Physical Attack Vulnerability of Hash-Based Signature Schemes

Verwundbarkeit von Hash-basierten Signaturverfahren durch Physikalische Angriffe
Master-Thesis von Matthias Julius Kannwischer
Tag der Einreichung:

1. Gutachten: Prof. Dr. Dr. h.c. Johannes A. Buchmann
2. Gutachten: Dr. Juliane Krämer
3. Gutachten: Dr. Denis Butin

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Computer Science
Cryptography and Computer Algebra

Physical Attack Vulnerability of Hash-Based Signature Schemes
Verwundbarkeit von Hash-basierten Signaturverfahren durch Physikalische Angriffe

Vorgelegte Master-Thesis von Matthias Julius Kannwischer

1. Gutachten: Prof. Dr. Dr. h.c. Johannes A. Buchmann
2. Gutachten: Dr. Juliane Krämer
3. Gutachten: Dr. Denis Butin

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den September 18, 2017

(M. Kannwischer)

Abstract

The eXtended Merkle signature scheme (XMSS), one of the most promising post-quantum digital signature schemes, is currently being standardized by the Internet Engineering Task Force (IETF). Once fully standardized, XMSS is expected to be implemented in a wide variety of applications to replace existing digital signature schemes like RSA and DSA which are vulnerable to quantum computer attacks. Secure implementations need to be resistant to physical attacks, i.e., fault attacks and side-channel attacks. This thesis provides an extensive analysis of the physical attack vulnerability of XMSS.

We confirm the general conjecture that hash-based signature schemes, including XMSS, inherently provide a very strong resistance against passive side-channel attacks. Timing attacks are impossible due to the constant runtime of all building blocks processing secret data. The only component that may be vulnerable to power analysis attacks is the pseudorandom number generator (PRNG) which is used to generate the Winternitz one-time signature (W-OTS)+ secret keys within XMSS. Since this component is not standardized by the XMSS Internet Draft, an implementer may choose a PRNG susceptible to power analysis attacks. This thesis proposes, implements, and simulates a differential power analysis (DPA) on a SHA2 PRNG which allows the recovery of a secret intermediate value. This enables an adversary to compute all W-OTS+ secret keys, consequently allowing universal XMSS forgeries. We show that the attack, while, in theory, also applicable to the PRNG recommended by the XMSS Internet Draft, is not relevant for practical parameters of XMSS. This emphasizes that the choice of the PRNG is essential for side-channel resistance of XMSS.

While XMSS provides strong resistance against passive side-channel attacks, we show that it is vulnerable to fault attacks when used in its hypertree variant $XMSS^{MT}$. This thesis adapts a very recent fault attack, which was initially proposed for the related hash-based signature scheme SPHINCS. The vulnerability is induced by the recomputation of W-OTS+ signatures for XMSS tree roots. We implement and simulate the attack upon XMSS and show that caching these signatures, which should be implemented for performance optimization anyway, entirely prevents the fault attack.

Contents

1 Introduction 1
1.1 Problem Statement 2
1.2 Contributions and Thesis Structure 3

2 Physical Attacks on Cryptographic Schemes 4
2.1 Power Analysis Attacks 5
2.1.1 Simple Power Analysis (SPA) 6
2.1.2 Differential Power Analysis (DPA) 7
2.2 Timing Attacks 8
2.3 Fault Attacks 10

3 Hash-Based Signature Schemes 12
3.1 One-Time Signature Schemes 12
3.1.1 Lamport-Diffie One-Time Signatures (LD-OTS) 12
3.1.2 Winternitz One-Time Signatures (W-OTS) 14
3.1.3 W-OTS+ 17
3.2 Construction of Many-Time Signature Schemes 18
3.2.1 Merkle Signature Scheme (MSS) 18
3.2.2 The eXtended Merkle Signature Scheme (XMSS) 20
3.2.3 Multi-Tree XMSS ($XMSS^{MT}$) 21
3.3 Related Work 22
3.3.1 McGrew Internet Draft and Leighton-Micali Signatures (LMS) 22
3.3.2 SPHINCS 23

4 Side-Channel Analysis of XMSS 26
4.1 Related Work 26
4.2 Assumptions 27
4.3 Timing Side-Channels 27
4.3.1 W-OTS+ 27
4.3.2 XMSS 28
4.3.3 Discussion 28
4.4 Power-related Side-Channels 29
4.4.1 W-OTS+ 29
4.4.2 XMSS 31
4.4.3 Discussion 32
4.5 Generalization to Other Hash-Based Schemes 32
4.5.1 LD-OTS, W-OTS and MSS 32
4.5.2 LMS 33
4.5.3 SPHINCS 33
4.6 Pseudorandom Number Generator (PRNG) and Hash Function Side-Channel Resistance 33
4.6.1 Hash Function Side-Channel Resistance 33
4.6.2 PRNG Side-Channel Resistance 35

5 Power Analysis Attack on the PRNG in XMSS 36
5.1 A DPA Attack on SHA2 HMAC . . . 36
5.2 Attack Design and Adversary Model . . . 38
5.3 Implementation . . . 40
5.3.1 Power Simulation . . . 40
5.3.2 DPA . . . 42
5.4 Results . . . 43
5.4.1 8-Bit Hamming Weight Leakage Model . . . 43
5.4.2 32-Bit Hamming Weight Leakage Model . . . 45
5.5 Applicability to Practical PRNGs . . . 48

6 Fault Attack on XMSS^{MT} 50
6.1 Fault Attack on SPHINCS . . . 50
6.2 Attack Design and Adversary Model . . . 51
6.3 Implementation . . . 54
6.4 Results . . . 56
6.5 Countermeasures . . . 58

7 Discussion 59
7.1 Summary . . . 59
7.2 Conclusion . . . 60
7.3 Recommendations for Implementers . . . 61
7.4 Future Work . . . 61

List of Figures

2.1 Power analysis attack setup 5
2.2 Power trace showing the 16 rounds of Data Encryption Standard (DES) [KJJ99] 6
2.3 Timing attack setup 8
2.4 Fault attack setup using the chosen-message model 10

3.1 Illustration of the LD-OTS secret key, public key, and signature 13
3.2 Illustration of the W-OTS secret key, public key, and signature 15
3.3 Illustration of the W-OTS+ secret key, public key, and signature 17
3.4 Construction of a Merkle tree 19

4.1 Parts of W-OTS relevant for side-channel analysis 28
4.2 Parts of XMSS relevant for side-channel analysis 29

5.1 DPA on SHA-256 HMAC (simplified from [BBD+13]) 36
5.2 Simulation of a DPA attack on a SHA2-based PRNG 40
5.3 Simulated power trace for $PRNG_{vuln}$ in the HW model for 32-bit words 41
5.4 Correlation values of correct and wrong key hypothesis over time 44
5.5 Maximum correlation of all possible key hypothesis 44
5.6 Success rate of the full DPA key recovery attack on the vulnerable PRNG in the 8-bit HW leakage model 45
5.7 Maximum partial correlation values of correct hypothesis 46
5.8 Success rate of recovering a single 32-bit value using a DPA on modular addition in the 32-bit HW leakage model 47
5.9 Success rate of recovering a single 32-bit value using a DPA on bitwise AND in the 32-bit HW leakage model 47

6.1 Proposed fault attack on XMSS^{MT} 51
6.2 Forging an XMSS^{MT} signature 53
6.3 Results of fault attack simulation: effect of n 56
6.4 Experimental results for the fault attack simulation for $n = 512$ and $p = 1$ 57
6.5 Results of fault attack simulation: effect of p 58

List of Tables

4.1 Guessing entropy for Hamming weight (HW)-leakage per byte 30

Listings

5.1 DPA attack on modular addition 42

List of Algorithms

5.1 SHA-256 compression function f [Nat15a] 37

6.1 Proposed fault attack on XMSS^{MT} 52
6.2 Fault attack: `extract_and_merge` 54
6.3 Fault attack: `forge_signature` 55

