number of iterations. The variation in time can then be used to infer information on the generated values. In case of the strongSwan implementation, a branch tracing technique can be applied, making it possible to determine the number of iterations involved.

Cache-Attack

An attack on the related scheme BLISS-B was proposed by Pessl et al. [37]. The lattice-based signature scheme is an improved version of BLISS and was proposed by Ducas [14]. BLISS-B uses a slightly modified key-generation and signing algorithm. Besides removing the constants C and N_k, described in Section 3.2, the sparse multiplication has been modified to minimise the Euclidean norm of $||\mathbf{Sc}||$, i.e., the vector $\mathbf{c}$ has been replaced by a ternary vector $\mathbf{c}' \equiv \mathbf{c} \bmod 2$. Additionally, during the key-generation keys are no longer rejected. However, signatures generated with BLISS or BLISS-B are interoperable. The keys, on the other hand, are only forward compatible, i.e., keys generated to be used with BLISS-B should not be used with BLISS.

The attack on BLISS-B targets the implementation of the algorithm in the strongSwan VPN solution. Despite the fact that the strongSwan implementation is not incorporating any countermeasures against side-channel attacks, with only a few real-world applications implementing an algorithm of the BLISS signature family this represents a realistic scenario.

The attack is based on an asynchronous cache-attack recovering the unknown noise-vector $\mathbf{y}_i$, as also seen on line 2 of Algorithm 3.2 and use it to acquire the secret key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)$. The information on $\mathbf{y}_i$ is then used to build a system of linear equations. It should also be noted that the multiplication is performed between the derived

and undisclosed vector $\mathbf{c}'$ instead of its original counterpart $\mathbf{c}$. Additionally, in case the elements were shuffled they would need to be reordered before setting up an equation $z_i = y_i + (-1)^b < \mathbf{s}, \mathbf{c}'_i >$ for each recovered element of $\mathbf{y}_i$. Due to the unknown signedness of $\mathbf{c}'$ this set of equations cannot simply be solved by Gaussian elimination.

Therefore, the second step involves solving the set of equations acquired during the cache-attack. By solving the equations over bits instead of over integers, the location of all nonzero elements can be predicted with a certain probability. However, the system of equations based on the noise vector $\mathbf{y}_i$ may contain a certain number of errors, due to the accuracy of the used side-channel attack. This can be solved using a Learning Parity with Noise algorithm [38]. Applying this method results in the location of $\lceil \delta_1 n \rceil$ recovered elements of the key $\mathbf{s}_1$ being $\{\pm 1\}$.

In some cases, depending on the parameter set of the BLISS-B algorithm in use, the key also contains coefficients with $\{\pm 2\}$. To recover the location of these elements a heuristic can be applied which is either using an Integer Programming solver or a Maximum Likelihood estimate, depending on the particular parameter set used.

In the last step, the magnitude of the coefficients is used in conjunction with the public key to construct a Shortest Vector Problem and retrieve the signedness of the elements. As shown in Algorithm 3.1 the public key is computed with $\mathbf{a}_q = (2\mathbf{g} + 1)/\mathbf{f} = \mathbf{s}_2/\mathbf{s}_1$ in $\mathcal{R}_q$. This can be rewritten as $\mathbf{s}_1 \cdot \mathbf{a}_q = \mathbf{s}_2$. As demonstrated in Section 2.3, this can then be used to construct a matrix-vector product $\mathbf{s}_1 \cdot \mathbf{a}_q = \mathbf{s}_2$. Rows in $\mathbf{A}_q$ affected by $|\mathbf{s}| = 0$ can then be eliminated. Discarding these rows leads to a reduced matrix $\mathbf{A}_q^\star$ with dimensions $\lceil \delta_1 n \rceil \times n$. The dimension can then be reduced further by discarding some of the upper coefficients, making it easier to find the shortest vector. The secret-key is recovered by solving this SVP using the BKZ lattice-reduction algorithm. While previously only the magnitude of the elements of the key has been known, i.e., the key element $|s_i| = \{0, 1, 2\}$, this step ensures the recovery of the signedness of the non-zero elements [37].

With the described attack it is possible to recover the secret-key with only 325 signatures.

4.7.2 Rejection Sampling

As described in Section 3.2.2, the BLISS signature algorithm performs a rejection sampling towards the end of the signing procedure. This step depends heavily on the squared norm $||\mathbf{Sc}||^2$ and the scalar product $< \mathbf{z}, \mathbf{Sc} >$, both involving the secret key $\mathbf{S}$ and the publicly known sparse vector $\mathbf{c}$. As shown by Espitau et al. [17] the scalar product could be used in conjunction with $\mathbf{z}_2$ to create a linear system and recover the secret key. However, due to the signature compression performed, which essentially drops the lower-order bits of $\mathbf{z}_2$, resulting in $\mathbf{z}_2^\dagger$, this is not possible.

Retrieving information on the norm $||\mathbf{Sc}||$ can be used to create an algebraic system, that can be solved using a more generalised version of the Howgrave-Graham-Szydlo [19] algorithm. While most of the attack on this part runs in polynomial time, it also requires factorisation of the norm. Therefore, this is only feasible for a certain number of easy to factorise numbers [17].

4.7.3 Sparse Multiplication

An additional target discussed by Espitau et al. [17] is the sparse multiplication. The sparse multiplication involves the secret key as well as the known sparse binary vector $\mathbf{c}$ with a Hamming weight of exactly κ. To save memory and speed up the multiplication the sparse

vector $\mathbf{c}$ is usually not stored itself, but instead only the indices of the elements being 1 are stored. Although the polynomial multiplication can be performed by using the NTT, the structure of the sparse vector $\mathbf{c}$ allows for an efficient multiplication by performing successive additions. First, the internal order in which the elements of the sparse vector are accessed has to be determined, because the order of the $\mathbf{c}$ indices can be random, as long as the correct elements affected by $\mathbf{c}$ are used.

In a second step, the result is suggested as the target of the attack because it gets updated κ-times. However, the issue with attacking the result is that its value is depending on the previous iteration. Using this observation and trying to solve it using as Markov process does only lead to mediocre results.

Better results can be obtained by distinguishing between the result being larger or equal to zero and being smaller than zero, based on many leading zeros or ones respectively. After distinguishing these cases, the elements of the secret key can be recovered the same way as solving an Integer Linear Programming problem [17]. This setting is supposed to perform better in situations with higher noise levels than viewing it as a Markov process.

4.7.4 Number Theoretic Transform

Primas et al. [43] published an attack on the Number Theoretic Transform (NTT). Due to the attack targeting the NTT, it affects most of the efficient and available lattice-based schemes using this transformation to speed up parts of their computation. The described attack is demonstrated on a software implementation of a *Ring Learning with Errors* encryption scheme, implemented on an ARM Cortex-M4F microcontroller. Due to its simple modular operations, the NTT presents a good target for a side-channel attack. Furthermore, the NTT can be drawn as a butterfly network, that gets applied repeatedly to the input data. Therefore, in case of a microcontroller, the same parts of the chip get repeatedly activated, leading to loop-invariant leakages.

In this attack, the inverse-NTT during the decryption process is targeted. The attack only needs one trace to recover the key with high probability using templates. In case of a masked computation, this attack is still applicable, because it requires only one trace and therefore allows to attack each share individually, recombine them and recover the secret key. Countermeasures based on polynomial blinding only show limited protection as well.

In the first step, the EM side-channel is used to profile the operations. This is then continued with a template matching to determine a probability vector for all possible inputs of the operation and each computed butterfly. The operations in the butterfly-network consist of modular multiplications as well as subtractions and additions.

Secondly, the results of the template matching phase are combined and used with the Belief Propagation (BP) algorithm proposed by Pearl [34]. BP is a method to estimate the marginalised probability distribution using the factorisation of a joint probability distribution. To apply the BP on the NTT, it is required to construct a factor-graph. A factor-graph consists of variables nodes and factor nodes. Each input and output of a butterfly is treated as a variable node in the graph. The factor nodes are labelled based on the operation in the butterfly network, therefore either as multiplication, addition or subtraction. It is possible to add further leakage points, like memory operations, to the graph and extend the model if necessary.

The BP is applied until convergence is reached, however applying it to the full graph does not lead to good results. Therefore, it is split into several subgraphs before applying the BP algorithm on the smaller parts to yield better results.

In the last step, the secret key is recovered. Unfortunately splitting the factor graph during the second step and applying the BP only on parts of the graph prevents determining the key using only linear algebra. However, linear equations can be created based on the intermediates of the graph and used to reduce the rank of the lattice-based on the public key of the scheme. Finally, the key is recovered using lattice reduction, utilising the linear equations recovered from the factor graph and combining it with the public key. By combining the two components, the rank of the lattice can be drastically reduced, making it feasible to solve [43].

While possibly making it necessary to select different intermediates in step 2, polynomial blinding as a countermeasure does not offer a significant amount of protection for this attack. Due to a higher amount of parallelism in a hardware implementation, it should be noted that this attack may not be directly applicable there.

4.7.5 Countermeasures

With several side-channel attacks on lattice-based schemes, countermeasures to protect the implementations of the algorithm become even more important. Saarinen [47] showed methods to protect against side-channel attacks at the two steps most prone to attacks for the BLISS-B signature scheme, a further development of the BLISS signature scheme. This section goes into detail with ways of protecting the polynomial multiplication and a proposed protection mechanism for Gaussian sampling during the signing process [47].

Polynomial Blinding

This countermeasure is supposed to make operations involving the secret key during the polynomial multiplication more randomised and hence, less prone to timing, power or emission based side-channel attacks.

BLISS-B operates on a ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ and ideal lattices. Similar to BLISS a polynomial multiplication is performed involving the secret key. One way to protect this operation is to multiply each of the two polynomials $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ with a random constant $a, b \in \mathbb{Z}_q$. Performing the multiplication then leads to the result $\mathbf{h}$:

$$\begin{aligned} \mathbf{h} &= a\mathbf{f} \cdot b\mathbf{g} \\ \mathbf{f} \cdot \mathbf{g} &= (ab)^{-1}\mathbf{h} \end{aligned} \tag{4.11}$$

This method is applicable to the NTT domain and the regular domain. However, in case of anti-cyclic multiplications in the NTT domain, the involved roots of unity have to be adapted, as stated in [47].

An alternative way of blinding that can be combined with the previous method involves the circular shifting of the two polynomials based on the work of Lee et al. [25] and Zheng et al. [60]. A polynomial $\mathbf{f}$ can be represented as $\sum_{i=0}^{n-1} f_i x^i$. This representation allows for easy shifting by an integer j leading to:

$$x^j \mathbf{f} = \sum_{i=0}^{n-1} f_i x^{i+j} = \sum_{i=0}^{n-1} f_{i-j} x^i \tag{4.12}$$

Both methods can be combined to a single function $\texttt{Blind}(\mathbf{v}, s, c)$ which first performs multiplicative blinding with the parameter c, followed by the circular shift of the polyno-

mial by s. The function reversing the blinding is denoted as $\texttt{Blind'}(\mathbf{v}, -s, c^{-1})$, basically reversing the shift and multiplying with the inverse constant.

To implement this combined countermeasure for the multiplication $\mathbf{f} \cdot \mathbf{g}$ both polynomials need to be blinded separately with the constants $a, b \in \mathbb{Z}_q$ and the shift values $r, s \in \mathbb{Z}_n$, then multiplied and finally the blinding needs to be reversed to retrieve the product of the operation:

$$\begin{aligned} \mathbf{f}' &= \texttt{Blind}(\mathbf{f}, r, a) \\ \mathbf{g}' &= \texttt{Blind}(\mathbf{g}, s, b) \\ \mathbf{h}' &= \mathbf{f}' \cdot \mathbf{g}' \\ \mathbf{f} \cdot \mathbf{g} &= \texttt{Blind'}(\mathbf{h}', -(r+s), (ab)^{-1}) \end{aligned} \tag{4.13}$$

To reduce the computation needed during the operation the constants for the multiplicative blinding can be chosen from the roots of unity. In this case, the blinding countermeasure introduces 36 bits of entropy [47].

Split Shuffled Sampling

Similar to BLISS, BLISS-B samples random vectors, following a discrete Gaussian distribution for creating the signature. However, if the random vector of length n is known the secret key can be recovered, as demonstrated in Section 4.7.1. With $D^n_{\mathbb{Z}}(\mu, \sigma^2)$ denoting n elements of a discrete Gaussian distribution with a mean of μ and a variance σ^2, the process of sampling a vector $\mathbf{y}_i$ from the distribution can be noted as $\mathbf{y}_i = D^n_{\mathbb{Z}}(0, \sigma^2)$

If implemented without any thoughts on side-channel attacks, the sequentially sampled values leak through their power consumption. However, to tackle this problem, Saarinen [47] proposed a shuffling mechanism, originally published by Roy et al. [46], paired with a split sampling of the random values. The latter utilises the additive property of variances of discrete Gaussian distributions. Given any two discrete Gaussian distributions $D^n_{\mathbb{Z}}(\mu_1, \sigma_1^2)$, $D^n_{\mathbb{Z}}(\mu_2, \sigma_2^2)$ their sum equals a Gaussian distribution the following way:

$$D^n_{\mathbb{Z}}(\mu_1, \sigma_1^2) + D^n_{\mathbb{Z}}(\mu_2, \sigma_2^2) = D^n_{\mathbb{Z}}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2) \tag{4.14}$$

This property can be applied to the sampling of the random vector such that the iterative addition of the sampled m random vectors of the distribution $D^n_{\mathbb{Z}}(0, (\frac{1}{\sqrt{m}}\sigma)^2)$ equals a one-time sampling over $D^n_{\mathbb{Z}}(\mu, \sigma^2)$. Different ways of splitting the sampling are possible as long as the additivity is complied with.

With this idea in mind, the performance is reduced by a factor of up to m, because the sampling has to be performed several times. However, due to many implementations using table-approaches for the discrete Gaussian sampling, smaller tables with fewer entries may be used. Additionally, because the implementation is better protected against SCA attacks, faster algorithms that would otherwise leak more information can be used [47].

Nevertheless, as analysed by Pessl [36] this countermeasure is not fully protecting the sampler itself. Even so, the number of samples needed to overcome this countermeasure is increased. Splitting the sampling with $m = 2$ makes it already a lot harder to attack the sampler [36].

Chapter 5

Side Channel Evaluation

In this section, a power side-channel analysis of the adapted hardware implementation of the BLISS signature scheme by Pöppelmann et al. [40] is presented. In Section 5.1 the hardware used for the evaluation and its feature are described. This is then followed by an overview of the hardware implementation used for the practical part, a description of the contribution to the source-code and also a short summary of initial issues that had to be overcome in Section 5.2. Section 5.3 describes the protocol used for the communication in the measurement setup between a PC and the hardware implementation acting as a co-processor.

The setup to acquire side channel information is then explained in detail in Section 5.4. Next, the attack is evaluated in Section 5.5, including the selection of the value of interest, but also explaining the processing of the captured data and finally going into detail with the results.