Acronyms

AES	Advanced Encryption Standard.
ASIC	application-specifc integrated circuit.
CPU	central processing unit.
DES	Data Encryption Standard.
DLP	discrete logarithm problem.
DPA	differential power analysis.
DRAM	dynamic random-access memory.
DSA	digital signature algorithm.
ECC	elliptic curve cryptography.
ECDSA	elliptic curve digital signature algorithm.
EU-CMA	existential unforgeability under adaptive chosen message attack.
FPGA	field programmable gate array.
GHz	gigahertz.
HD	Hamming distance.
HMAC	hash-based message authentication code.
HW	Hamming weight.
IETF	Internet Engineering Task Force.
IV	initialization vector.
KiB	kibibyte.
LD-OTS	Lamport-Diffie one-time signature.
LM-OTS	Leighton Micali one-time signature.
LMS	Leighton Micali signature.
LUT	look-up table.
MAC	message authentication code.
MHz	megahertz.
MiB	mebibyte.
MSS	Merkle signature scheme.
NIST	National Institute of Standards and Technology.

OTS one-time signature.

PRF pseudorandom function.
PRNG pseudorandom number generator.

SPA simple power analysis.

TiB tebibyte.

W-OTS Winternitz one-time signature.

XMSS eXtended Merkle signature scheme.

1 Introduction

The majority of the currently deployed cryptographic public key schemes are at risk of becoming insecure once large scale quantum computers become practical. This is due to Shor's algorithm [Sho97] which enables the factoring of large integers and the computation of discrete logarithms in polynomial time. Since these are the two fundamental mathematical problems upon which our current public key cryptography including RSA, digital signature algorithm (DSA), and ElGamal is built, it will become insecure as soon as quantum computers exist. Also, the elliptic curve cryptography (ECC) version of the discrete logarithm problem (DLP) which are, e.g., used in the elliptic curve digital signature algorithm (ECDSA) and EC-ElGamal will be broken then.

Current research in modern cryptography is trying to develop schemes that resist quantum attacks and, thus, are secure in the long term. This area of research is called *post-quantum cryptography* or *quantum-resistant cryptography* and denotes cryptographic schemes that can resist both, attacks carried out on classical computers and quantum computers. Note that these cryptographic schemes can be implemented and used on classical computers, such that a quantum computer is not required.

Although it is not certain if a large scale quantum computer will be available in the near future, recent advances in quantum computer research [IBM17, BIS+16, Rey17] both, at large corporations and universities, alert the cryptographic community. A replacement of current schemes is imperative and time-critical. The National Institute of Standards and Technology (NIST) announced in December 2016, that it will move to post-quantum cryptography [Nat16b] and calls for proposals of suitable cryptographic schemes with a deadline of November 2017.

Even though there are attempts to future proof the existing schemes, the resulting schemes currently are infeasible in practice. For example, Bernstein et al. proposed a post-quantum version of RSA called pqRSA [BHLV17] which can provide a post quantum security of approximately 100 bits, i.e., 2^{100} quantum operations are required to decrypt a given ciphertext without the knowledge of the corresponding secret key. However, to achieve this level of security a 1 tebibyte (TiB) secret key is required, which is infeasible for most applications. Additionally, the creation of such a key alone requires several days of computation on current hardware, while the encryption and decryption last several hours. Thus, while Bernstein et al. proved that a scheme based upon the hardness of factoring large integers can be built to resist quantum attacks, these schemes are currently of little practical value.

Most other post-quantum research over the last decades focused on proposing schemes that are built upon different mathematical problems, for which currently no quantum attacks are known, i.e., they are believed to be post-quantum secure. There are currently five categories of post-quantum problems upon which several schemes have been proposed; these are code-based cryptography [McE78], lattice-based cryptography [GGH97, HPS98], multivariate cryptography [MI88, KPG99], hash-based cryptography [Mer90, DSS05, BDS09, BDH11], and the relatively young field of isogeny-based cryptography [JF11].

One promising hash-based scheme is XMSS, proposed by Buchmann et al. [BDH11], which is currently being standardized by the IETF [HBGM17] and is expected to become an Internet standard soon. XMSS is an improved version of the Merkle signature scheme (MSS), which dates back nearly 40 years to the thesis of Merkle [Mer79]. XMSS comes with several security proofs that provide strong evidence that it is post-quantum secure under minimal security assumptions when treated as a black-box.

1.1 Problem Statement

Once XMSS has been fully standardized, the next step is the efficient implementation of the scheme in software and hardware. However, physical attacks, also known as implementation attacks, have been used to attack and break various implementations of cryptographic schemes that were believed to be secure in a mathematical sense. Thus, implementers of cryptographic schemes today try to prevent or mitigate such attacks upfront. To the best of the author's knowledge, there is no extensive analysis of physical attack vulnerability of XMSS, which is required to create such resistant implementations. This thesis aims to provide such an analysis. Physical attacks include side-channel attacks and fault attacks, which are both covered by this thesis.

The key point of side-channel attacks is that, while the inputs and outputs (i.e., the primary channel) of a cryptographic operation (e.g., signature generation) are not enough to break the scheme, some additional information leaked via a different channel (e.g., power consumption) allows an adversary to attack it (e.g., recover the key or forge a signature). In this thesis we consider timing and power side-channels, since the majority of attacks are based upon them.

In the context of side-channel attacks, the terms *side-channel resistant* and *side-channel resilience* can be easily mixed up, despite having completely different meanings: An implementation of a cryptographic scheme is considered *side-channel resistant* if the leaked information is not sufficient for a successful attack. The resistance is specific to the actual implementation and cannot be proven for a scheme in general. On the other hand, a cryptographic scheme is called *λ-leakage resilient (or side-channel resilient)*, if the scheme remains secure, even if at most λ bits of the secret key are leaked. The resilience is independent of the implementation, which allows for exact mathematical proofs. However, it remains crucial to ensure that the leakage bound is not exceeded by an implementation. This thesis focuses on the *side-channel resistance* of XMSS.

Fault attacks, which are the second category of physical attacks covered in this thesis, have also been used to break the security of various cryptographic schemes in the past [Ott04, BMM00, BOS06]. A *fault*, which can be either natural or malicious, is a misbehavior of a device that causes the computation to deviate from its specification. For example, this can be the flipping of a bit in a certain memory cell. In a fault attack, an adversary actively injects malicious faults into a cryptographic device, such that it outputs faulty data. This faulty output, which is potentially combined with several other faulty and valid outputs, is then used to reconstruct parts of the secret key or any other secret value.

Overall this thesis tries to answer the central research question:
How can XMSS be implemented in practice to resist physical attacks?

1.2 Contributions and Thesis Structure

The previous section presented the general motivation and problem statement of this thesis. The rest of the thesis is structured as follows: Chapter 2 provides the necessary background on physical attacks with a focus on the techniques applicable to hash-based cryptography. In Chapter 3, a brief overview of hash-based cryptography is given with emphasis on the work from which XMSS and, thus, the Internet Draft was derived. Chapter 4 to 7 constitute the main contributions of this thesis which are as follows:

- **Side-channel analysis of XMSS**: The extensive analysis of the XMSS side-channel resistance in Chapter 4 confirms the conjecture, that XMSS inherently provides a good protection against side-channel attacks, if the building blocks, namely the used hash functions and PRNG, are side-channel resistant. The results are coherent with related work on other hash-based signature schemes. We find that hash functions can be assumed to provide strong side-channel resistance, while the PRNG can be vulnerable in certain scenarios.