5.1 Measurement Platform

To run and attack the VHDL implementation of the BLISS signature algorithm on hardware, the SAKURA-G (**S**ide-channel **A**ttac**K** **U**ser **R**eference **A**rchitecture) board was used. The board features two Xilinx Spartan-6 FPGA cores, with the larger one (Xilinx Spartan-6 XC6SLX75-2CSG484C) running the BLISS signature algorithm. The smaller, less capable, chip (Xilinx Spartan-6 XC6SLX9-2CSG225C) acts as a communication interface or controller between the signature core and a USB interface connected to a desktop computer.

The BLISS implementation by Pöppelmann et al. [40] targets the Xilinx Spartan-6 FPGA family, which makes the SAKURA-G a perfect fit for running the analysis. Besides having a compatible FPGA onboard, it also features a 26 pin GPIO header accessible by the FPGA and also an amplified measurement point to be used for the attack.

5.1.1 Clock

On the SAKURA-G board, the two FPGAs are connected to dedicated oscillators with a frequency of 48 MHz. Additionally, a 60 MHz clock is available through the USB interface chip [57].

To simplify the communication between the controller and the main FPGA, the former provides a clock for both devices. The LBUS interface connecting both FPGAs (Section 5.3) specifies a wire to transmit the clock signal of the bus. The main FPGA uses

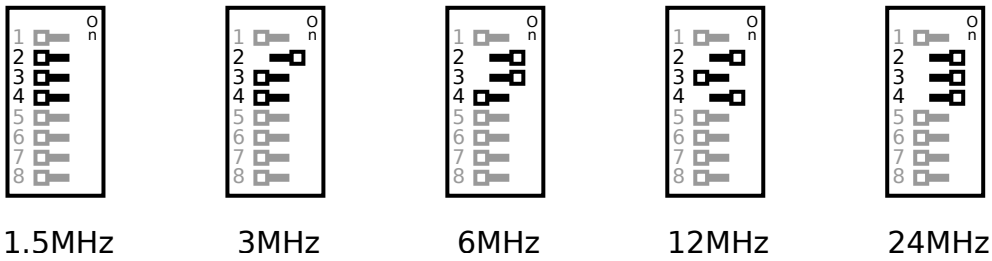


Figure 5.1: On-board DIP switches to set the clock frequency of the controller and the main FPGA.

this LBUS clock as clock source for all its operations. The clock `CLK` for the LBUS can be configured by setting the DIP switch `SW10` as shown in Figure 5.1. The clock can be altered between 1.5 MHz, 3 MHz, 6 MHz, 12 MHz and 24 MHz. However, it should be noted that the setting for 24 MHz makes the implementation prone to transmission errors on the LBUS. Internally the controller uses its 48 MHz clock and a clock divider based on the setting of the DIP switch to generate the clock signal.

The selected clock frequency throughout this thesis is 12 MHz.

5.2 High-Level Architecture

This section takes a closer look at the structure and implementation of the BLISS signature algorithm on the SAKURA-G board. First, a brief summary of how to run the signature scheme on actual hardware is given in Section 5.2.1. This includes a brief discussion of several issues that came with the implementation and a summary of the contribution to the hardware implementation. Furthermore, a top-level overview of the signing algorithm in hardware is shown in Section 5.2.2.

5.2.1 Porting the Implementation to SAKURA-G

As a first step, it is necessary to get the BLISS implementation developed in VHDL by Pöppelmann et al. [39, 40] to run on the FPGA. Albeit the project has been implemented and configured for the Xilinx Spartan-6 FPGA family, it is not ready for the SAKURA-G board out-of-the-box. Initially, the design consists of a signer, a verifier, three external dual-port RAM modules and some additional logic, all residing in a test bench to be used for evaluation purposes. The main test bench performs several sign-verification cycles testing the basic functionality of the design. Additional test benches check parts of the design or run the BLISS signature algorithm with different parameter sets.

Reference Implementations Issues

Two issues needed to be addressed before evaluating the side-channel characteristics of the design. The first one involved the communication with the used Keccak implementation mid-range core by Bertoni et al. [6]. Upon feeding message blocks to the Keccak module, the signal to make Keccak absorb the message needs to be a logic `1` for a specified number of clock cycles. However, this was initially not the case, neither for feeding the message μ, nor for feeding the vector $\mathbf{u}$ as indicated in 3.2. This error led to a missing transformation

resulting in a different initial state of Keccak. Due to the modules being reused for signing and verifying, thus making the same error on both sides, the signer and verifier were compatible with each other, but not to the adapted BLISS reference implementation [16].

Secondly, the rejection sampling did not work if the number of cores used for the sparse multiplication was set to small values. In case of a rejection, several modules are reset and made ready to restart the signing process based on the same, internally cached, message. However, due to some modules not being appropriately reset, two effects occurred: First, some signatures were initially rejected and then immediately afterwards accepted without being recalculated. This prevented a proper distribution of the resulting signatures. Second, the implementation would come to a hold, because certain modules were reset and waiting for an input while others were waiting on signals from the former.

Contribution to VHDL implementation

Starting with the signer, verifier and BRAMs as the initial building blocks, the project has been made compatible with the SAKURA-G FPGA board. The LBUS protocol and interface, as described in Section 5.3, have been adapted and extended for the BLISS implementation.

The issues described in Section 5.2.1 have been fixed, making the design compatible with any number of cores configurable for the sparse multiplication. Furthermore, this made the signatures created by the FPGA implementation compatible with the reference software-implementation from Ducas and Lepoint [16]. For this, the software implementation has been adapted to use Keccak as its hash function instead of SHA-512. Testbenches were added that write signatures computed in the simulation to files compatible with the reference implementation as well.

5.2.2 Top-Level Module

A top-level view of the hardware implementation can be seen in Figure 5.2. The top module consists of six main blocks. The added LBUS host interface handles the LBUS communication with the second FPGA and subsequently the desktop PC that interprets the received commands. Furthermore, the BLISS signer and the BLISS verifier modules are located here. The signatures are stored in the three dual-port block RAMs (BRAMs) *Z1*, *Z2* and *C*. One port is shared between the LBUS interface (to read and write its contents over the bus) and the BLISS verifier (to read and verify the signature), the BLISS signer entity has full control over the second port. Some additional logic situated at the top level to control the access to the shared port of the BRAM to ensure it is used exclusively by either the LBUS host interface or the BLISS Verifier.

5.3 Local BUS Interface

The SAKURA-G FPGA board comes with a demo showcasing the communication between the two FPGAs on the board [56]. In this example code, the smaller FPGA offers a USB interface to receive and send data to a desktop PC and also a Local BUS (LBUS) interface to forward data to the coprocessor. This approach offered a solid basis and thus hasn't been altered.

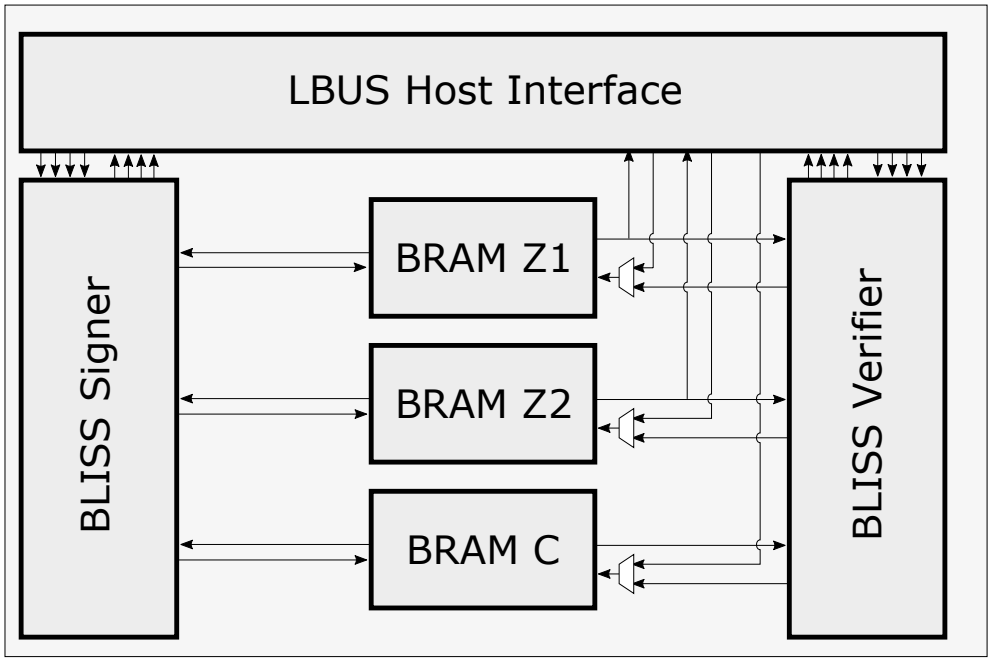


Figure 5.2: High level structure of the FPGA implementation with the LBUS interface and the sign, verify and RAM blocks.

Aiming to make the BLISS implementation available as a coprocessor to a Desktop PC, the code for the Xilinx Spartan-6 XC6SLX75 has been replaced by the BLISS signature hardware implementation with an additional LBUS interface. Because of the example implementation for the main FPGA being developed in Verilog for a different cryptographic primitive, the implementation of the LBUS on the BLISS coprocessor required some adjustments to be fully functional. With BLISS being implemented in VHDL, the LBUS protocol was rewritten in VHDL and extended with commands and word sizes more suitable for the BLISS signature algorithm.

As seen in Table 5.1, the implementation relies on an external clock generated by the controller FPGA to drive the chip and the LBUS. The smaller FPGA is also capable of resetting the BLISS coprocessor by setting the `rstn` pin to a logic 0.

Of the remaining nine signals, `aemp` and `aful` are not used. If the device is in the process of creating a signature, the `rdy` line is pulled to a logic `0`.

The triple consisting of `wd`, `we` and `ful` are used to for sending data to the device. While `ful` is used to show that the coprocessor is ready to receive data, `wd` and `we` are used in conjunction with the control FPGA to signal that the data on the bus is in fact valid.

Similarly, the `rd`, `re` and `emp` data lines are used for reading data from the main FPGA. The signature device uses the `rd` lines to expose the requested data to the bus and notifies the controller using the `emp` signal. In return, the controller is then using `re` for confirmation that the data has been read successfully.

Name	Direction	Size (bits)	Description
`rstn`	in	1	Reset pin, active low
`clk`	in	1	External clock
`rdy`	out	1	Logic 0 = Device is busy Logic 1 = Device is ready for commands
`wd`	in	8	Write data - the data written to the device
`we`	in	1	Write enable Logic 0 = `wd` is invalid Logic 1 = `wd` is valid
`ful`	out	1	Logic 0 = `wd` is ready to write Logic 1 = `wd` is not ready to write
`aful`	out	1	Not used
`rd`	out	8	Read data - the data read from the device
`re`	in	1	Read enable Logic 0 = `rd` has not been read Logic 1 = `rd` has been read
`emp`	out	1	Logic 0 = `rd` holds valid data Logic 1 = `rd` holds invalid data
`aemp`	out	1	Not used

Table 5.1: Configuration and functionality of the pins dedicated to the LBUS. The direction refers to the point of view of the main FPGA running the BLISS signature algorithm.

5.3.1 Communication Wrapper

For the measurements to take place, it is necessary for the PC to communicate with the signing FPGA. Because the controller FPGA is connected over USB and needs to receive the commands to be sent over the LBUS, a wrapper is required that can be addressed easily. This wrapper is a DLL that allows sending data to and receiving data from the BLISS signing FPGA. It offers wrapper functions for setting the key, starting the signing process for a given message and reading contents of the BRAMs to name a few. Internally it creates the necessary LBUS byte stream, that is sent to the controller FPGA over the USB interface.

Additionally, the data from the replies is also extracted from the byte stream and returned to the callee.

5.3.2 Commands

Using the LBUS protocol described in Section 5.3, data can be written to or read from defined addresses on the BLISS signature device. The available addresses are documented in Table 5.2. The first nibble is used to split the address space into separate groups. It should be noted that `BRAM-Z1` and `BRAM-Z2` are only readable whereas `BRAM-C` is also writeable. This is due to the measurement setup described in Section 5.4 that doesn't aim for verifying an externally generated signature. Moreover, the setup allows for a mode only

performing the sparse multiplication, justifying the need to make the `BRAM-C` externally writeable.

The write data command consists of five bytes and the read command of three bytes. In both cases, they are sent over the `wd` bus. The first byte is either `0x00` to read or `0x01` to write. For reading as well as writing the next two bytes indicate the affected address in the big-endian format. In case a write command is transmitted, two more bytes in the big-endian format are appended containing the data to be written. An illustration of an LBUS read and write can be seen in Figure 5.3.

The size of the actually available address space available for `0x1` to `0x5` and `0x8` depends on the size of n of the BLISS parameter set. The message memory is configured for a word size of 64 bits. With the LBUS only being able to transmit 16 bits at once, it is necessary to write to 4 subsequent addresses before the word gets committed internally. Also, for setting the message, it is necessary to send `0x1` to the address `0xC..2` to signal the end of the transmitted message.

As indicated in Table 5.2 addresses starting with `0xC` act as command or status registers. Most notably are `0xC..3` to start signing the message that has previously been transmitted, `0xC..4` to reset all submodules to their initial state and `0xC..6` to verify the signature that is located in the `BRAM-Z1`, `BRAM-Z2` and `BRAM-C`. The signature located in the respective BRAMs is generated by the signing module, but this functionality could be extended easily to signatures written externally to the addresses `0x1`, `0x2` and `0x3`.

Address	Read/Write	Function
`0x1...`	R	$\mathbf{z1}$ stored in `BRAM-Z1`
`0x2...`	R	$\mathbf{z2}^{\dagger}$ stored in `BRAM-Z2`
`0x3...`	R/W	$\mathbf{c}$ stored in `BRAM-C`
`0x4...`	W	Secret key $\mathbf{s}_1$
`0x5...`	W	Secret key $\mathbf{s}_2$
`0x6...`	W	Message μ RAM.
`0x7...`	R	Returns defined pattern over LBUS for testing
`0x8...`	W	Set matching public key for the secret key $\mathbf{S}$
`0xC...`	R/W	Control Register
`0xC..0`	R	Status of signature generation (0 = not ready, 1 = ready)
`0xC..2`	W	Message μ finished
`0xC..3`	W	Start signing
`0xC..4`	W	Reset device
`0xC..5`	W	Setting mode to sign/verify
`0xC..6`	W	Verify signature of `BRAM-{Z1,Z2,C}` content

Table 5.2: Overview of the readable/writeable addresses available on the BLISS signature device over the LBUS.

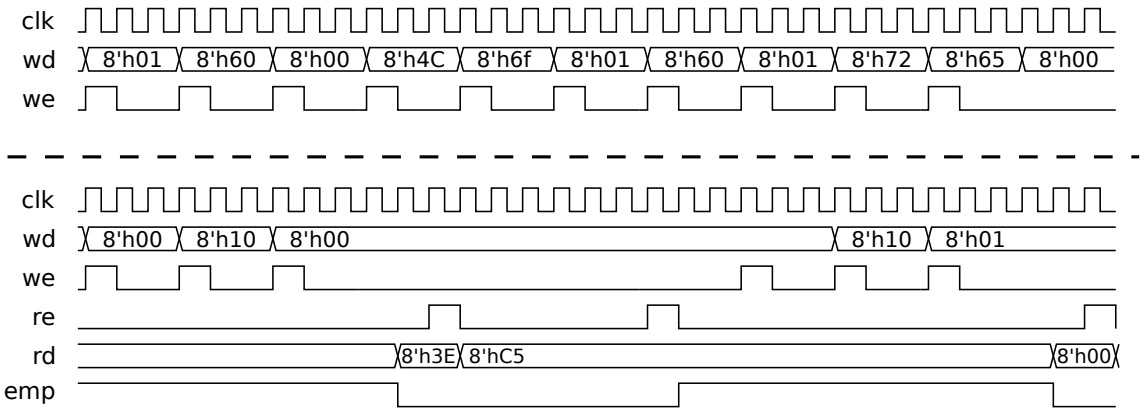


Figure 5.3: Waveform of a write (top) and a read (bottom) operation using the LBUS protocol. In the write example the `ful` signal has been omitted as it's constant due to signature device only taking one cycle to process the information in this example.

5.4 Measurement Setup

A general overview of the measurement setup is given in this section. An illustration of the setup can also be seen in Figure 5.4. Fundamentally, the setup involves a desktop PC, the SAKURA-G board as well as a multichannel oscilloscope. For the oscilloscope, a *Picoscope 6404C* was used, offering four channels with a combined sampling rate of 5 GS/s. In this setup, each channel can sample with 1.25 GS/s. On the PC a Matlab instance controls the measurement procedure and later also the storing and pre-processing of the acquired traces. The wrapper described in Section 5.3.1 is loaded by Matlab and used to send commands to the device and receive the results afterwards. The FPGA is initialised with a pre-generated signing key using the wrapper. The plaintexts are pseudorandom messages following a discrete uniform distribution and generated by Matlab. For each measurement a new message is sent to the device, the signing key remains the same during all measurements. The FPGA controller then forwards the messages over the LBUS to the BLISS signature FPGA.

As seen in Figure 5.4, the oscilloscope is connected to the SAKURA-G board over three probes on three channels. The leftmost connection to the board in the illustration is connected to an AD8000 amplifier as seen in Figure 5.5. This amplifier boosts the voltage drop of the supply voltage over the resistor R_{12} with an impedance of 1Ω. The measured output of the amplifier is inversely proportional to the power consumption of the FPGA itself. The probe is connected to the measurement point J_3.

Two more channels are connected to the GPIO pin header and act as trigger signals. One connector is used as a trigger for the oscilloscope and to signal the start and the end of the entire sparse multiplication. The other one is used to indicate the κ subsequent additions during the computation of a single coefficient during the sparse multiplication. The traces measured by the oscilloscope are then fed back to Matlab for storage and further processing.

Due to the BLISS signature algorithm performing a rejection sampling, the oscilloscope cannot be used in single-shot mode, as the first trace might not correspond to the power consumption connected to the signature output. Furthermore, the number of rejections is not known in advance. To circumvent this the oscilloscope was configured in *sequence-*

mode with a sufficiently large number of repetitions, that is most likely not exceeded. The measurement is then aborted after receiving the signature from the FPGA and the last element of the buffer used as the power trace linked to the signature output.

For the evaluation, sets of 10,000 traces were captured in different configurations of the implementation. Depending on factors like the number of cores configured for the sparse multiplication or the usage of only the module performing the sparse multiplication, the time needed for capturing the data ranges from about 3 to 12 hours.

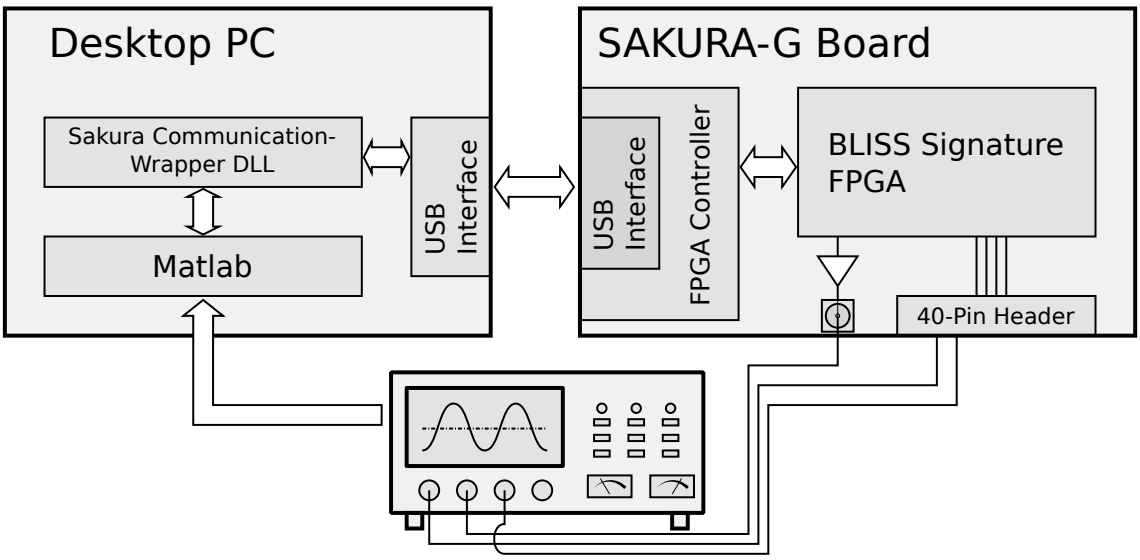


Figure 5.4: Setup for the automated measurement of the BLISS signature hardware implementation.

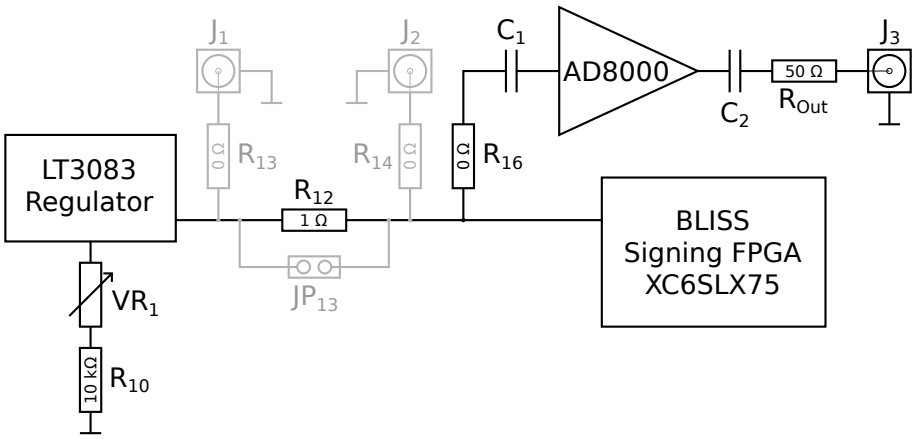


Figure 5.5: Schematic of the measurement point reading the power consumption of the signing FPGA.

5.5 Attack Description

This section describes the side-channel attack on the implementation of the BLISS signature algorithm on an FPGA. First, the selection of the intermediate value for the power analysis attack is discussed in Section 5.5.1.

For the practical attack, the implementation has been restricted to perform a sparse multiplication only, configured to one core for $\mathbf{s}_1$, with the computations involving $\mathbf{s}_2$ deactivated. The attack then targets only the secret key $\mathbf{s}_1$, to simplify the setting. The indices of the sparse vector $\mathbf{c}$ are generated by Matlab before performing the measurement and written to the `BRAM-C` over the LBUS. In this scenario, the sparse multiplication performed by the chip is precisely the same as the one used during the actual signing process, but without the noise introduced by the computation of the squared norm and scalar product which are computed in parallel on the FPGA.

The used set of parameters is BLISS-I from Table 3.1, most importantly using $n = 512$ as the size of the secret key, and $\kappa = 23$ as the Hamming weight of the sparse vector $\mathbf{c}$.

5.5.1 Intermediate Value Selection

Based on the overview of possible weak spots given in Section 4.7, the sparse multiplication has been selected as the target of the SCA-attack. The sparse multiplication poses as an ideal target, as it directly involves the secret key. Furthermore, this operation is widely used in efficient lattice-based primitives and findings from this attack are likely to apply to other algorithms as well.

As discussed in Section 3.5.2, the FPGA design computes one output coefficient per core at once over κ clock cycles by adding or subtracting one element of the secret key $\mathbf{s}_i$ per cycle. Figure 5.6 shows 100 overlayed power traces of the sparse multiplication without any pre-processing. The purple line indicates the start and end of $\kappa + 1$ cycles.

With the attacked step of the BLISS algorithm known, the leveraged intermediate value needs to be further narrowed down for a successful attack. Espitau et al. [17] propose to attack the intermediate result of the sum of the secret key elements. However, preliminary tests showed only very limited leakage from this operation, because for storing the result only a single 7-bit register is used in the hardware implementation. Additionally, predicting the i-th partial sum requires guessing i key coefficients. Due to the sparse multiplication always using different elements of the secret key, anything after the first element cannot be predicted efficiently.

More leakage was obtainable from accesses to the secret value itself. Each core involved in the sparse multiplication holds a full copy of the secret key $\mathbf{s}_i$. This is necessary due to the implementation of the sparse multiplication as described in Section 3.5.2 and each core computing a different output coefficient requiring access to a different element of the secret key $\mathbf{s}_i$. Due to the secret key being stored in a block RAM the leakage has shown to be more prominent. Also, there is no need to include assumptions on the likeliness of transitions as it would be necessary for attacks involving the result register. This results in choosing the loading of the elements of the secret key $\mathbf{s}_i$ during the sparse multiplication as the target for the SCA attack.

5.5.2 Pre-processing

Before attacking the device, a certain degree of pre-processing has to be done to optimise the captured data. Each trace consists of the computations of multiple output coefficients,

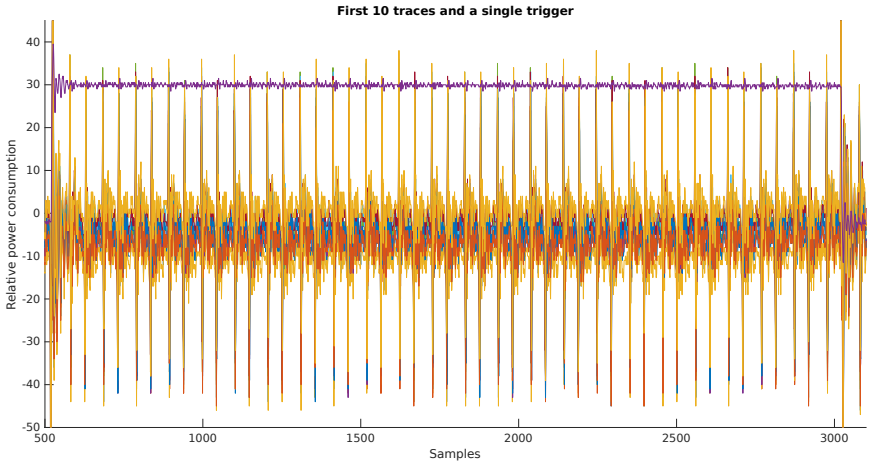


Figure 5.6: Power consumption of the computation of the first output coefficient. The off-center purple trace shows a trigger signal indicating $\kappa + 1$ cycles, with $\kappa = 23$, during the multiplication.

each with κ additions and κ loads of secret key elements. For the further analysis, the goal is to extract information from each of the additions, independent of their location in the trace. This requires a certain amount of pre-processing to attribute for changes over time.

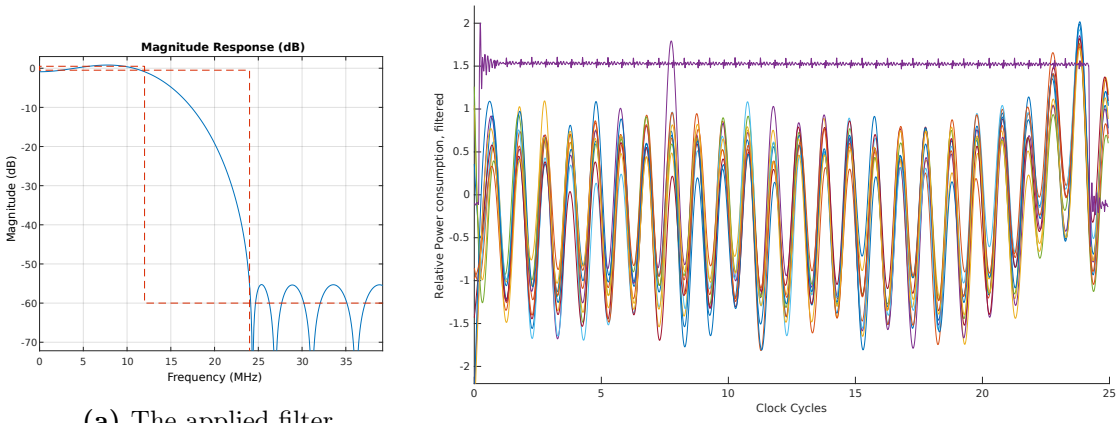
First of all, the aforementioned second trigger signal indicates the interval of $\kappa + 1$ additions and is used to split the traces into equally sized chunks. These chunks contain the power information of the κ additions that lead to one coefficient of the output, the result of the multiplication. The number of multiplication intervals available in each trace is determined by the parameter n and the number of cores available in the design, due to the cores processing the multiplications in parallel. It should also be noted that each addition takes one clock-cycle. Therefore the computation of one output coefficient takes 23 clock cycles, making a further separation into the addition operations merely a matter of parting the interval in $\kappa + 1$ fragments.

Next, each chunk is adjusted such that the mean over its values equals zero. This counteracts offsets in the power consumption observed when doing many measurements repeatedly over a more prolonged timespan.

The traces were captured using a sampling rate of 1.25 GS/s. With the signing FPGA only running at 12 MHz, the traces contain a lot of high-frequency noise. Therefore, a low pass filter, as seen in Figure 5.7a, is applied with a passband frequency equal to the clock frequency (12 MHz) and twice that for the cut-off frequency (24 MHz). The pre-processing results in multiplication blocks without high-frequency noise, normalised to a mean of zero in each block as seen in Figure 5.7b

5.5.3 Visual Inspection and Power Model

After pre-processing the traces, the data can be checked for power leakage caused by the secret key. The signature contains the indices of the sparse vector $\mathbf{c}$, which is used during the sparse multiplication to determine the involved elements of the secret key $\mathbf{s}_1$ of Algorithm 3.2.