- **A differential power analysis (DPA) attack on PRNG**: Since an implementer of XMSS can choose which PRNG to use, he may choose a PRNG susceptible to power analysis attacks. We propose, implement, and evaluate a DPA on a SHA2-based PRNG in Chapter 5. We simulate our attack and show that the attack can be used to fully recover all W-OTS+ secret keys used for XMSS, which trivially allows an adversary to forge XMSS signatures.e show that the attack, while, in theory, also applicable to the PRNG recommended by the XMSS Internet Draft, is not relevant for practical parameters of XMSS. This emphasizes that the choice of the PRNG is essential for side-channel resistance of XMSS.

- **Fault attack on XMSS^{MT}**: We propose a modified version of a very recent fault attack on SPHINCS in Chapter 6, which can be used to break the multi-tree variant of XMSS. The attack is implemented and simulated and shows that a very small error, occurring during a very long computation period, leads to the leakage of a partial W-OTS+ secret key. This allows an adversary to create an existential W-OTS+ forgery, which in consequence allows him to create universal XMSS^{MT} forgeries.

- **Countermeasures against fault attacks**: We show in Section 6.5 that the use of existing XMSS performance optimizations mitigates the fault vulnerability presented. These optimizations should be implemented to properly protected against powerful adversaries, even if they are declared optional by the XMSS Internet Draft.

- **Generalization to other advanced hash-based signatures**: The comparison of XMSS to two other recent practical hash-based signature schemes, namely SPHINCS and the Leighton Micali signature (LMS) scheme, shows that they provide similar side-channel resistance and the same attacks apply to them.

2 Physical Attacks on Cryptographic Schemes

Physical attacks, also known as implementation attacks, have been used to attack and break various implementations of cryptographic schemes that are believed to be secure in a mathematical sense [Koc96, KJJ99, MOP07]. These attacks usually target different categories of *cryptographic devices*. A *cryptographic device* is a device which implements cryptographic operations and stores secret keys, i.e., it can be anything from a highly specialized implementation with a single purpose (e.g., a smart card) to a multi-purpose PC with a full-fledged operating system [MOP07].

The key point of side-channel attacks is that, while the inputs and outputs (i.e., the primary channel) of the cryptographic device are not enough to break the scheme, some additional information leaked via a different channel (e.g., power consumption) allows an adversary to attack it (e.g., recover the key). In this thesis we consider timing [Koc96, RMB15] and power side-channels [KJJ99, MOP07] since the majority of attacks are based upon them. However, other side-channels have also been used to successfully attack cryptographic schemes including electromagnetic [QS01, GMO01], photonic [KNSS13, CSW17] and acoustic emissions [GST14]. These side-channels are explicitly out-of-scope of this thesis and may form the basis for future work.

Another category of physical attacks considered are fault attacks [Ott04, BMM00], in which an adversary actively tampers with the cryptographic device, such that it behaves abnormally and outputs faulty data. This may allow an adversary to deduce parts of the secret key. In the worst case, the cryptographic device outputs the secret key itself. Forcing a device to behave abnormally can be achieved using a wide variety of techniques, but the most prominent ones are introducing power spikes and clock glitches [Ott04]. The tampering is often possible without the permanent destruction of the cryptographic device, such that no evidence is left behind.

While classical cryptography often excludes such "powerful" physical attacks, a lot of research has shown that the assumptions being made in such attacks about the adversaries' power are practical [HMP10, BB03, FLRV09]. Indeed, physical attacks present a severe threat to real world cryptography.

To describe the attacks presented and used in this thesis and emphasize the required adversary capabilities, the following classification of attacks as adapted from [MOP07] is helpful:

- The literature distinguishes *invasive* and *non-invasive* physical attacks. *Non-invasive* attacks do not damage the cryptographic device, e.g., the adversary is not allowed to de-package the chip, solder additional probes to it or permanently damage a hardware component. *Non-invasive* attacks can often stay undetected by the owner of the cryptographic device, while *invasive* attacks often tamper with the device such that it cannot be returned to the owner without notice of the attack. Both side-channel and fault attacks can be either *invasive* or *non-invasive*, but for most published attacks a *non-invasive* model suffices.

- Physical attacks can be *active* or *passive*. In a *passive* attack, the adversary uses the cryptographic device in the intended manner while collecting information leaked via some primary and side-channel. In contrast, an *active* adversary is allowed to inject faults into the cryptographic device that cause abnormal behavior. We consider side-channel attacks to be purely passive and fault attacks to be active. Some related work [Ott04] refers to fault attacks as active side-channel attacks and, thus, using side-channel attacks in a broader sense. However, we stick to the terminology used in [Krä15] to emphasize the fundamental difference of passive side-channel and fault attacks.

- Attacks can be carried out *locally* or *remotely*. When an attack is *local*, the adversary needs to be in (temporary) possession of the cryptographic device or at least in close proximity. While the majority of physical attacks are limited to *local* adversaries, some attacks can be carried out *remotely* (e.g., timing attacks on RSA [BB03] or ECDSA [BT11]).

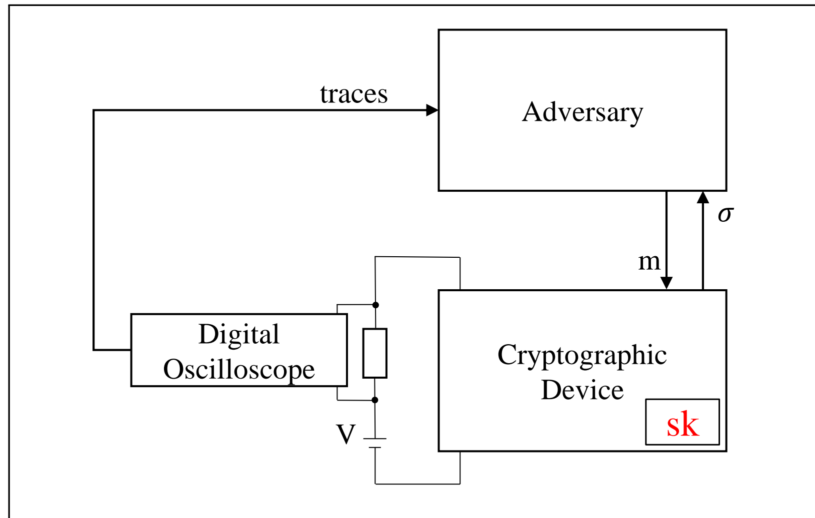


Figure 2.1: Power analysis attack setup

- Since digital signatures are considered in this thesis, we use the common distinction between *known message attacks* and *(adaptively) chosen message attacks*. Both fault attacks and side-channel attacks can use both models.

The following subsections present the considered attacks in this thesis in a very brief fashion. Section 2.1 covers power analysis attacks including simple power analysis (SPA) and differential power analysis (DPA). Section 2.2 introduces timing attacks starting with the first practical timing attack proposed by Kocher in 1996 [Koc96] and briefly sketching more advanced timing attacks. Fault attacks are covered in Section 2.3.

2.1 Power Analysis Attacks

Power analysis attacks mainly target relatively simple cryptographic devices that implement cryptographic primitives (e.g., smart cards, specialized microprocessors or Field programmable gate array (FPGA)). By carefully monitoring the power consumption of the cryptographic device during the computation of the primitives, usually collecting thousands of power traces, the adversary tries to recover some secret information (e.g., the secret key). If the devices were more complex, e.g., running a full operating system, the traces would contain too much noise for successful recovery. [MOP07]

Figure 2.1 illustrates a possible setup of a power analysis attack, although others are possible. The adversary is in (temporary) possession of the cryptographic device and is allowed to create signatures for a limited number of messages of his choice, i.e., a chosen-message attack. During the computations he collects power traces using a digital oscilloscope with a high sampling rate of several hundred megahertz (MHz) to a few gigahertz (GHz). To do so, the adversary inserts a small resistor (e.g., 1Ω) into the power supply and measures the voltage drop at the resistor which is proportional to the power consumption of the device. The adversary then uses the traces together with the known message-signature pairs to recover some secret data that allows him to forge a signature. This might be the entire secret key or some intermediate value.