(a) The applied filter

(b) Filtered power traces

Figure 5.7: Filter used to preprocess the traces. Exemplary filtered traces like are shown on the right.

For a first analysis, a known secret key $\mathbf{S}$ is used. The 256th out of the 512 output coefficients is analysed by linking the power trace to the element of the sparse vector. This pair is then grouped based on the value of the secret key element involved in the first addition of the 256th output coefficient. These output coefficients will also be referred to as *multiplication blocks* when discussing the traces or operations to calculate the output coefficient. Determining the involved elements of the key is done by using a slightly modified version of Algorithm 3.4, returning only the secret key indices involved in the computation. A coefficient in the middle has been chosen, because as seen in Algorithm 3.4 the index of the output coefficient determines if the secret key element is inverted or not. With later coefficients the number of inverted elements increases. On the halfway, the number of flipped and non-flipped key elements is about the same for a sufficiently large number of signatures.

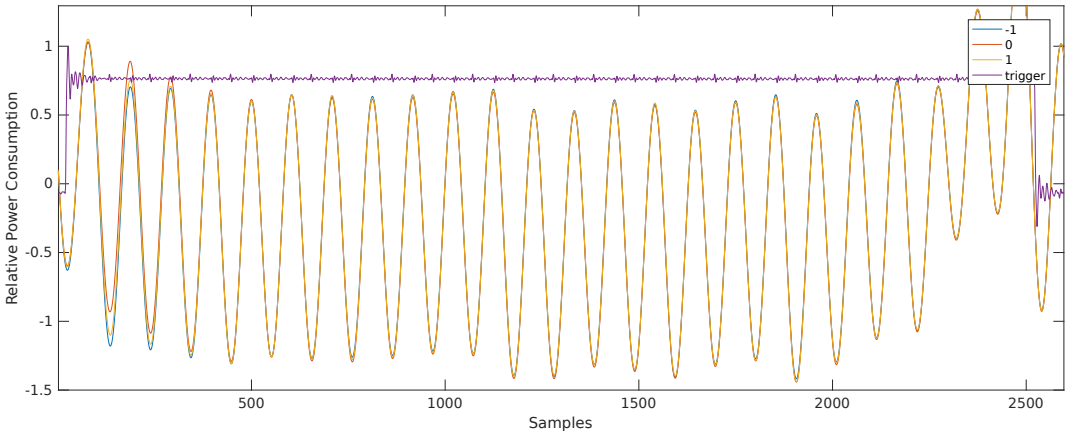


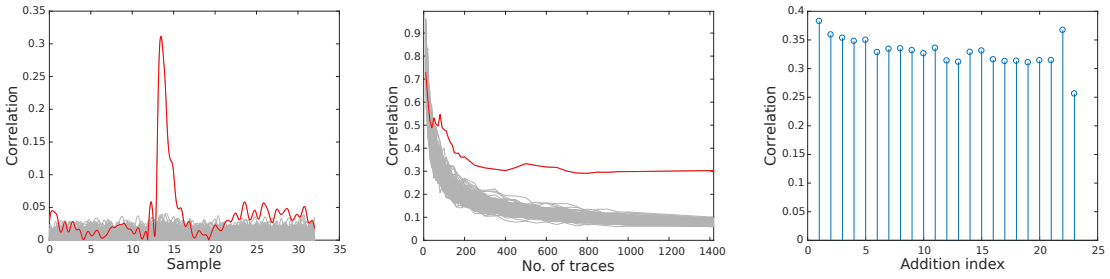
Figure 5.8: Mean value of the traces when grouped together based on the the value of the first involved secret key element. The purple line indicates the trigger signal marking the start and end of a single sparse multiplication.

First, the traces are grouped based on the first secret key element and a mean trace based on each category is calculated. For attacks on $\mathbf{s}_1$ the traces are then grouped into -1,0 and 1, the three possible values of each secret key coefficient. The result of this step for the targeted multiplication, as indicated by the purple trace, can be seen in Figure 5.8. At the beginning of the trace some data dependency, as indicated by the distance between the three means, can be observed. This power leakage vanishes over the following cycles, due to the grouping not being correct for the subsequent additions.

With this first indication on the power leakage of the involved key elements, further tests are performed.

DPA on Same Output Coefficient, Same Addition

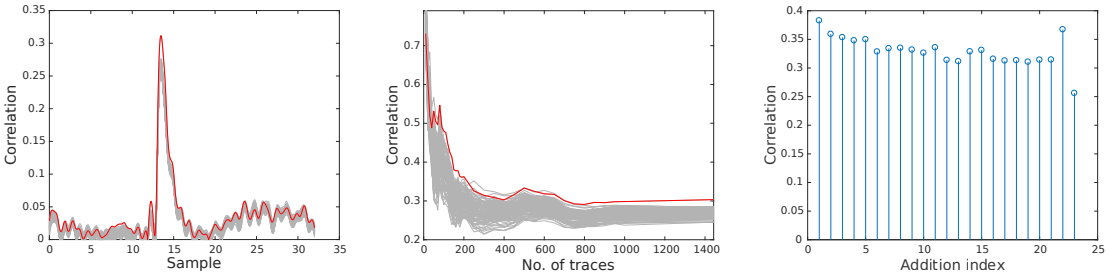
After this first visual inspection, the traces are checked for leakage using methods from differential power analysis (DPA) described in Section 4.5. For the first analysis, the same output coefficients and addition index are used throughout the traces, so no alignment of the traces is needed. However, it is not possible to use a straight-forward DPA because every key coefficient can be involved and it would be necessary to account for all keys. Instead, at first 99 random keys are generated to be used for analysis. Additionally, the analysis is performed with 99 keys derived from the correct secret key by inverting a random number of key elements. These derived keys are similar to correct one in terms of the location of their non-zero key elements, but cannot be used to create a signature that verifies under the same public key. Next, for each signature, the sparse vector $\mathbf{c}$ is used to compute all involved secret key elements of the multiplication of the 256th output coefficient. This results in a matrix with a size of 100 keys times 23 additions times 10000 traces. In other words, it holds the involved key value for each addition during the 256th multiplication for each captured trace and acts as the basis for the power model. As seen in Figure 5.8, the power consumption of elements being -1 is higher than for 1, higher than for 0. Therefore, a power model based on the Hamming weight with a key element 0 translating to 0, 1 to 1 and -1 to 2 is derived from the matrix as mentioned above for the continued evaluation.



(a) Correlation along samples **(b)** Distinction of keys **(c)** Correlation per addition

Figure 5.9: Result of the DPA when using random, fake keys and applying the power model to the same multiplication block and the same addition with index 13 in each trace.

The power model for one specified single addition is then evaluated using the correlation coefficient as given in Equation (4.10) with the captured traces during the 256th multiplication. The result for the correct and 99 random keys can be seen in Figure 5.9. As seen in



(a) Correlation along samples **(b)** Distinction of keys **(c)** Correlation per addition

Figure 5.10: Result of the DPA when applying the power model to the same multiplication block and the same addition with index 13 in each trace.

Figure 5.9a the correct key, marked in red, shows a significantly higher correlation than the random keys at the 13th sample. Furthermore, as seen in Figure 5.9b, starting at about 50 traces the correlation of the correct key can be clearly distinguished from the random keys. Finally, Figure 5.9c shows the maximum correlation for each addition during the multiplication. The correlation is about 0.25 to 0.38 depending on the addition involved.

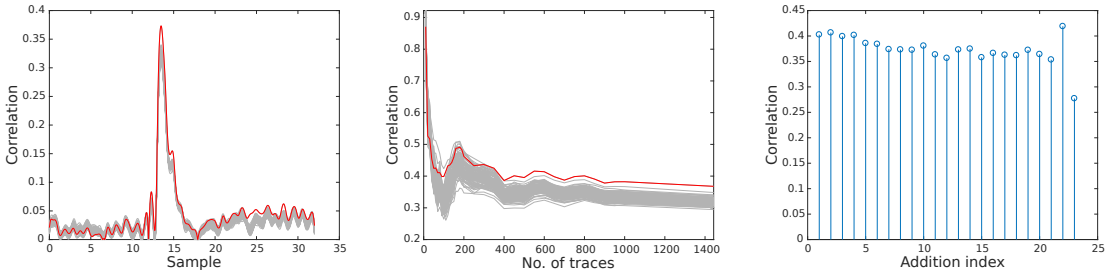
The results of the analysis using the derived keys can be seen in Figure 5.10. Figure 5.10a displays the correlation of the power model of the 13th addition with the measured power consumption of the multiplication. This also shows a significantly higher correlation at about the 13th sample, giving a good indication of the power leakage. However, the correlation is similarly high for fake keys, because the locations of the key elements being zero are identical even for similar keys. As seen in Figure 5.10b, starting at about 800 traces the correlation of the correct key can be clearly distinguished from the fake keys. Finally, Figure 5.10c shows the maximum correlation for each addition during the multiplication. The correlation is about 0.25 to 0.35 depending on the addition involved. This also fits the other two graphs, showing a similarly high correlation.

The DPA shows a data dependency between the operations at the same point in time along the power trace. However, a single trace captures the sequential computation of many output coefficients, depending on the number of cores configured in the design. To ensure that more information can be extracted from a single trace, it needs to be shown that the power leakage is time-invariant, i.e. that each addition and multiplication block leak the involved secret key the same way, independently of their location in the trace.

When doing a visual inspection of the traces, a slight offset can be observed when inspecting the full trace. Therefore each trace is normalised by subtracting the mean of all traces at any given point from each trace. Any further descriptions evaluated combinations in this section are based on these normalised traces. Due to derived keys requiring more traces to be distinguished from the correct key than the random keys, the derived keys are used for the subsequent tests.

DPA on Varying Output Coefficient, Same Addition

For this evaluation, the same method is applied as before, but instead of using the same multiplication index for each trace, a random index is used and a power model is built. Basically, this overlays multiplication blocks from different locations along the time axis to



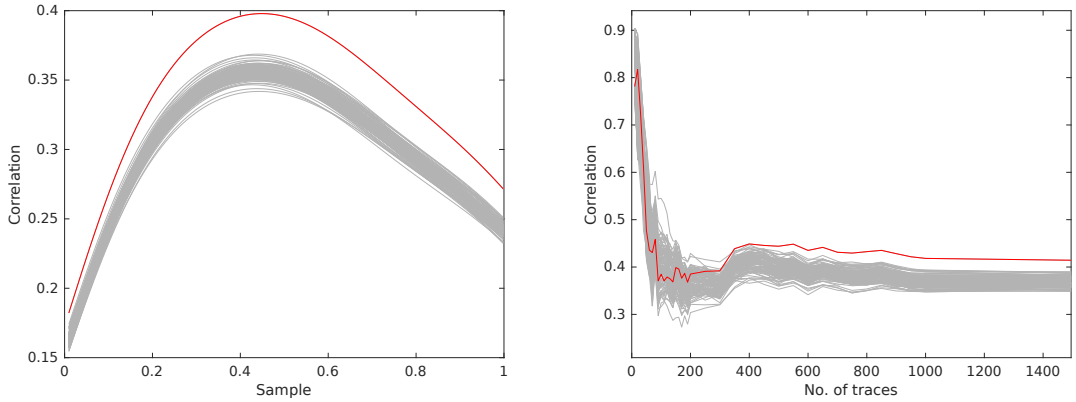
(a) Correlation along samples **(b)** Distinction of keys **(c)** Correlation per addition

Figure 5.11: Result of the DPA when applying the power model to different multiplication blocks but the same addition with index 13 in each block.

ensure the model being time-invariant. Additionally, the effectiveness of the attack can be increased drastically because instead of getting information on the secret key only once per trace, the number of recoverable leakage points is multiplied by the number of sequentially computed coefficients.

The result can be seen in Figure 5.11. With Figure 5.11a showing a similarly high correlation as for the fixed multiplication blocks, Figure 5.11b indicates that in this scenario only about 400 traces are needed to distinguish the correct- from the fake key. However, it should be pointed out that besides the use of different output coefficients, the traces were also made mean-free improving the results of the correlation. Figure 5.11c also shows a comparable correlation at the location of the additions.

DPA on Same Output Coefficient, Varying Addition



(a) Correlation along samples **(b)** Distinction of keys

Figure 5.12: Result of the DPA when applying the power model to the same multiplication block but different additions in each trace.

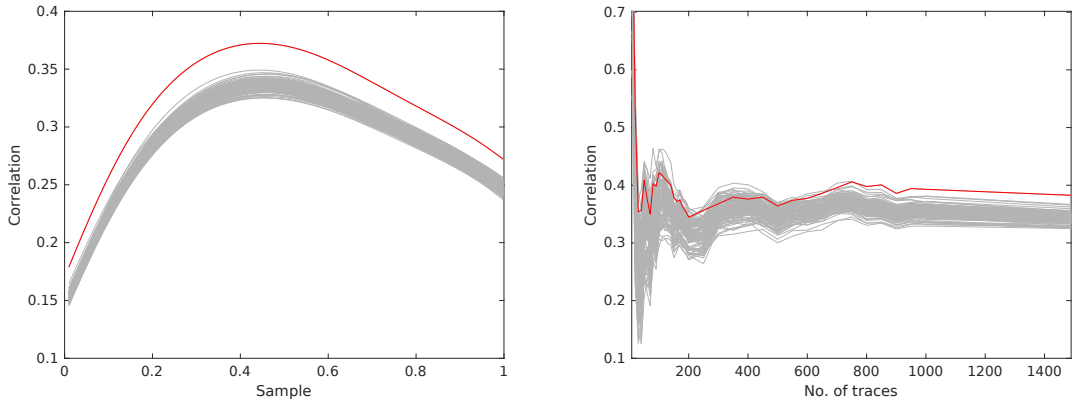
Another improvement can be made by also utilising information on the additions independently of their position during the multiplication, i.e., the element of the secret key loaded in the first round leaks information the same way as the item does for the last ad-

dition of the same multiplication block. To verify this, a random addition index is selected from the same multiplication block of each trace. Again the involved secret key element for each of the fake keys and the single correct key are determined and used to create a power model for this scenario.

The result can be seen in Figure 5.12. Due to the power-model correlated with only one addition, the appearance in Figure 5.12a looks different than in the previous examples and covers only a single clock cycle. However, the peak in correlation with about 0.4 is similar to the previous scenarios. Moreover, the number of needed traces to distinguish the fake keys from the correct key at about 450, as seen in Figure 5.13b, shows a comparable result as well.

This shows that the power leakage of additions at different positions of the same output coefficient can be combined and show a similar correlation to the previous experiments.

DPA on Varying Output Coefficient, Varying Addition



(a) Correlation along samples **(b)** Distinction of keys

Figure 5.13: Result of the DPA when applying the power model to different multiplication blocks but different additions in each trace.

A final configuration needs to be performed to check all combinations. For this scenario, for each trace, a random output coefficient is used. Moreover, this is combined with a random addition and aims at showing that the power information of any addition of any multiplication block can be combined with any likewise random addition from any other trace.