Attacks like these are possible, because the power consumption of a cryptographic device depends on the data processed and the instructions executed. This is modeled using a *leakage function*. A *leakage function* maps the internal data to a value observable by an adversary. When using the most advanced (but purely theoretical) adversary model, the attacker is allowed to choose this function on his own and is sometimes also allowed to change the function in-between executions of the cryptographic primitive. However, this is a highly unrealistic model [SPY+10]. In practice, the *leakage function* mainly depends

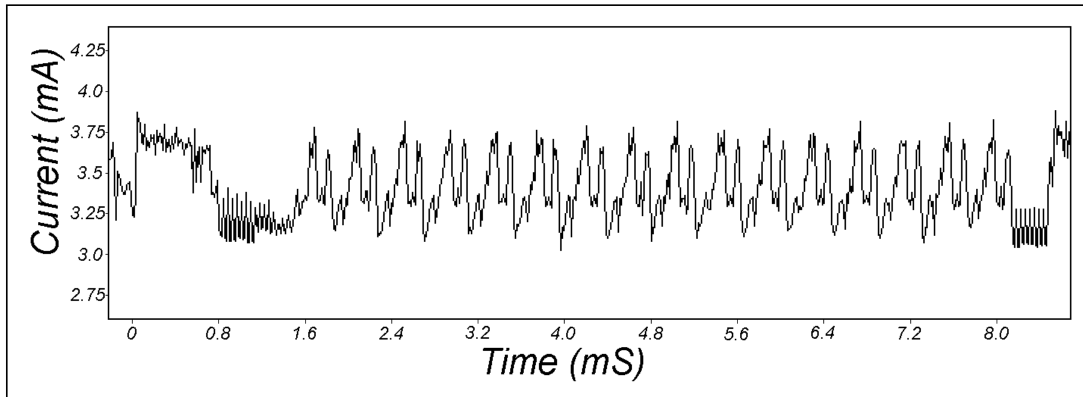


Figure 2.2: Power trace showing the 16 rounds of DES [KJJ99]

on the hardware used. Two models used for practical attacks are the Hamming weight (HW)-model and the Hamming distance (HD)-model. In the HW-model, the leakage of the Hamming weight, i.e., the number of 1's of each intermediate value, is assumed. The HW-model mainly applies to implementations on microprocessors due to their internal architecture. When using the HD-model, the device is assumed to leak the Hamming distance ($HD(x, y) = HW(x \oplus y)$) of two consecutive values in a certain register. The HD model works best for FPGA and application-specifc integrated circuit (ASIC) implementations [MOP07].

Although not in the focus of this thesis, mitigating those attacks is an important field of research. Two common countermeasures that are often implemented to prevent power analysis attacks are hiding and masking. Hiding tries to remove the data dependency of the power consumption of a cryptographic device. This can either be achieved in software, e.g., by randomizing the execution of the algorithm, or in hardware, by changing the cryptographic device, such that each operation requires approximately the same amount of energy or a random amount of energy. The data dependency can usually not be removed entirely, but reducing it makes attacks a lot harder. Masking uses randomization of the intermediate values to produce random power consumption. This needs to be integrated into the cryptographic primitive and is, thus, a software countermeasure. [MOP07]

The following sections introduce the two main types of power analysis attacks that have been successfully used to break many cryptographic schemes in the past.

2.1.1 Simple Power Analysis (SPA)

In SPA attacks the adversary directly analyzes and interprets the acquired power. SPA attacks use either a single or a very limited number of traces to recover the key or gain some additional knowledge about the implementation of the cryptographic scheme. The simplest form of this is a visual analysis of the plot of a power trace. Since the power consumptions depends on the instructions executed, it is possible to identify different parts of the algorithm. This can be very helpful for reverse engineering an implementation for which actual code or even the cryptographic scheme is unknown to an adversary. Figure 2.2 shows the power trace of an entire Data Encryption Standard (DES) encryption. It is possible to distinguish the 16 encryption rounds, which might enable an experienced adversary to detect that this device is indeed executing DES.

By zooming into the power trace, it is possible to distinguish different instructions. Each instruction has a characteristic power trace and if the adversary is able to record several samples per clock cycle, it would be possible to identify the actual executed instructions. If the executed code were known, which is often the case, an adversary would be able to recover the execution path for an execution. If the execution path depends upon the secret key, these leaks of information might be enough to allow the adversary to recover the key. In an extreme case, the implementation has conditional branches that

depend on a single bit of the secret key, which can then, in consequence, be easily recovered. SPA has been used to successfully attack several implementations of asymmetric cryptographic schemes, where the execution path either leaked the entire secret key or determined enough key bits that the remaining search space can be iterated exhaustively by an adversary [MOP07].

2.1.2 Differential Power Analysis (DPA)

A more advanced family of power analysis attacks are DPA attacks which were first proposed by Kocher in 1999 [KJJ99]. They exploit the data dependency of the power consumption, i.e., the property that the power consumption is dependent upon the processed data. While this is also the case for SPA, DPA attacks use traces of many computations of the cryptographic primitive for different input values, to find correlations within the traces, hence their name. Given enough traces, DPA attacks are able to find even the tiniest correlations, no matter how much noise is included in the traces. Additionally, DPA attacks do not require detailed knowledge about the cryptographic device to enable key recovery. It is often enough to know the cryptographic scheme that is executed [KJJR11]. The idea is best illustrated using a straightforward example: Assume that a cryptographic device that implements a symmetric cipher is attacked using a chosen-message attack. It is known that for each encryption the same key k is used and at some point of the algorithm a byte-wise XOR of key and message is computed ($k_i \oplus m_i$), where k_i and m_i denote the i-th byte each. This is the case for several widely used ciphers, e.g., Advanced Encryption Standard (AES) [Nat01], DES [Nat99], and Camellia [MMN04]. If the device implementing this operation is unprotected and an HW leakage model is assumed, the attack is simple: The device is queried with 8 different plain texts for each message byte $m_i = 2^j, \quad 0 \le j < 8$. Thus, the HW of $k_i \oplus m_{i,j}$ is either 0 or 1, which directly corresponds to the value of the j-th bit of k_i. However, in practice several factors prevent that only such few traces suffice to mount an attack. Firstly, there is noise included in the traces collected, which is both caused by the measurement setup and the physical properties of the attacked device. Secondly, the adversary does not know exactly which sample of the power trace corresponds to the computation he wants to attack. Therefore, the attack needs to be generalized, which was done by Mangard et al. [MOP07]. They describe 5 steps of a DPA attack:

- **Step 1**: The adversary picks an intermediate value that is computed somewhere in the algorithm. It needs to be a function $f(d, k)$ of some known variable value d (e.g., the message digest to be signed) and a part of the secret key k. A DPA requires the calculation of hypothetical power consumption values for each possible key candidate. Therefore, the adversary cannot attack the entire key, but only a smaller sub-key. It is important that the size of the sub-key space is small enough, such that it is possible to iterate over it. Usually, this is done for each key byte separately. Let K denote the size of the sub-key space, e.g., $K = 256$.

- **Step 2**: The adversary executes the cryptographic scheme and collects D (usually several thousand) traces of length T for different (e.g., random) input values $\mathbf{d} = (d_1, ..., d_D)$. Thus, given the traces $\mathbf{t_i} = (t_{i,1}, ..., t_{i,T}), \quad 1 \le i \le D$, this results in a matrix $\mathbf{T}$ of size $D \times T$.

- **Step 3**: For each input value d_i and each key candidate k_i (e.g., $0, 1, ..., 255$), the adversary calculates the hypothetical intermediate result corresponding to the chosen function f:

  $$v_{i,j} = f(d_i, k_j), \quad 1 \le i \le D \quad 1 \le j \le K$$

  This results in the matrix $\mathbf{V}$ of dimension $D \times K$.

- **Step 4**: The calculated hypothetical intermediate values are mapped to hypothetical power consumption values. This is usually the HW of the value, i.e.,

  $$h_{i,j} = HW(v_{i,j}), \quad 1 \le i \le D \quad 1 \le j \le K$$

  This yields the matrix $\mathbf{H}$, also of dimension $D \times K$.

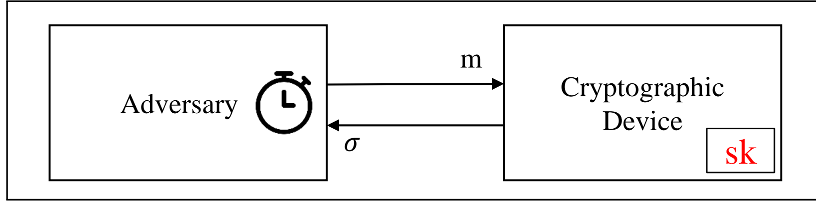


Figure 2.3: Timing attack setup

- **Step 5**: The adversary wants to find out which hypothetical power consumption values correlate the most with the collected traces. There are several statistical measures that can be used to achieve this, one of which is the Pearson correlation coefficient [MOP07]:

$$r_{i,j} = \frac{\sum_{d=0}^{D}\left(h_{d,i} - \overline{\mathbf{h_i}}\right) \cdot \left(t_{d,j} - \overline{\mathbf{t_j}}\right)}{\sqrt{\sum_{d=0}^{D}\left(h_{d,i} - \overline{\mathbf{h_i}}\right)^2 \cdot \sum_{d=0}^{D}\left(t_{d,j} - \overline{\mathbf{t_j}}\right)^2}}, \qquad 1 \leq i \leq K, \quad 1 \leq j \leq T$$

  Intuitively, the correlation coefficient evaluates for each sample in a trace $\mathbf{t_i}$ how much it correlates with the hypothetical power consumption value. Calculating $r_{i,j}$ for each column i of $\mathbf{H}$ and each column j of $\mathbf{T}$ yields a $K \times T$ matrix. The maximum value $r_{i,j}$ in this matrix corresponds to the correct key k_i.

After the first sub-key is successfully recovered, the adversary repeats the same steps to recover the other parts of the key using the same set of traces and potentially using the result of the previous DPA.

2.2 Timing Attacks

The second side-channel this thesis considers are timing channels. When exploiting a timing side-channel, an adversary measures the time required to complete a cryptographic operation. If the execution time depends upon the secret key, this leaks information about it. In some cases this attack can be used to extract the entire secret key and, thus, break the scheme under attack [Koc96, Ber05]. As all physical attacks, timing attacks attack the actual implementation of the scheme. Thus, a scheme cannot be proved to be timing side-channel resistant, since the implementation can still introduce additional side-channels.

This thesis distinguishes between platform-independent timing attacks that exploit conditional branches depending on secret data and advanced timing attacks that exploit more sophisticated features of modern processors like cache hierarchies or branch predictors.

Other than power analysis attacks, timing attacks are not limited to cryptographic devices, but can also be used to attack general purpose computers locally or even remotely [BB03, BT11].

To illustrate the idea of timing attacks, the original attack on RSA, DSA and other schemes which involve modular exponentiation proposed in a paper by Kocher in 1996 [Koc96] is sketched: Suppose RSA is used to sign a message m using the secret key d. To create the signature s the modular exponentiation $s = m^d \mod n$ is computed, which can be done using the square-and-multiply algorithm. The algorithm iterates over the bits of the exponent d, and either performs a squaring (if $d_i = 0$) or a squaring followed by a multiplication (if $d_1 = 1$). This obviously has an impact on the runtime of the modular exponentiation. The attack works as follows: Let $T = e + \sum_{i=0}^{w-1} t_i$ denote the overall runtime of a signature generation. t_i is the time required for iteration i of the loop, whereas e includes everything else like loop overhead and measurement error. The basis of the attack is, that a modular multiplication is fast for some values and very slow for other values (depending if a modular reduction step is required or not). The time of multiple signature generations for different messages m is measured. The bits of d are then recovered iteratively:

- Assume all bits d_c for $c < b$ (initially $b = 0$) are already known, where d_0 is the most significant bit of d.
- The adversary guesses d_b.
- Since all $d_c, c < b$ are known tot he adversary, he can estimate $\sum_{i=0}^{b} t_i$ for each message, since he knows which multiplications will be slow depending on the message and the known exponent bits.
- Given the total measured time T, the adversary calculates $T' = T - \sum_{i=0}^{b-1} t_i = e + \sum_{i=b}^{w-1} t_i$.
- If the key guess d_b is correct, the variance of T' is expected to be $Var(e) + (w - b)Var(t)$.
- If the key guess d_b is incorrect, the variance of T' is expected higher, since the estimate for the b-th iteration will be inaccurate.
- By picking the key guess with the lower variance, the adversary recovers bit d_b.

It is important to note, that for all iterations the same measurements can be used.

Although this first attack only works for a straightforward implementation of the standard RSA signature scheme, the same paper also proposes an attack on an implementation using the Montgomery multiplication which is used for faster modular multiplication without the need for costly modular reductions after each step.

Since the initial proposal of timing attacks, a lot of other attacks based on timing side-channels have been published. While most of them are due to conditional branches and are relatively easy to mitigate, another, more sophisticated category has been found recently that exploits the architectural features of modern Central processing unit (CPU). A recent book by Rebeiro et al. [RMB15] gives the current state-of-the-art of such advanced timing attacks. While providing an extensive overview over this area is far beyond the scope of this thesis, two two types of attacks need to be pointed out that show that timing channels can be very subtle and easily overseen.

These attacks target general purpose PCs instead of special cryptographic devices and assume that the adversary exactly knows the hardware used for the computations. Since they also require precise timing, it usually requires some kind of malware on the attacked host as well. Although there are a lot more prerequisites for a successful attack, the assumptions don't seem too unrealistic.

The first family of advanced attacks are cache attacks [Ber05]. Cache attacks exploit that all modern CPU use cache hierarchies, usually L1 to L3. If a cache line already resides in the L1-cache, a load instruction will execute much faster than if the cache line needs to be fetched from L2, L3 or even dynamic random-access memory (DRAM). This feature, which is essential for performance of modern applications, can be exploited to mount an attack on cryptographic schemes. For example, efficient implementations of AES use Look-up table (LUT) to implement the `SubBytes` step during encryption and decryption. In this step each byte of the state is mapped to its multiplicative inverse in the Galois field $GF(2^8)$. By using a LUT containing all 256 precomputed substitutions, a complex inversion can be replaced by a simple memory lookup. However, since the LUT occupies multiple cache lines and the value of the state byte is used as an index, an adversary can obtain information about which state bytes lead to colliding accesses, i.e., accesses to the same cache line. Since the each state byte is computed as $k_i \oplus m_i$ before the `SubBytes` step, these collisions leak information about the key k_i. Similar to Kocher's Timing attack, the adversary combines the timing measurements of many encryption runs with different plaintexts m to recover the full key.