The result for this can be seen in Figure 5.13. Like in the previous example Figure 5.13a only shows the interval of a single addition horizontally. The displayed correlation is similarly high as in the previous examples. However, with about 800 needed traces, as seen in Figure 5.13b, to clearly distinguish fake keys from the correct key in terms of their correlation it is slightly worse than the results above.

Finally, this shows that from a single trace, leakage can be obtained from any addition of any output coefficient. This means that with a configured $n = 512$ and $\kappa = 23$, each power trace gives 11776 leakage points in total for a configuration using only one sparse core for the multiplication. Alternatively, this can be seen as for a given trace, κ leakages can be obtained for each secret key element. However, this is also dependant on the number

of cores used for the multiplication. If more than one unit is used, the number of leakage points is divided by the number of cores, due to the operations being processed in parallel.

Although a power leakage is visible in all of the presented scenarios, a DPA cannot be used to recover the secret key, because all key coefficients are involved in the operation.

5.5.4 Clustering-based Attack Evaluation

The differential power analysis shows a data dependent power leakage of the traces. To recover the secret key elements, a clustering based attack is now presented. The number of traces needed to perform the attack is also evaluated.

As a first step, the traces are compressed based on the results of the DPA with the randomly selected multiplication and addition as described in Section 5.5.3. The results of the DPA shows the location of the highest correlation of each addition, as seen in Figure 5.13a. Therefore, the compression picks a single measurement sample for each addition of each multiplication to simplify the processing. The compressed traces can then be used to recover the secret key and evaluate the number of needed traces.

For a first inspection of the clustering-based attack, a total of 10000 traces is used. Similarly to the DPA performed previously, a variation of Algorithm 3.4 is used to determine the involved secret key elements and is applied to the compressed power traces to assign them to the correct indices. This information is then linked to the samples of the compressed power traces and grouped based on the index of the secret key element. Next, the mean value for each index of the secret key is computed. The result is a vector with a single value for each secret key element. The mean value of each secret key element can be seen in the histogram in Figure 5.14. The graph shows the grouped distribution of the mean value determined for each of the n coefficients of the secret key. The colour of the bin indicates the related value of a known secret key. Therefore, the distribution as seen in Figure 5.14 is well suited to classify the traces based on their nearest cluster. However, it should be noted that each cluster in Figure 5.14 is normalised to 1.

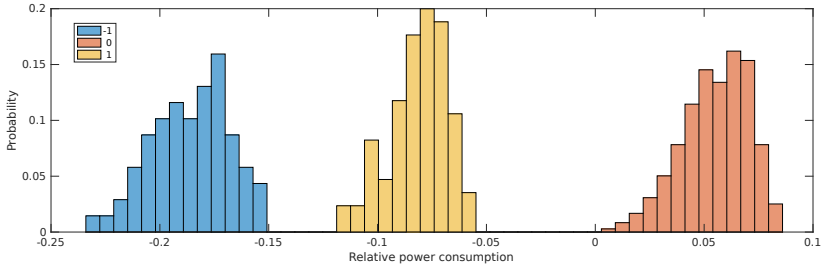


Figure 5.14: Distribution of the mean values of the traces with their colour demonstrating the respective key value in the secret key.

The attack on the implementation works by combining the findings of the DPA with the distributions from Figure 5.14 and information on the used BLISS parameter-set. As described in Section 3.2.1, the number of non-zero key elements in $\mathbf{s}_1$ is given by $\lceil \delta_1 n \rceil$. First, the means for each index of the secret-key are sorted by their value. Based on the DPA, higher values correspond to a 0, lower values to a 1 and the lowest to -1. Therefore, the highest $\lceil \delta_1 n \rceil$ indices can then be classified as the zeros of the secret key. The remaining $n - \lceil \delta_1 n \rceil$ elements are ± 1. However, all values below a certain threshold can be classified

as -1, the values above as $+1$. Due to the low number of elements and their appearance in the right order, all possible $n - \lceil \delta_1 n \rceil$ thresholds can be tested. The resulting key candidates can then be verified against the public key.

To verify the effectiveness of the attack, an evaluation process is performed to check how many traces are needed to correctly distinguish a certain percentage of key elements. Therefore, an increasing number of random compressed power traces is used. On the one hand, a classification based on their exact value and on the other hand a classification between zero or not zero is performed. As before, the values are grouped according to their secret key index and then reduced to a mean value for each index.

With the used parameter-set this results in 154 key elements being ± 1 and 358 elements being 0. With the histogram in mind, the 358 indices with the highest mean value are selected to determine the key elements being zero. The exact split between 1 and -1 is unknown, i.e., all elements -1 are smaller than the $+1$. Therefore, it is tested if the order in which the secret key elements appear is correct. This is then repeated multiple times with different sets and an increasing number of traces to determine the accuracy of the method.

The same process is repeated but with the goal to only distinguish between key elements being zero and not zero to lower the number of required traces. As this only reveals the locations of the zeros and the magnitude of the key elements, the sign of the non-zero key elements needs to be recovered. The recovery is performed by using the method by Pessl et al. [37] described in Section 4.7.1. It allows a key recovery if only the magnitude of each key element is known. Both classifications are then checked by comparing the result to the known secret key.

The results can be seen in Figure 5.15. It shows the number of traces needed to distinguish between $+1$, -1 and 0. This classification needs about 36 traces. However, a 95% accuracy can already be achieved with half of that. As expected the number of traces to distinguish key elements being zero from those being non-zero is visibly lower. The classification in only two groups only requires ten traces which can be further lowered to about seven traces for 95% accuracy.

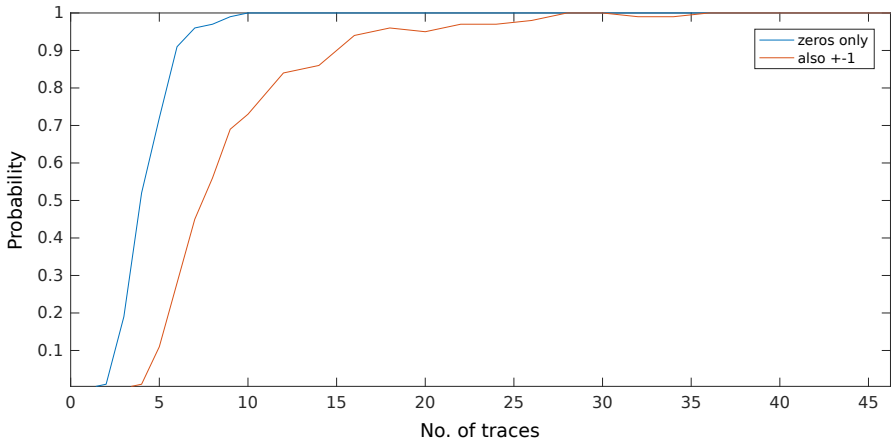
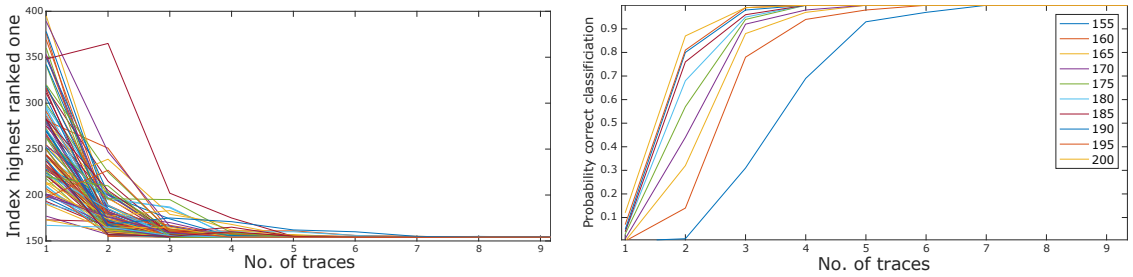


Figure 5.15: Probability of successful classification of all zeros, and of all zeros as well as the signedness of the elements with a magnitude of 1.

As an improvement, the number of required traces can be even further reduced by lowering the needed accuracy of the classification. The clustering-based attack uses a certain number of highest indices sorted by their linked mean value. As observed in Figure 5.16a starting at three traces, all key elements with a magnitude of 1 are within the 200 lowest positions by doing this classification. The method proposed by Pessl et al. [37] can be adopted for such a scenario. For the attack, it is sufficient to verify a set of indices which contain all ± 1 elements. This set can be a superset of the ± 1 indices, i.e. larger than $\lceil \delta_1 n \rceil$. However, using a larger set increases the difficulty of the SVP to be solved. This is further elaborated in Figure 5.16b. The plot shows the probability depending on the number of traces for a successful classification if the requirement is to determine a superset with a size of 155-200 elements containing all of the 154 elements being ± 1.



(a) Accuracy of classified ones

(b) Success rate with less strict requirements

Figure 5.16: Index of the classified ones depending on the number of traces on the left and a comparison of required traces such that all ones are within the first 155-200 most likely positions.

By selecting 175 elements, only four traces are necessary for an almost certain classification. However, by using this method additional processing is required, which takes a few hours for a set size of 175 and several hours to a few days for 180 elements, indicating that the time required for solving the resulting SVP increases drastically. Therefore, this technique can be used to decrease the number of required traces at the cost of additional computations during the key recovery. It should be kept in mind, that it is only feasible up to a certain point and a trade-off has to be made as reducing the number of required traces to three or halving them to two, decreases the success rate and more importantly increases the computational effort to a multitude.

Chapter 6

Countermeasure Evaluation

In this section, the previously attacked unprotected implementation of the sparse multiplication is analysed and potential countermeasures are discussed. One of the countermeasures, namely masking, is then implemented and the protected hardware implementation evaluated. The result of the evaluation then shows the effectiveness of the countermeasure.

In Section 6.1, properties of polynomial blinding, shuffling of coefficients and masking are explained and their advantages and disadvantages are discussed. Next, the implementation of the proposed masking scheme in hardware is presented and also the necessary changes to the existing hardware design are reviewed in Section 6.2. Furthermore, it goes into detail with the costs linked to the protected implementation in terms of additional circuitry required and also potential time penalties that come with the countermeasure. Finally, in Section 6.3 the same attack consisting of a differential power analysis and a template attack as performed on the unprotected implementation is executed on the protected implementation, the effectiveness of the countermeasure is discussed and also comparisons between the attacks are made.

6.1 Applicable Countermeasures

The evaluation of side-channel leakage as described in Chapter 5 demonstrated a significant leakage of the unprotected computation of the sparse multiplication. This raises the need for means to prevent the operation from leaking secret information through a power side-channel. Therefore, this section will discuss ways to protect the hardware implementation and their implications on the design.

6.1.1 Polynomial Blinding

As proposed by Saarinen [47] and already described in Section 4.7.5, polynomial blinding represents an applicable countermeasure to protect the sparse multiplication $\mathbf{s}_i\mathbf{c}$ with $i \in \{1, 2\}$. The presented mechanism aims at protecting the successor of the BLISS signature algorithm BLISS-B [14]. Due to the vast similarities between both schemes, the attacked operation works the same way in both algorithms and can be used for BLISS as well.

Recall that the described countermeasure is based on multiple operations. It is necessary to sample additional constants a, b, multiply the constants with both multiplication

terms $\mathbf{s}, \mathbf{c}$, compute the product of the multiplication and perform another multiplication:

$$\begin{aligned} \mathbf{h} &= a\mathbf{s} \cdot b\mathbf{c} \\ \mathbf{s} \cdot \mathbf{c} &= (ab)^{-1}\mathbf{h} \end{aligned} \tag{6.1}$$

Additionally, a circular shifting of the polynomial has to be performed with the amount of shifted positions determined at runtime. However, this section only discusses the blinding by multiplication, because shifting is only a simplified version of shuffling, which will be discussed later.

The multiplication with the random constants affects both the secret key $\mathbf{s}_i$ and the sparse vector $\mathbf{c}$. Their sparsity remains, i.e., most of the coefficients are still 0. Due to the multiplication with the random constant, the non-zero elements of $\mathbf{c}$ are no longer equal to one. Therefore, the current multiplication algorithm, consisting of κ sequential additions of determined secret key elements, has to be altered increasing the complexity. However, alternatively to the regular domain, the multiplication can be performed in the NTT domain, requiring additional pre-processing, but yielding faster multiplications [47].

This type of countermeasure is not very well suited for the hardware implementation, due to the described properties, the introduction of three additional multiplications, the increase in runtime and the additional circuitry required. Additionally, as described by Espitau et al. [17] attacking the blinded multiplication is still feasible because the unblinded sparse vector $\mathbf{c}$ is publicly known and the range of possible parameters for the involved shifting and the random multiplicative constant is small. It should also be noted that the blinding involves one constant for each polynomial, not for each coefficient. A further downside of this approach is that elements that are zero in the first place, i.e., most elements of the secret key, stay zero after the multiplication. The attack proposed in Section 5.5.4 only needs to distinguish between values being zero and non-zero. Thus, it will likely still work with the countermeasure in place.

6.1.2 Shuffling

In the unprotected implementation, the sparse index vector $\mathbf{c}$ directly determines the order in which the secret key elements are processed during the sparse multiplication. This information is utilised for the attack to map and evaluate the power leakage of the elements of the sparse multiplication to their respective secret key indices. The sparse multiplication represents an operation that allows for a countermeasure based on hiding and reducing the leakage by hiding in the time dimension as described in Section 4.6.1.

First, it is possible to randomise the order in which the output coefficients are computed; this can be done because no data dependency between the output coefficients exists. Due to $n = 512$ and only one core performing the multiplication, this would allow for 512! possible permutations of the output coefficients.

Furthermore, each multiplication consists of κ additions over the quotient ring $\mathbb{Z}_{2q}[x]/(x^n + 1)$, which is performed by additions over $\mathbb{Z}$ due to the involved secret key elements being small [15]. Such additions are of course commutative, i.e., the order of the coefficients does not influence the result. As an example, the computation of a single output coefficient $z_{1,i}$ under an arbitrary sparse vector $\mathbf{c}$ with a Hamming-weight of 4 can be written as:

$$z_{1,i} = s_7 + s_8 + s_9 + s_{10} = s_9 + s_8 + s_{10} + s_7 \tag{6.2}$$

Shuffling could, therefore, reorder the processing of the n output coefficients as well as the sequence of κ additions, resulting in $n! \cdot \kappa!$ possible combinations.

For a DPA attack, based on Mangard et al. [28], shuffling the operations does increase the number of needed traces significantly, especially given the size of $n = 512$. However, first-order DPA attacks might still be possible.

6.1.3 Masking

A third way of protecting the sparse multiplication involves splitting the secret key $\mathbf{s}_i$, which is defined over a quotient ring. The basic idea of masking is to split the secret key into separate shares and then perform the sparse multiplication with both shares separately. Finally, the shares are recombined to retrieve the correct value for the output coefficient.

The idea of sharing the polynomial multiplication based on its properties has been proposed similarly in Section 6.1.1 but involved a multiplicative sharing of the secret key $\mathbf{s}_i$ as well as the sparse vector $\mathbf{c}$. This approach has two downsides. First, the sharing of $\mathbf{c}$ requires the sparse multiplication to be performed using multiplications instead of additions. Second, elements being zero are zero after applying the multiplicative masking.

In contrast to the multiplicative masking, the idea is to perform an additive masking solving both problems mentioned before. The elements of the secret key $\mathbf{s}_i$ are defined over $\mathbb{Z}_{2q}$. However, sharing over $\mathbb{Z}_{2q}$ requires modular arithmetic and thus has higher resource requirements. Instead, due to the elements of the result of $\mathbf{s}_i\mathbf{c}$ being small, the sharing can be done over $\mathbb{Z}_k$ with $k = 2^d$. The variable d needs to be chosen such that any possible output coefficient resulting from the κ additions can be represented. The splitting of the secret key is then done by sampling a uniformly random vector $\mathbf{r}$ from the discrete uniform distribution $\mathcal{U}_k$.

$$\mathbf{r} \sim \mathcal{U}_k \tag{6.3}$$

This sampled vector $\mathbf{r}$ is then used as one share $\mathbf{s}'_i$ of the split secret key. The second part $\mathbf{s}''_i$ is then calculated by subtracting the first share from the secret key $\mathbf{s}_i$:

$$\begin{aligned} \mathbf{s}_i &= \mathbf{s}'_i + \mathbf{s}''_i \\ \mathbf{s}'_i &= \mathbf{r} \\ \mathbf{s}''_i &= \mathbf{s}_i - \mathbf{r} \end{aligned} \tag{6.4}$$

To compute the product $\mathbf{s}_i\mathbf{c}$, the unchanged sparse multiplication is performed separately on both shares $\mathbf{s}'_i$ and $\mathbf{s}''_i$ to get the result $\mathbf{v}'_i$ and $\mathbf{v}''_i$. To retrieve the result of the multiplication the sum of $\mathbf{v}'_i$ and $\mathbf{v}''_i$ is computed:

$$\begin{aligned} \mathbf{v}'_i &= \mathbf{s}'_i\mathbf{c} \\ \mathbf{v}''_i &= \mathbf{s}''_i\mathbf{c} \\ \mathbf{v}_i &= \mathbf{v}'_i + \mathbf{v}''_i \end{aligned} \tag{6.5}$$

This result can then be further used as part of the signature creation. Based on this modification, Line 6f. of Algorithm 3.2 transforms to:

$$\begin{aligned} \mathbf{z}_i &= \mathbf{y}_i + (-1)^b\mathbf{v}_i \\ &= \mathbf{y}_i + (-1)^b(\mathbf{v}'_i + \mathbf{v}''_i) \\ &= \mathbf{y}_i + (-1)^b(\mathbf{s}'_i\mathbf{c} + \mathbf{s}''_i\mathbf{c}) \\ &= \mathbf{y}_i + (-1)^b(\mathbf{s}'_i + \mathbf{s}''_i)\mathbf{c} \\ &= \mathbf{y}_i + (-1)^b\mathbf{s}_i\mathbf{c} \end{aligned} \tag{6.6}$$

This results in unmasking the shares directly after the sparse multiplication. However, each output coefficient depends on κ different secret key elements, making the DPA harder if not infeasible.

6.2 Masked Sparse Multiplication in Hardware

The existing hardware implementation is now extended by the masking scheme described above. First of all, a random number generator is required to sample the random vector $\mathbf{r}$. The design already utilises three instances of the Trivium stream cipher [11] as part of the sampling process in the Gaussian sampler [39]. The used Trivium implementation outputs a single bit in each clock cycle. While for a minimalistic design the existing instances of the samplers could be reused for the generation of the masks, for evaluation purposes additional instances were added. The number of supplementarily generated Trivium instances is configurable and allows for evaluation regarding size and speed.

The size of the mask was based on the `result` register. This register is used to save the output coefficients of the $\mathbf{s}_i\mathbf{c}$ multiplication. As $\delta_2 = 0$, the sum of the output coefficients is between $-(2\kappa+1)$ and $+(2\kappa+1)$, i.e., ± 47. Therefore the width of the `result` register and also the mask is fixed to $d = 7$ bits. Originally, due to the key-sizes of $\mathbf{s}_i$, the width of the memory of the secret key has only been 2 bits for $\mathbf{s}_1$ and 3 bits for $\mathbf{s}_2$. To enable the masking, the width of the secret key memory was increased to the same width of 7 bits. One mask polynomial is created for each of the secret keys $\mathbf{s}_1$ and $\mathbf{s}_2$. However, if more than one multiplication core is configured, the same mask is applied to the same key elements on different cores. This keeps the necessary entropy at a steady level. More cores lead to a wider datapath introducing more noise which increases the difficulty of the attack [28].

For efficiency reasons, the number of cores has essentially been doubled. In this configuration, two cores can be seen as a pair and are tasked with computing the same output coefficient based on each of their shares in parallel. Initially, the secret key storage of one multiplication core is initialised with the secret key, while the other is completely set to zero.

In addition to the module that generates the randomised masks, the finite state machine as seen in Figure 6.1 has been updated as well. The newly added states, marked in red, performs the masking of the secret key. To do so, the module initially sits in the `Idle` state. Upon receiving the signal to start, it transitions to the `Mask` state, where it resides until the Trivium instances generate the necessary 14 random bits needed to mask both $\mathbf{s}_1$ and $\mathbf{s}_2$. Depending on the number of Trivium instances configured this takes between 1 and 14 clock-cycles, due to each Trivium instance generating a single random bit per cycle.

Once the necessary number of random bits for the random vector $\mathbf{r}$ is generated the module switches to the `Mask_Update` state. At this point, the random bit vector is subtracted from the key share of one of the multiplication cores, while it is added to the other. This procedure represents the actual masking of the secret key, with the key being stored in the difference of the two shares. This process is reiterated for each of the n elements of the secret key $\mathbf{s}_i$, before transitioning to the `Mask_Done` state that starts the process of the sparse multiplication.

In Figure 6.1 the sparse multiplication is indicated by the three states in the grey box. Each output coefficient is computed by a series of transitions between the `Compute`, `Wait_Cycle` and `Output` states. This implements the operation to calculate $\mathbf{s}_i\mathbf{c}$ as described in Section 3.5.2. Each of the states might take more than one cycle. The `Compute` state

iterates over all κ indices of the vector $\mathbf{c}$, the `Wait_Cycle` waits for the cores to finish their operation and the `Output` state clocks out the result of the calculation one after the other.

However, the operation performed in the `Output` state has been modified slightly. Due to each result containing the product of the sparse multiplication based on only one of the shares, both parts are added on-the-fly revealing the correct outcome of the computation. After calculating the last coefficient, the module transitions to the `Finish` state before going back, waiting for a new input during the `Idle` state.

It should also be noted that for any consecutive iterations the BRAMs storing the secret key are not reset to their initial state, but instead, the newly generated masks are merely added to the existing content. Thus the masks are continuously updated ensuring that each secret key coefficient is only used κ-times with the same mask.

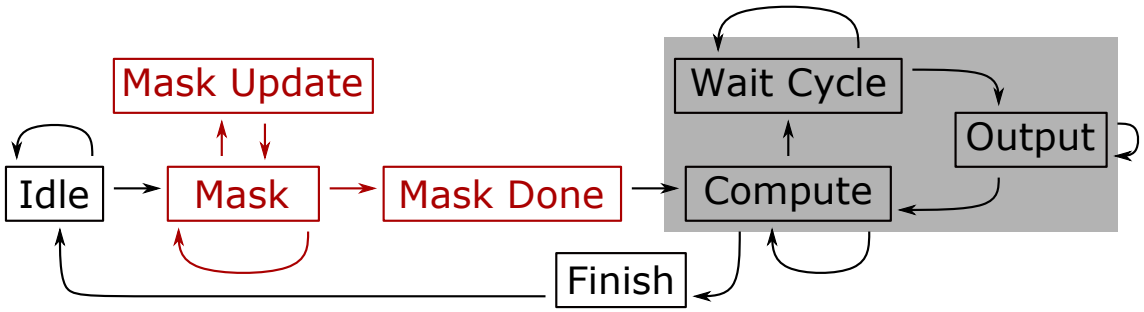


Figure 6.1: Finite state machine including the masking scheme as proposed in Section 6.1.3. The states in the grey area compute the n output coefficients.

6.2.1 Cost

With the structure and functionality of the masking side-channel protection discussed, this section goes into detail with the cost of the countermeasure. An overview is given in Table 6.1. The unprotected implementation is compared to the protected implementation, each with different configurations concerning the number of cores configured for the sparse multiplication. In case of the protected implementation comparisons are made regarding the number of additionally used Trivium instances, generating the random masks.

The unprotected implementations have been configured to 1, 4 and 8 cores. The single core variant represents the configuration with the lowest requirement in size but is also the one with the lowest throughput due to a low level of parallelism. On the other end of the spectrum is the implementation utilising 8 cores, which is the recommended configuration by Pöppelmann et al. [39] and is a compromise between throughput and demanded area on the FPGA. The implementation using four cores is situated in between.

For the protected implementation a minimalistic configuration is shown with one core for the multiplication and one Trivium instance. However, here the configuration using only one sparse core refers to the module fulfilling the functionality of one core. Due to the implemented masking scheme, a single output coefficient is internally computed by two cores in parallel. As an alternative to a protected single core implementation, a protected variant using four cores is listed. Both configurations are listed in their slow, but small form using only one Trivium instance and a very high-speed alternative utilising 14 Trivium instances.

A comparison in terms of required area for the implementation shows that the number of digital signal processors (DSP) and the number of Block RAMs ($BRAM$) do not change

at all. The number of lookup-tables (LUT) varies between 10,581 for the unprotected single core variant and 13,873 for the protected implementations with four cores. Based on Table 6.1 this translated to about 125-150 LUTs per instance. Similarly, a high number of Trivium instances increases the number of Flip-Flops (FF) noticeably as well with about 250 to 290 FFs per Trivium.

On the one hand, the overall size increases due to the generation of the mask but also due to the increase in the size of the datapath to 7 bits for each of the stored key elements.

Due to the rejection sampling, the signature generation is *not* constant time. Therefore, several metrics have been chosen to indicate the performance of the implementation. They were measured using 1000 signatures for each configuration. The number of cycles covers the moment the signing core received the start signal until it signals the end of the signing process and is reported by the implementation, therefore excluding the protocol overhead involved. Furthermore, the standard deviation has been calculated to give a better indication of the spread of the results. Also, the minimum and the maximum number of measured cycles is given. Given that 1000 signature have been analysed and based on the repetition rate of 1.6 as seen in Table 3.1, the minimum number of cycles most likely corresponds to signatures that were not rejected and mark the minimum time required.

A comparison shows that the protected implementation takes about 10% longer if enough randomness is instantly available during the masking of the keys. This overhead is due to the masking still requiring to iterate over all key elements and storing the masked values back to the memory. If only one Trivium instance is used, signatures take noticeably longer because of the additional wait times introduced during the sampling of the random mask.

	Unprotected			**Protected**			
# Sparse Cores	1	4	8	1	1	4	4
# Trivium	-	-	-	1	14	1	14
LUT	10,581	10,705	10,982	11,086	12,716	11,905	13,873
FF	10,220	10,322	10,517	10,571	13,876	10,875	14,644
DSP	8	8	8	8	8	8	8
BRAM	9,5	9,5	9,5	9,5	9,5	9,5	9,5
avg. cycles /sig	33250	14537	12749	46547	36404	27554	17525
σ	21372	9572	9763	29538	23072	17117	10941
min cycles	20046	8530	6610	28228	22098	16710	10578
max cycles	220624	101650	109049	338849	265554	200510	116356

Table 6.1: Size and performance comparison of implementations of the BLISS signature scheme with different number of multiplication cores and Trivium instances.

6.3 Attack on Masked Sparse Multiplication

In this section, the attack described in Section 5.5 that was executed successfully on the unprotected implementation is applied to the protected sparse multiplication. For this evaluation 10,000 power traces have been captured using the setup described in Section 5.4. As with the unprotected implementation, the BLISS signer has been configured only to perform the sparse multiplications based on the data generated in Matlab and to only use a single core for the sparse multiplication resulting in 512 multiplications in each trace. The computation involving the secret key $\mathbf{s}_2$ has been entirely disabled to further reduce the noise introduced by other components. However, due to the splitting of the secret key into two parts, two cores run at the same time. This setup is ideal for an adversary as it limits the noise from other components. Thus, it allows for a worst-case evaluation and a proper analysis of the countermeasure.

As with the unprotected implementation, the power traces were first preprocessed by making the segments mean free and by removing high-frequency noise and other parasitic effects by applying a filter resulting in traces as seen in Figure 6.2. Due to the design performing both multiplications in parallel, the amplitude of the resulting sinusoidal power traces is higher than for the unprotected implementation.

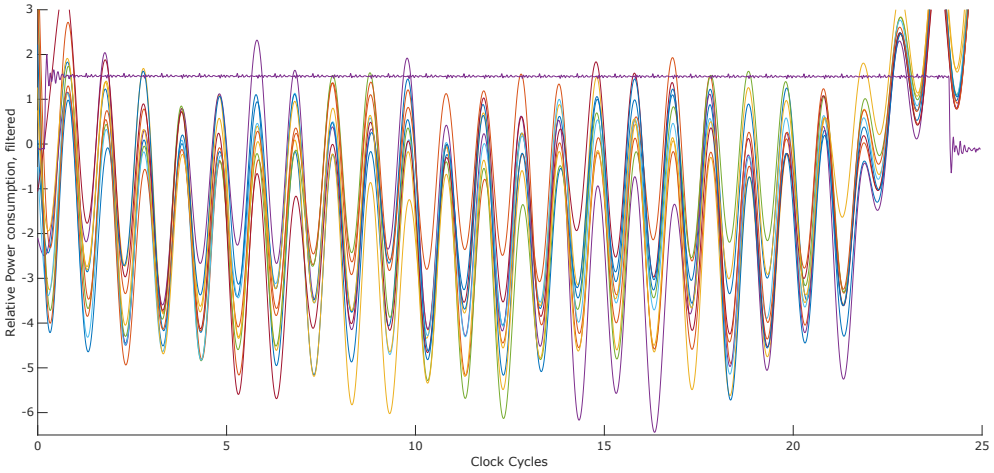


Figure 6.2: Filtered power traces of the protected implementation.

Next, as a first classification, the power traces are categorised based on the value of the key element involved in the first addition during the 256th multiplication. Moreover, the mean value of the traces in each category is computed. However, in contrast to the unprotected implementation, no noteworthy difference between the categories can be seen in Figure 6.3.