The second family of timing side-channels that emphasizes that timing channels can be very hard to detect and mitigate are branch prediction attacks. All modern CPU are pipelining the execution of instructions, i.e., working on multiple instruction at once to achieve better performance. While this works very well for independent instructions, it leads to problems when instructions depend on each other. For example, if a branch instruction is faced, the processor does not know if the branch will be taken or not until the instruction reaches the end of the pipeline. Since waiting for the result will cause a

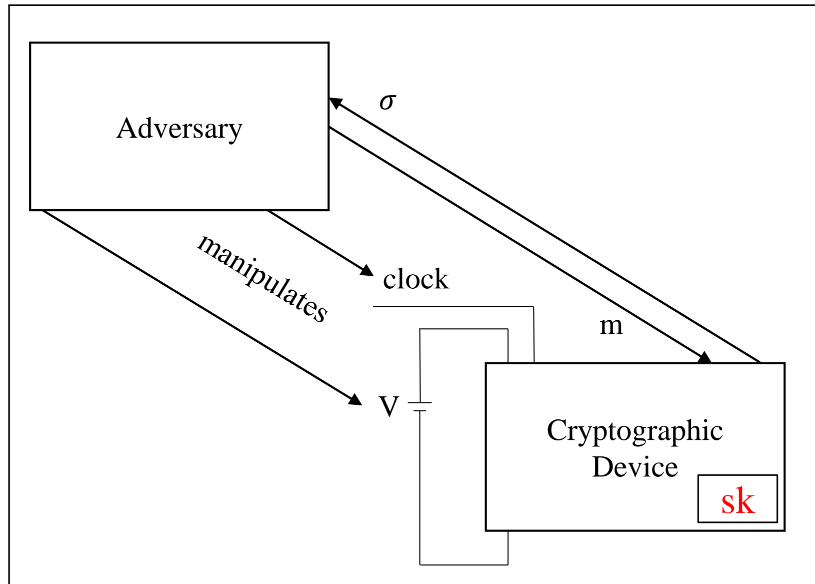


Figure 2.4: Fault attack setup using the chosen-message model

high delay, the processors tries to guess if a branch will be taken or not and continues to fetch instructions correspondingly. This is called branch prediction. Once the actual branch condition is evaluated, it is clear if the branch predictor guessed correctly or not. In case of a wrong guess, the pipeline needs to be stalled and the processor starts fetching the instructions from the correct code location. Since this stalling has a huge performance impact, this can be used for a timing attack.

If an adversary knows or can guess the method used for the branch prediction and can execute own code before executing the cryptographic primitive, he can manipulate the state of the branch prediction unit such that he controls whether it will predict a branch or no branch during the execution of the cryptographic scheme. If the branch is dependent on secret data, this can leak valuable information. Although this attack see it also works if both branches would take the same time. Mitigating this is, thus, far more difficult.

2.3 Fault Attacks

A *fault*, which can be either natural or malicious, is a misbehavior of a device that causes the computation to deviate from its specification. For example, this can be the flipping of a bit in a certain memory cell.

In a fault attack, an adversary actively injects malicious faults into a cryptographic device, such that it outputs faulty data. This invalid output, which is potentially combined with several other faulty and valid outputs, is then used to reconstruct parts of the secret key or any other secret value. Since this is an active attack, we do not consider this as a side-channel attack, although related work does.

Research during the last two decades found that many widely used schemes can be broken by fault attacks [BDL01, BMM00, BOS06] with the successful first attack dating back to 1997 [BDL97]. To illustrate the idea of fault attacks, the original attack by Boneh et al. [BDL97] on the RSA signature scheme is sketched: To sign a message m, the signer needs to compute $s = m^d \mod n$. Suppose this modular exponentiation is done using the Chinese remainder theorem for performance optimization. The signature can then be generated by computing $s = as_1 + bs_2 \mod n$ with $s_1 = m^d \mod p$ and $s_2 = m^d \mod q$ and the precomputed values for a and b, such that $a \equiv 1 \mod p$, $a \equiv 0 \mod q$, $b \equiv 0 \mod p$, $b \equiv 1 \mod q$.

An adversary creates two signatures of the same message m and injects a fault in the computation of s_1 for the second signature. Thus, he obtains one correct signatures s and one faulty signature $\hat{s}$ for message m. Since the fault only affects $\hat{s}_1$, the adversary knows that $s_2 = \hat{s}_2$ and $s_1 \neq \hat{s}_1$. Thus, $s - \hat{s} = a(s_1 - \hat{s}_1)$. Since $a \equiv 1 \mod p$ and $a \equiv 0 \mod q$, the adversary knows that $gcd(a(s_1 - \hat{s}_1), n) = q$, because a must

be divisible by q and cannot equal 0. This allows to efficiently factor n and, thus, compute the private key d. If the fault occurs during the computation of s_2, the formula can be easily adapted. This attack, although very simple, shows that an unprotected straightforward implementation of a mathematically secure scheme can be easily broken by an adversary capable of injecting faults.

These attacks are primarily relevant for cryptographic devices like smart cards or cryptographic co-processors, because an adversary can determine precisely when the computation is happening either because he possesses the specification (e.g., code or netlist) or by reverse engineering it. If attacking more complex devices like PCs, the injection of faults will most likely result in unpredictable behavior or the crash of the operating system.

Faults can be induced in various ways, but the most prominent ones are exposing the device with high voltage or manipulating the clock frequency, such that they are outside of the tolerance of the cryptographic devices. This is illustrated in Figure 2.4. Since smart cards require an external power supply and clock signal, they present an optimal target for such an attack and mounting it is relatively straightforward. Other techniques to induce faults are cosmic, $\alpha-$, $\beta-$ and X-rays, heat, light, electric fields and focused ion beams [Ott04]. Some of which are either very hard to tune finely or just too expensive.

When analyzing the fault vulnerability of an implementation, it is important to take into account how precisely the adversary can control the injected faults in terms of fault location, timing, number of bits affected, fault type, success probability, and duration. The more precisely the fault injection can be tuned, the more powerful attacks are possible.

Otto describes 4 different fault types [Ott04], each of which is defined on an arbitrary set of bits stored in memory:

- *Stuck-at fault*: The bits are fixed to either 0 or 1. If the implementation tries to overwrite them later, they stay unchanged.
- *Bit flip fault*: The affected bits are flipped.
- *Random fault*: All bits are set to a random value. This is the most realistic fault type.
- *Bit set or reset fault*: The adversary chooses which bits are set and which are reset. They can be changed later by the implementation.

To mitigate fault attacks, there are two typical categories of countermeasures: There are hardware countermeasures, that change the hardware to prevent or detect the injection of faults and there are software countermeasures, i.e., the development of new algorithms and schemes that are immune against fault attacks by either introducing fault detection or completely preventing the vulnerability. Due to the wide variety of ways to inject faults, it is difficult to protect an implementation from each and every attack vector available.

3 Hash-Based Signature Schemes

One of the most promising classes of cryptographic schemes that are believed to be quantum-resistant are hash-based signatures. A cryptographic hash function is a function of the form $h : \{0,1\}^* \rightarrow \{0,1\}^n$, i.e., it maps data of arbitrary length to a *hash digest* of fixed length. Usually it is required to have the following properties [Buc02]:

- *Preimage resistance*: Given a hash digest d it is infeasible to find $x \in \{0,1\}^*$, such that $d = h(x)$.
- *Second-preimage resistance*: Given a $x \in \{0,1\}^*$, it is infeasible to find a $x' \in \{0,1\}^*, x' \neq x$ that has the same hash as x, i.e., $h(x) = h(x')$.
- *Collision resistance*: It is infeasible to find $x, x' \in \{0,1\}^*, x \neq x'$, such that they have the same hash digest, i.e., $h(x) = h(x')$. *Collision resistance* implies *second-preimage resistance*, but is not required for all applications of cryptographic hash functions.

If not stated differently, the hash functions in this thesis are assumed to have all three properties.