6.3.1 Differential Power Analysis

At first glance, the power traces do not show any significant leakages. However, the power model that has previously been shown to be successful in Section 5.5.3 is applied to these traces as well, because due to the masked and split secret key the precise value is unknown. Therefore, the correct and 99 similar keys, with same magnitude at the same locations but

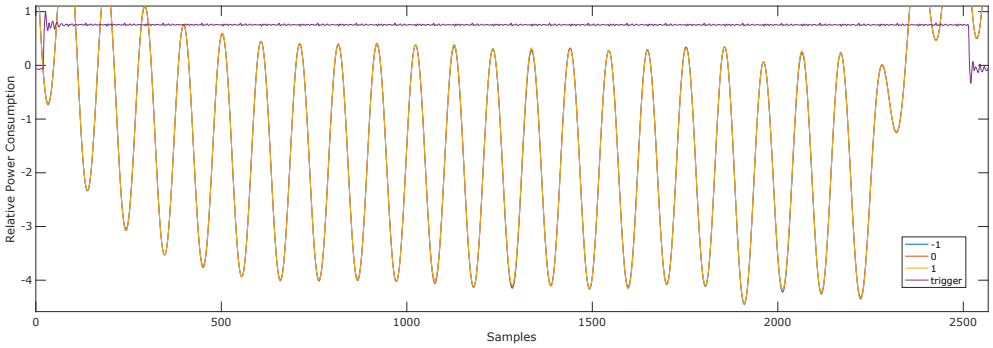
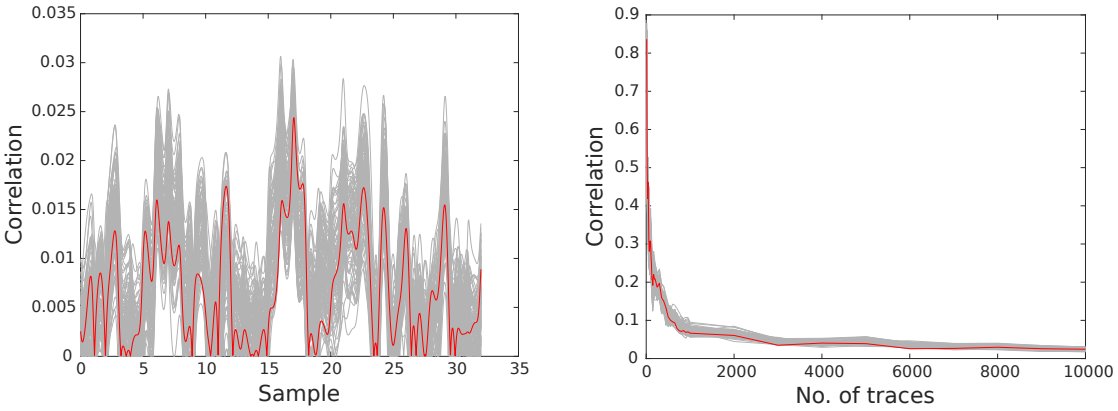


Figure 6.3: Mean value of the traces when grouped together based on the the value of the first involved secret key element. The purple line indicates the trigger signal marking the start and end of a single sparse multiplication.

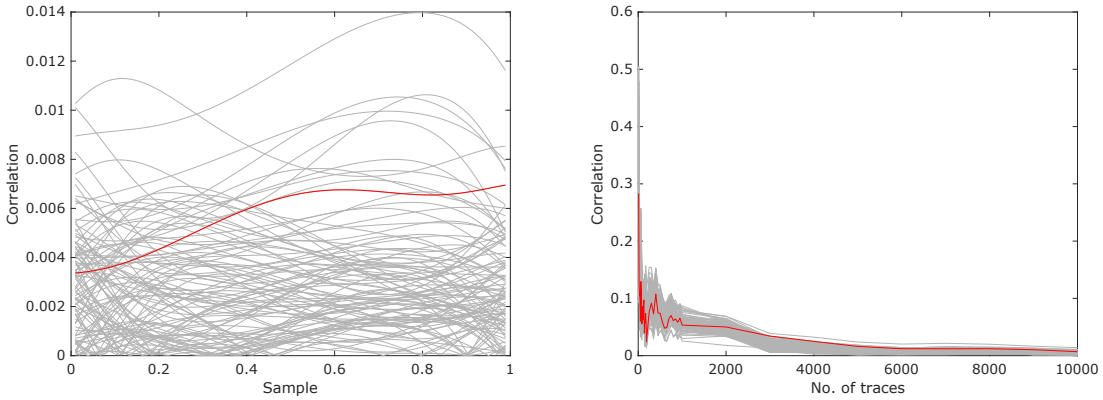
different signs are generated. The power model is then based on the 100 keys combined with the vector **c** for each trace. Each measurement point of the output coefficient with the same index is then correlated with the power model. The results are displayed in Figure 6.4. As can be seen, there is no significant correlation between the power traces and the power model. Additionally, the correlation with the power model based on the correct key, as marked in red, does not display a higher correlation than the fake keys, making the power traces indistinguishable as seen in Figure 6.4a. This is further illustrated in Figure 6.4b showing the highest correlation found in the computation involving an increasing number of traces. Again, the correlation of the power model based on the correct key does not show a higher correlation than the others even at 10,000 traces.



(a) Correlation along samples **(b)** Distinction of keys

Figure 6.4: Result of the DPA when applying the power model to the same multiplication block and the same addition with index 13 in each trace.

Moreover, the DPA is used to evaluate the use of a random addition of a random multiplication block for the set of available power traces as described in Section 5.5.3, with the same pre-processing applied to the traces. The result can be seen in Figure 6.5. It can be seen that the correct key does not separate itself from the fake keys and while the



(a) Correlation along samples **(b)** Distinction of keys

Figure 6.5: Result of the DPA when applying the power model to different multiplication blocks but different additions in each trace.

correlation is very low, the distribution along the time-axis differs vastly from the attack on the unprotected implementation. Furthermore, when comparing the correlation over an increased number of utilised traces, as displayed in Figure 6.5b, the correct key cannot separate itself from the fake-ones, making it indistinguishable from the other keys at up to 10,000 power traces. This represents a stark contrast to the unprotected implementation, which showed a clear partition starting at 800 traces using the same methodology.

6.3.2 Clustering-based Attack

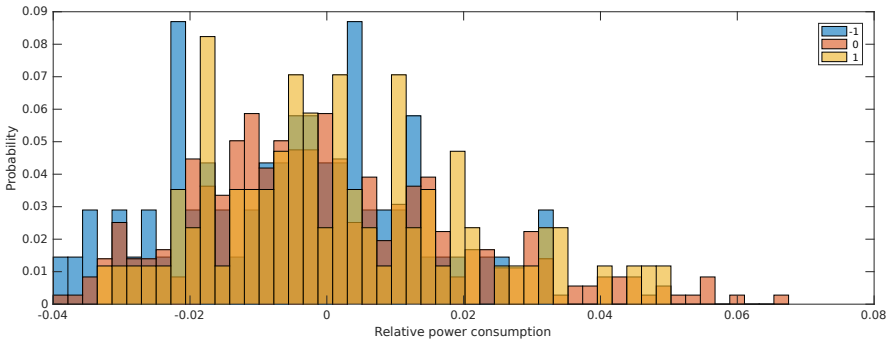


Figure 6.6: Distribution of the mean values of the traces with their colour demonstrating the respective key value in the secret key.

Using the obtained power traces a clustering-based attack is executed on the device, as described in Section 5.5.4. In case of the unprotected implementation, this reduced the number of required traces to a fraction of the required traces for the DPA. Therefore, using the same process, first, the traces are compressed based on the points with the highest correlation during the DPA. This results in a single value for each addition of the sparse

multiplication. The values are then grouped based on the index of the involved secret key element. This links the power consumption to the secret key index. Next, a mean value for each secret key index is calculated. The resulting mean values are shown in the histogram in Figure 6.6. The colour of each bin shows the actual value of the secret key element. While the unprotected implementation, as seen in Figure 5.14, showed clearly separated distributions for each of the three possible values, in case of the protected implementation this transformed to a single distribution around the zero-mark.

This distribution of the secret key values of the masked implementation cannot be used to distinguish the secret elements. Therefore, it is not possible to perform a clustering based on locating the zeros by selecting the indices with the highest value.

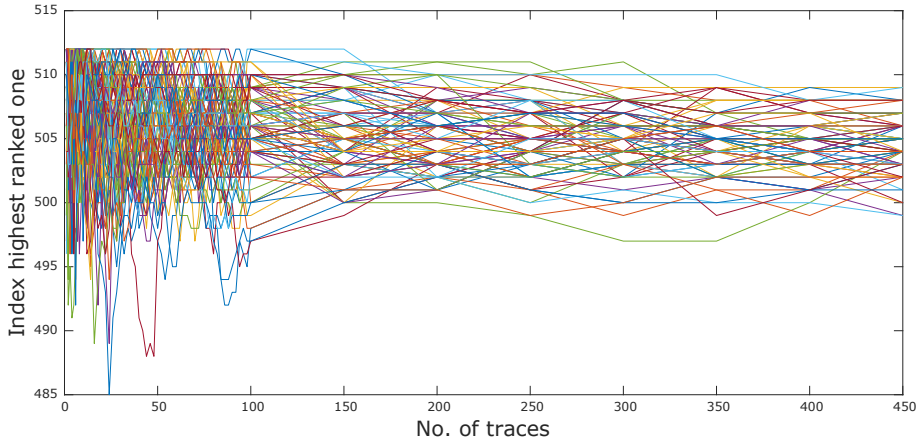


Figure 6.7: Index of the classified ones depending on the number of traces.

However, the attack on the unprotected implementation offered the possibility the further decrease the number of required traces by classifying a larger set of traces and then using this superset and the method by Pessl et al. [37] to determine which 154 elements of the set are ± 1. As seen in Figure 6.7 the accuracy of the classification stays at about the same level even for several hundred traces as compared to rapidly converging towards 154 as seen with the unprotected implementation. Additionally, it can be seen that the lowest value dips at about 485, with the majority of values being situated higher than 500 and therefore completely unsuited for this attack.

Chapter 7

Conclusions

This section summarises the findings of this thesis and its impact on schemes similar to BLISS. Furthermore, an outlook on possible future work is given, including ideas for further optimisations and additional countermeasures.

In this thesis, the hardware implementation of BLISS, a lattice-based signature algorithm, was analysed. The targeted implementation by Pöppelmann et al. [39] was designed for the Spartan 6 FPGA family. Besides discussing fundamental properties of lattice-based cryptography in general and detailing the BLISS signature algorithm, the principles of side-channel attacks were discussed. Furthermore, existing proposals for attacks and countermeasures of the BLISS signature scheme, or its related algorithm BLISS-B, were explained and analysed in terms of their feasibility when applied to a hardware implementation.

After an overview of available options, the sparse multiplication has been chosen as the point of attack as it directly incorporates the secret key used for the signature creation. Moreover, by using the sparse multiplication as the point of attack, the described method is not only specifically applicable to BLISS but for any scheme involving the multiplication of a secret key and some sparse vector. After isolating the sparse multiplication as it is used in the BLISS signature scheme, a series of power analysis attacks were performed on the unprotected implementation.

7.1 Results

A first inspection of the traces after pre-processing and sorting them by the key element used at a specified location showed a visible power leakage. By performing a differential power analysis, it could not only be shown that a power leakage exists for traces at the same point on the time axis, but also that the leakage is time-invariant. Therefore, the presented attack can extract more information from a single power trace and can recover the secret key more efficiently by combining the data. Using only means of a differential power analysis, it could be shown that the correlation of the correct key guess is visibly different from similar fake keys starting at about 800 traces, by only leveraging the same addition of the same output coefficient in all traces. By utilising a varying output coefficient but the same addition, this could be further reduced to about 400 traces. About the same result of 400 traces was observable by using the same output coefficient in each trace but varying the index of the addition. Making both components variable did not further improve the result but showed a visible difference starting at about 800 traces. Although a

power leakage is visible in all of the presented scenarios, a DPA cannot be used to recover the secret key, because all key coefficients are involved in the operation.

It was then shown that a clustering-based attack could be applied as well. After a trace compression based on the knowledge obtained during the differential power analysis, the trace values were grouped based on the corresponding index of the secret key. The mean for each index was calculated resulting in a single value for each index of the secret key. The values were plotted and showed a clear separation between three clusters of secret key values, giving a good indication of the feasibility of the attack. The clusters were then used to determine the secret key. The identification of the value and the correct sign required about 36 traces. With the goal to only separate the key into elements with a value of zero and a magnitude of 1, it could be shown that only about 10 traces are needed. However, this required additional processing using the method proposed by Pessl et al. [37]. Further tests were made under the premise that a superset of indices contains all elements of the key with a value of ± 1, but also a certain number of false positives. Using the method proposed by Pessl et al. [37] allowed for successful key recovery in a reasonable time if the set stays smaller than 180 elements. Moreover, this additional step and analysis showed that an attack on the unprotected implementation is possible with as little as four traces.

The success of the attack showed the need for an effective countermeasure. After comparison of countermeasures proposed in the literature, an approach based on splitting and additive masking of the secret key has been proposed. Further details on the extension of the hardware implementation by Pöppelmann et al. [39] were then discussed. By separating each secret key element into two shares, the link between the data and the power consumption was removed effectively. This was achieved by generating a random number and adding it to one share while subtracting it from the other, such that the actual secret key is only stored in the difference between the two shares. Furthermore, the cost of this countermeasure has been demonstrated showing that the countermeasure requires twice the number of cores to perform the computation in the about the same time. Furthermore, using a naive approach on the masking, the signature generation gets slowed down if not enough cached entropy is available. As a result, the unprotected four core implementation requires about 10700 LUTs and 10300 FFs, whereas the protected implementation requires about 11000 to 13900 LUTs and about 10600 to 14600 FFs, depending on the exact configuration used. In terms of performance, the average number of cycles increased from about 14500 cycles for the unprotected implementation with four cores to 17500 to 27500 cycles for the protected implementation depending on the available entropy for the masks.

The resistance to power side channel attacks has then be shown by performing the previously successful attack on the protected implementation as well. However, with the protected implementation neither the initial visual inspection nor the differential power analysis nor the clustering-based attack yielded any indication of the power attack being able to recover the secret key or parts of it with up to 10000 power traces. This holds true even though the setup poses ideal conditions for the attack due to the sparse multiplication running isolated from the rest of the chip.

7.2 Future Work

This section gives an outlook on several improvements that should be considered in the future to improve the current results

7.2.1 Shuffling

Shuffling as a countermeasure has been described in Section 6.1.2. Shuffling basically changes the order of the computed output coefficients and also the order of the additions to hide the computation in the time domain. While this alone does not protect against higher-order DPA attacks, it increases the difficulty and can also be used in conjunction with the masking scheme. Therefore, it would be of interest to test the effect of this countermeasure on both the unmasked as well as the masked hardware implementation and also determining the amount of additional hardware required. If used together with the masking countermeasure this could allow for the masking scheme with reduced entropy. Finally, depending on the effectiveness of the shuffling countermeasure this could lead to an overall smaller implementation.