This chapter provides an overview of current state-of-the-art hash-based cryptography. Section 3.1 introduces one-time signature (OTS) that form the basis of today's hash-based signatures. The introduction of the original Lamport-Diffie one-time signature (LD-OTS) [Lam79] is followed by the improved scheme called W-OTS which is also a part of the Internet Draft of Hülsing et al. [HBGM17].
Section 3.2 continues with the explanation of the MSS [Mer79, Mer90] and its improved version XMSS [BDH11] which uses OTS to construct many-time signatures. The idea of using multiple layers of XMSS-trees, which is named XMSS^MT (multi-tree) is also introduced.
Section 3.3 emphasizes the differences of the considered Internet Draft by Hülsing et al. [HBGM17] to related work. Firstly, we compare it to another Internet Draft by McGrew at al. [MCF17] Secondly, we compare it to SPHINCS [BHH+15] - an extension of XMSS which removes the statefulness using few-time signatures instead of OTS.

3.1 One-Time Signature Schemes

OTS form the central building block for all current hash-based signature schemes. As the name suggests, each secret key must only be used once. If a key is used more than once, the security of the scheme may be compromised. OTS date back to 1979, when Lamport and Diffie published a report on how to construct a signature scheme using one-way functions [Lam79]. However, Lamport-Diffie one-time signature (LD-OTS) never became practical due to their huge key and signature sizes. In 2005 Dods et al. [DSS05] extended the idea of Lamport and Diffie and introduced W-OTS, which uses much smaller keys and signatures than LD-OTS, while still providing a similar security level. This scheme was further elaborated by Buchmann et al. in 2009 [BDS09]. A further extended version of W-OTS, which is called W-OTS+, was introduced by Hülsing in 2013 [Hül13] and is included in the current version of the Internet Draft [HBGM17]. LD-OTS, W-OTS and W-OTS+ are introduced in chronological order.

3.1.1 Lamport-Diffie One-Time Signatures (LD-OTS)

LD-OTS were first introduced in 1979 [Lam79] and form the basis of the hash-based signature schemes available today [Buc16]. To explain the more modern schemes, it is reasonable to look at the original scheme first and understand why the security of the scheme solely relies on the one-wayness of the used function and why it is essential that each key is only used once. To define a signature scheme, it is sufficient to define the three algorithms for key generation, signature generation and signature verification.

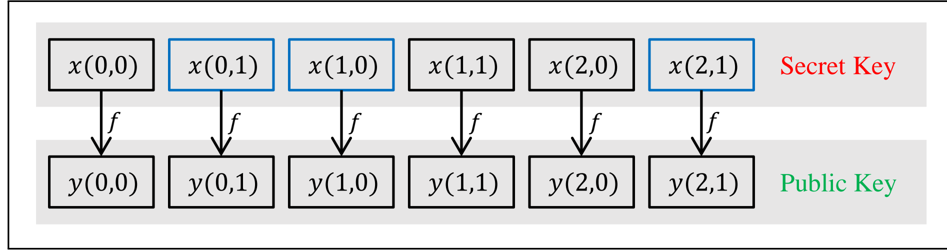


Figure 3.1: Illustration of the LD-OTS secret key, public key, and signature for $n = 3$. The signature $\sigma(m) = x(0,1)||x(1,0)||x(2,1)$ for $m = 101$ is highlighted in blue.

Key Generation

Given a security parameter n (message digest length and hash output length) and a cryptographic hash function $f : \{0,1\}^n \rightarrow \{0,1\}^n$, an LD-OTS key pair is generated as follows: The private key $\mathbf{X}$ is chosen uniformly at random:

$$\mathbf{X} = \big(x(0,0), x(0,1), x(1,0), x(1,1), ..., x(n-1,0), x(n-1,1)\big) \in_R (\{0,1\}^n)^{2\times n}$$

Where $\in_R$ denotes chosen uniformly at random. Therefore, there is one random n-bit string for each possible value (0,1) for each bit in the message string, i.e., $x(i,j) \in_R \{0,1\}^n, 0 \leq i < n; j \in \{0,1\}$.

The public key can be computed from the private key by applying the hash function to the secret key parts:

$$\mathbf{Y} = \big(y(0,0), y(0,1), y(1,0), y(1,1), ..., y(n-1,0), y(n-1,1)\big) \in (\{0,1\}^n)^{2\times n}$$

$$y(i,j) = f(x(i,j)), \quad 0 \leq i < n; j \in \{0,1\}$$

Thus, the public key has the same size as the private key. The public key computation is illustrated in Figure 3.1.

Signature Generation

Given the private key $\mathbf{X}$ and the digest $d \in \{0,1\}^n$, which is a fixed-length representation of the message $m \in \{0,1\}^*$ computed using a cryptographic hash function $h : \{0,1\}^* \rightarrow \{0,1\}^n$, a LD-OTS for m is generated by picking the parts of the private key that correspond to the bits in d:

$$\sigma(m) = \Big(x\big(0, d(0)\big), x\big(1, d(1)\big), ..., x\big(n-1, d(n-1)\big)\Big)$$

Thus, if a bit $d(i)$ in the digest is 0, $x(i,0)$ is used and if $d(i) = 1$, $x(i,1)$ is used. The signature, thus, has half the size of the public or private key, i.e., $\sigma(m) \in \{0,1\}^{n\times n}$.

Figure 3.1 illustrates in blue which parts of $\mathbf{X}$ are used for the signature. It can be seen that a single signature reveals half of the private key. This emphasizes the one-time nature of this signature scheme. If a key is used more than once, more and more of the private key is revealed. For example, if one first signs $d_1 = 0^n$ and then $d_2 = 1^n$ using the same LD-OTS key pair, the two signatures can be combined to form the entire private key, which entirely breaks the security of the scheme. However, recent research [BH16] found that the impact is not as drastic as the example above suggests, since d cannot be chosen arbitrarily by an adversary, but is a digest of a chosen message, which should be computed using a collision resistant one-way function. Therefore, the security of the scheme does not vanish after two messages, but "degrades gracefully" [BH16]. Nevertheless, the keys of LD-OTS should never be used more than once, since the security degradation is not intended and often not considered when choosing the security parameter n.

Signature Verification

Given a digest $d \in \{0,1\}^n$, the signature $\sigma(m)$ and the public key $\mathbf{Y}$, the signature can be verified by applying f to each part of $\sigma(m)$ which yields part of the public key. If these parts are equal to the actual public key, the signature is accepted, otherwise it is rejected.

$$f(\sigma(m)_i) \stackrel{?}{=} y(i, d(i)), \quad 0 \leq i < n$$

Discussion

The LD-OTS is a signature scheme that solely relies on the one-wayness of the function f. It is believed that there will still be functions that fulfil this requirement if quantum computers exist. Even if efficient quantum attacks are found against the currently standardized hash function families (SHA2 [Nat15a] and SHA3 [Nat15b]), they can be easily replaced by new resistant ones.

However, if quantum computers are available there is an efficient generic attack based on Grover's algorithm that halves the bit security of all cryptographic hash functions [HRS16b]. Therefore, it will be required to use twice the private key size n to achieve the same level of security in the quantum setting.

Although LD-OTS provides very strong security on minimal assumptions, it has some major downsides which prevented its wide adoption. Firstly, it is one-time, which is not suitable for the majority of use cases of digital signatures. A solution to this problems, i.e., the construction of a many-time signature scheme using a one-time signature scheme, is the MSS and will be explained in Section 3.2.

Secondly, the keys and the signatures are extremely large. A practical parameter choice would be $n = 256$ to provide a (non-quantum) security of around 128 bits, which would lead to a private and public key size of $2 \times n \times n = 131072 bits$ = 16 kibibyte (KiB). The signatures would then have half of this size, i.e., 8 KiB. When comparing this to RSA, where a signature has 384 bytes for a modulus of 3082, which according to NIST provides a similar security of 128 bits [Nat16a], LD-OTS are around 43× larger than RSA signatures which is not acceptable in many settings. This major disadvantage of LD-OTS is addressed by W-OTS which is introduced next.