7.2.2 Performance Improvements

As mentioned in Section 6.2, the mask is applied before each sparse multiplication, such that even the first computation is performed using the masked secret key. However, it would also be possible to apply the masks either before the signal to perform the multiplication is received (e.g., during the polynomial multiplication) or start the (re-)masking immediately after the computation of the output coefficients. This could reduce the time needed down to a point where the masking does not influence the runtime of the signature creation.

7.2.3 Reduction in Size

Section 6.2.1 compared a variety of implementations with and without the masking scheme in place. Each of the protected implementations showed a corner-case regarding the number of Trivium instances, with one opting for low-area and the other one for high-performance. The Trivium instances are continuously generating one bit of randomness per cycle and instance. By using an intermediate storage of appropriate size, it should be possible that during the masking there is no need to wait for more randomness to become available. This would further keep the performance at the same level while reducing the number of required Trivium instances and therefore saving area on the chip. When optimising for a low-area implementation, it should also be possible to get better results with larger intermediate storage for the obtained randomness to get a significant boost in performance with hardly any increase in area required.

7.2.4 Full Protection

The countermeasure described in this thesis only protects the sparse multiplication, which is just one part of the signature creation involving elements worthy of protection. Therefore, the other weak points including the rejection sampling and the Gaussian sampling should also be protected to obtain a fully side-channel protected implementation of BLISS. Furthermore, already published countermeasures like split shuffled sampling by Saarinen [47] should be evaluated for hardware implementations.

7.2.5 Attack on NTT

The attack on the Number Theoretic Transform by Primas et al. [43] has been shown successfully for microcontroller implementations of *Ring LWE* schemes. However, due to the attack being applicable for any scheme involving the NTT, the attack is in theory

also applicable to a hardware implementation of BLISS. Therefore, a further interesting task would be to apply the attack by Primas et al. [43] on the BLISS signature algorithm, especially on a hardware implementation. However, based on the higher parallelism of such an implementation it may be necessary to perform certain changes to the attack.

Bibliography

[1] M. Ajtai. "Generating Hard Instances of Lattice Problems (Extended Abstract)". In: *In Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*. ACM, 1996, pp. 99–108.

[2] Miklós Ajtai and Cynthia Dwork. "A Public-key Cryptosystem with Worst-case/Average-case Equivalence". In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 284–293. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258604. URL: http://doi.acm.org/10.1145/258533.258604.

[3] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-quantum Key Exchange—A New Hope". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 327–343. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim.

[4] Selçuk Baktir and Berk Sunar. "Achieving Efficient Polynomial Multiplication in Fermat Fields Using the Fast Fourier Transform". In: *Proceedings of the 44th Annual Southeast Regional Conference*. ACM-SE 44. Melbourne, Florida: ACM, 2006, pp. 549–554. ISBN: 1-59593-315-8. DOI: 10.1145/1185448.1185568. URL: https://doi.acm.org/10.1145/1185448.1185568.

[5] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks". In: *Proceedings of the IEEE* 94.2 (Feb. 2006), pp. 370–382. ISSN: 0018-9219. DOI: 10.1109/JPROC.2005.862424.

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

[7] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults". In: *Advances in Cryptology — EUROCRYPT '97*. Ed. by Walter Fumy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 37–51. ISBN: 978-3-540-69053-5.

[8] Google - Matt Braithwaite. *Experimenting with Post-Quantum Cryptography*. 2016. URL: https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[9] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. *Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme*. Cryptology ePrint Archive, Report 2016/300. https://eprint.iacr.org/2016/300. 2016.

[10] Johannes Buchmann, Alexander May, and Ulrich Vollmer. "Perspectives for Cryptographic Long-term Security". In: *Commun. ACM* 49.9 (Sept. 2006), pp. 50–55. ISSN: 0001-0782. DOI: 10.1145/1151030.1151055. URL: http://doi.acm.org/10.1145/1151030.1151055.

[11] Christophe De Cannière. "Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles". In: *Information Security*. Ed. by Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 171–186. ISBN: 978-3-540-38343-7.

[12] M. H. Devoret and R. J. Schoelkopf. "Superconducting Circuits for Quantum Information: An Outlook". In: *Science* 339.6124 (2013), pp. 1169–1174. ISSN: 0036-8075. DOI: 10.1126/science.1231930. eprint: http://science.sciencemag.org/content/339/6124/1169.full.pdf. URL: http://science.sciencemag.org/content/339/6124/1169.

[13] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. "A Practical Implementation of the Timing Attack". In: *Smart Card Research and Applications*. Ed. by Jean-Jacques Quisquater and Bruce Schneier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 167–182. ISBN: 978-3-540-44534-0.

[14] Léo Ducas. *Accelerating Bliss: the geometry of ternary polynomials*. Cryptology ePrint Archive, Report 2014/874. https://eprint.iacr.org/2014/874. 2014.

[15] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. *Lattice Signatures and Bimodal Gaussians*. Cryptology ePrint Archive, Report 2013/383. https://eprint.iacr.org/2013/383. 2013.

[16] Léo Ducas and Tancrède Lepoint. *A Proof-on-Concept Implementation of BLISS*. http://bliss.di.ens.fr/.

[17] Thomas Espitau, Pierre-Alain Fouque, Benoit Gerard, and Mehdi Tibouchi. *Side-Channel Attacks on BLISS Lattice-Based Signatures – Exploiting Branch Tracing Against strongSwan and Electromagnetic Emanations in Microcontrollers*. Cryptology ePrint Archive, Report 2017/505. https://eprint.iacr.org/2017/505. 2017.

[18] N. J. Al Fardan and K. G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy(SP)*. Vol. 00. May 2013, pp. 526–540. DOI: 10.1109/SP.2013.42. URL: doi.ieeecomputersociety.org/10.1109/SP.2013.42.

[19] Nick Howgrave-Graham and Mike Szydlo. "A Method to Solve Cyclotomic Norm Equations". In: *Algorithmic Number Theory*. Ed. by Duncan Buell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 272–279. ISBN: 978-3-540-24847-7.

[20] Michael Hutter and Jörn-Marc Schmidt. "The Temperature Side-Channel and Heating Fault Attacks". In: vol. 8419. Nov. 2013.

[21] Anatolii Karatsuba and Yu Ofman. "Multiplication of Multidigit Numbers on Automata". In: 7 (Dec. 1962), p. 595.

[22] Neal Koblitz. "Elliptic Curve Cryptosystems". In: *Mathematics of Computation* 48.177 (Jan. 1987), pp. 203–209. ISSN: 0025-5718.

[23] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[24] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. London, UK, UK: Springer-Verlag, 1999, pp. 388–397. ISBN: 3-540-66347-9. URL: https://dl.acm.org/citation.cfm?id=646764.703989.

[25] Mun-Kyu Lee, Jeong Eun Song, Dooho Choi, and Dong-Guk Han. "Countermeasures against Power Analysis Attacks for the NTRU Public Key Cryptosystem". In: 93-A (Jan. 2010), pp. 153–163.

[26] Z. Li, N. S. Dattani, X. Chen, X. Liu, H. Wang, R. Tanburn, H. Chen, X. Peng, and J. Du. "High-fidelity adiabatic quantum computation using the intrinsic Hamiltonian of a spin system: Application to the experimental factorization of 291311". In: *ArXiv e-prints* (June 2017). arXiv: 1706.08061 [quant-ph].

[27] Vadim Lyubashevsky and Daniele Micciancio. "Generalized Compact Knapsacks Are Collision Resistant". In: *Automata, Languages and Programming*. Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 144–155. ISBN: 978-3-540-35908-1.

[28] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 0387308571.

[29] Daniele Micciancio. "Generalized Compact Knapsacks, Cyclic Lattices, and Efficient One-Way Functions". In: *computational complexity* 16.4 (Dec. 2007), pp. 365–411. ISSN: 1420-8954. DOI: 10.1007/s00037-007-0234-9. URL: https://doi.org/10.1007/s00037-007-0234-9.

[30] Daniele Micciancio and Oded Regev. *Lattice-based Cryptography*. https://cims.nyu.edu/~regev/papers/pqc.pdf. 2008.

[31] Daniele Micciancio and Oded Regev. "Lattice-based Cryptography". In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 147–191. ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_5. URL: https://doi.org/10.1007/978-3-540-88702-7_5.

[32] Victor S. Miller. "Use of Elliptic Curves in Cryptography". In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Ed. by Hugh C. Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426. ISBN: 978-3-540-39799-1.

[33] Michele Mosca. *Cybersecurity in an era with quantum computers: will we be ready?* Cryptology ePrint Archive, Report 2015/1075. https://eprint.iacr.org/2015/1075. 2015.

[34] Judea Pearl. "Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach". In: *Proceedings of the Second AAAI Conference on Artificial Intelligence*. AAAI'82. Pittsburgh, Pennsylvania: AAAI Press, 1982, pp. 133–136. URL: https://dl.acm.org/citation.cfm?id=2876686.2876719.

[35] Chris Peikert. "A Decade of Lattice Cryptography". In: *Found. Trends Theor. Comput. Sci.* 10.4 (Mar. 2016), pp. 283–424. ISSN: 1551-305X. DOI: 10.1561/0400000074. URL: https://dx.doi.org/10.1561/0400000074.

[36] Peter Pessl. *Analyzing the Shuffling Side-Channel Countermeasure for Lattice-Based Signatures*. Cryptology ePrint Archive, Report 2017/033. https://eprint.iacr.org/2017/033. 2017.

[37] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. *To BLISS-B or not to be - Attacking strongSwan's Implementation of Post-Quantum Signatures*. Cryptology ePrint Archive, Report 2017/490. https://eprint.iacr.org/2017/490. 2017.

[38] Peter Pessl and Stefan Mangard. "Enhancing Side-Channel Analysis of Binary-Field Multiplication with Bit Reliability". In: *Proceedings of the RSA Conference on Topics in Cryptology - CT-RSA 2016 - Volume 9610*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 255–270. ISBN: 978-3-319-29484-1. DOI: 10.1007/978-3-319-29485-8_15. URL: https://dx.doi.org/10.1007/978-3-319-29485-8_15.

[39] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. *Enhanced Lattice-Based Signatures on Reconfigurable Hardware*. Cryptology ePrint Archive, Report 2014/254. https://eprint.iacr.org/2014/254. 2014.

[40] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. *FPGA Implementation of the BLISS Signature Scheme*. https://sha.rub.de/research/projects/lattice (Last retrieved: August 2017). 2014.

[41] Thomas Pöppelmann and Tim Güneysu. "Towards Efficient Arithmetic for Lattice-based Cryptography on Reconfigurable Hardware". In: *Proceedings of the 2Nd International Conference on Cryptology and Information Security in Latin America*. LATINCRYPT'12. Santiago, Chile: Springer-Verlag, 2012, pp. 139–158. ISBN: 978-3-642-33480-1. DOI: 10.1007/978-3-642-33481-8_8. URL: https://dx.doi.org/10.1007/978-3-642-33481-8_8.

[42] Thomas Pöppelmann and Tim Güneysu. "Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware". In: *Revised Selected Papers on Selected Areas in Cryptography – SAC 2013 - Volume 8282*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 68–85. ISBN: 978-3-662-43413-0. DOI: 10.1007/978-3-662-43414-7_4. URL: https://www.ei.rub.de/media/sh/veroeffentlichungen/2013/08/14/lwe_encrypt.pdf.

[43] Robert Primas, Peter Pessl, and Stefan Mangard. *Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption*. Cryptology ePrint Archive, Report 2017/594. https://eprint.iacr.org/2017/594. 2017.

[44] Steven Rich and Barton Gellman. *NSA seeks to build quantum computer that could crack most types of encryption*. 2014. URL: https://www.washingtonpost.com/world/national-security/nsa-seeks-to-build-quantum-computer-that-could-crack-most-types-of-encryption/2014/01/02/8fff297e-7195-11e3-8def-a33011492df2_story.html.

[45] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: http://doi.acm.org/10.1145/359340.359342.

[46] Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. *Compact and Side Channel Secure Discrete Gaussian Sampling*. Cryptology ePrint Archive, Report 2014/591. https://eprint.iacr.org/2014/591. 2014.

[47] Markku-Juhani O. Saarinen. *Arithmetic coding and blinding countermeasures for lattice signatures*. Cryptology ePrint Archive, Report 2016/276. https://eprint.iacr.org/2016/276. 2016.

[48] David Samyde, Sergei Skorobogatov, Ross Anderson, and Jean-Jacques Quisquater. "On a New Way to Read Data from Memory". In: *Proceedings of the First International IEEE Security in Storage Workshop*. SISW '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 65–. ISBN: 0-7695-1888-5. URL: https://dl.acm.org/citation.cfm?id=829507.830220.

[49] Google - Emily Schechter. *Communicating the Dangers of Non-Secure HTTP*. 2018. URL: https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html.

[50] Google - Emily Schechter. *Say "yes" to HTTPS: Chrome secures the web, one site at a time*. 2017. URL: https://www.blog.google/topics/safety-security/say-yes-https-chrome-secures-web-one-site-time/.

[51] Bruce Schneier. *NSA Plans for a Post-Quantum World*. 2015. URL: https://www.schneier.com/blog/archives/2015/08/nsa_plans_for_a.html.

[52] P. W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. SFCS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 124–134. ISBN: 0-8186-6580-7. DOI: 10.1109/SFCS.1994.365700. URL: https://doi.org/10.1109/SFCS.1994.365700.

[53] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM J. Comput.* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 0097-5397. DOI: 10.1137/S0097539795293172. URL: http://dx.doi.org/10.1137/S0097539795293172.

[54] Sergei P. Skorobogatov and Ross J. Anderson. "Optical Fault Induction Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–12. ISBN: 978-3-540-36400-9.

[55] NIST - National Institute of Standards and Technology. *Post-Quantum Cryptography, Call for Proposals*. 2017. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/call-for-proposals.

[56] Ltd TROCHE Co. *SAKURA-G FPGA board*. http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G.html. 2014.

[57] Ltd TROCHE Co. *SAKURA-G, Side-channel AttacK User Reference Architecture Specification*. Specification. MORITA TECH CO., LTD., 2013.

[58] Mozilla - Tanvi Vyasa and Peter Dolanjski. *Communicating the Dangers of Non-Secure HTTP*. 2017. URL: https://blog.mozilla.org/security/2017/01/20/communicating-the-dangers-of-non-secure-http/.

[59] M. Wan, Z. He, S. Han, K. Dai, and X. Zou. "An Invasive-Attack-Resistant PUF Based On Switched-Capacitor Circuit". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 62.8 (Aug. 2015), pp. 2024–2034. ISSN: 1549-8328. DOI: 10.1109/TCSI.2015.2440739.

[60] Xuexin Zheng, An Wang, and Wei Wei. "First-order Collision Attack on Protected NTRU Cryptosystem". In: *Microprocess. Microsyst.* 37.6-7 (Aug. 2013), pp. 601–609. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2013.04.008. URL: https://dx.doi.org/10.1016/j.micpro.2013.04.008.