3.1.2 Winternitz One-Time Signatures (W-OTS)

As concluded in the previous section, LD-OTS provides strong security, but never became practical because of the size of keys and signatures. W-OTS tries to mitigate this disadvantage while still providing similar security. It was first proposed in Merkle's thesis in 1979 [Mer79] and was described in detail by Dods et al. in 2005 [DSS05] and Buchmann et al. in 2009 [BDS09]. The idea of W-OTS is to trade-off the space and time, i.e., to reduce the space that is required for keys and signatures while increasing the time that is required for key generation, signature generation and signature verification. The required space is reduced by building hash chains that are then used to sign multiple bits per secret key block of length n. A hash chain is the repeated application of a hash function f to a value, e.g., $f(f(f(x))) = f^3(x)$ is a hash chain of length 3. W-OTS is parameterized by the Winternitz parameter $w = 2^t$ which determines how many bits are signed by each hash chain and, in consequence, how long the hash chains are. Thus, it is space-time trade-off parameter. Note that Dods et al. [DSS05] call $w = 2^t$ the Winternitz parameter, while in Buchmann et al. [BDS09] t is called the Winternitz parameter and named w. In this thesis, the original notation of Dods et al. is used.

W-OTS as LD-OTS requires a collision resistant one way function $f : \{0,1\}^n \rightarrow \{0,1\}^n$. Given w, the three lengths are computed:

$$\ell_1 = \left\lceil \frac{n}{t} \right\rceil \qquad \ell_2 = \left\lceil \frac{\lfloor log_2 \ell_1 \rfloor + 1 + t}{t} \right\rceil \qquad \ell = \ell_1 + \ell_2$$

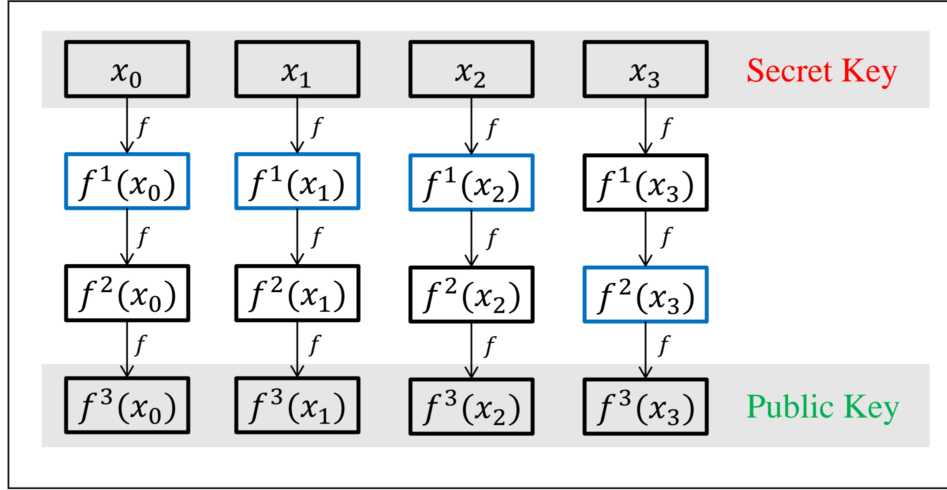


Figure 3.2: Illustration of the W-OTS secret key, public key, and signature for $t = 2$, $n = 4$, i.e., $\ell_1 = 2$, $\ell_2 = 2$, $\ell = 4$. The signature $\sigma(m) = f^2(x_3)||f^1(x_2)||f^1(x_1)||f^2(x_0)$ for the message $d = 10||01$ and the corresponding checksum $C = 01||01$ is illustrated in blue.

Since every chain is used to sign t bits, ℓ_1 represents the number of chains required to sign a message digest of length n. Additionally, a checksum, which will be introduced later, is used which requires ℓ_2 chains. ℓ is the sum of these two lengths and is the total number of required chains and, thus, key blocks of length n. Therefore, $\ell \cdot n$ is the size of the secret key, public key, and signature respectively. W-OTS is again fully specified by defining the algorithms for key generation, signature generation, and signature verification.

Key Generation

Given the security parameter n and length ℓ, the W-OTS private key $\mathbf{X}$ is chosen at random and the public key Y is computed from $\mathbf{X}$ by applying f $w-1$ times:

$$\mathbf{X} = (x_0, ..., x_{\ell-1}) \in_R \{0,1\}^{n\times\ell} \qquad \mathbf{Y} = (y_0, ..., y_{\ell-1}) \in \{0,1\}^{n\times\ell} \qquad y_i = f^{w-1}(x_i), \quad 0 \le i < \ell$$

Thus, the computations are very similar to LD-OTS, but the keys are smaller and the function f is evaluated multiple times. The public key computation is illustrated by Figure 3.2 using $n = 4$ and $t = 2$, i.e., $w = 4$.

Signature Generation

Given the private key $\mathbf{X}$ and the digest $d \in \{0,1\}^n$ of a message m, the digest d is divided into ℓ_1 blocks of t bits each: $d = b_{\ell-1}||...||b_{\ell-\ell_1}$. In case the length n is not divisible by t, zeros are appended. Afterwards the checksum for this message digest $C = \sum_{i=\ell-\ell_1}^{\ell-1}(w - b_i)$ is computed, which is again divided into t-bit blocks $C = b_{\ell_2-1}||...||b_0$. This checksum is crucial for the security of W-OTS and we will elaborate upon this in the discussion section. The blocks $b_{\ell-1}, ..., b_0$ are now used to calculate the signature:

$$\sigma(m) = \left(f^{b_{\ell-1}}(x_{\ell-1}), ..., f^{b_1}(x_1), f^{b_0}(x_0)\right)$$

Thus, the blocks b_i determine how often the one way function f is applied to the secret key blocks x_i. It is applied between zero and w-1 times. The computation is also illustrated in Figure 3.2 in blue. Hence, the intermediate values of the hash chains are included in the signature, which can also include some secret key parts x_i (if $b_i = 0$) and public key parts y_j (if $b_j = w - 1$).

Signature Verification

The signature verification algorithm takes as input a digest $d \in \{0,1\}^n$ of a message m, the signature $\sigma(m)$ and the public key $\mathbf{Y}$. First the checksum and the blocks $b_{\ell-1}, ..., b_0$ are computed as above. The signature is accepted if

$$y_i \stackrel{?}{=} f^{w-1-b_i}(\sigma_i), \quad 0 \le i < \ell$$

If this equality does not hold for at least one block, the signature is rejected. Hence, the hash function is applied $w-1-b_i$ times to the signature parts σ_i. The result is then expected to equal the public key block y_i, because $f^{w-1-b_i}(\sigma_i) = f^{w-1-b_i}(f^{b_i}(x_i)) = f^{w-1}(x_i) = y_i$.

Discussion

The W-OTS is very similar to the original LD-OTS. However, it implements a space-time trade-off which is controlled by the Winternitz parameter w. If w is large, the keys and signatures are small, but the time required for key generation, signature generation and signature verification is larger, since f is applied more often. Space might be much more constrained than time especially on embedded devices. Additionally, the hash functions are usually very efficient and might even be implemented in hardware components. Dods et al. recommend to use $t = 4$ (i.e., $w = 16$), because it provides short signatures, yet is still fast enough [DSS05].

As already mentioned, the checksum is crucial for the security of W-OTS. Assume that the checksum would not be included and the digest $d = 0^n$ is signed. This would result in all $b_i = 0$ and, thus, the entire private key would be published as a signature, which in consequence breaks the security. The checksum ensures that for any digest for at least a few blocks it holds that $b_i > 0$. It is important to note that W-OTS as LD-OTS is one-time. If a W-OTS key is used more than once, the security of the scheme degrades or vanishes. Bruinderink and Hülsing found that the security of W-OTS degrades much faster than that of LD-OTS when more than one message is signed using a single key [BH16].

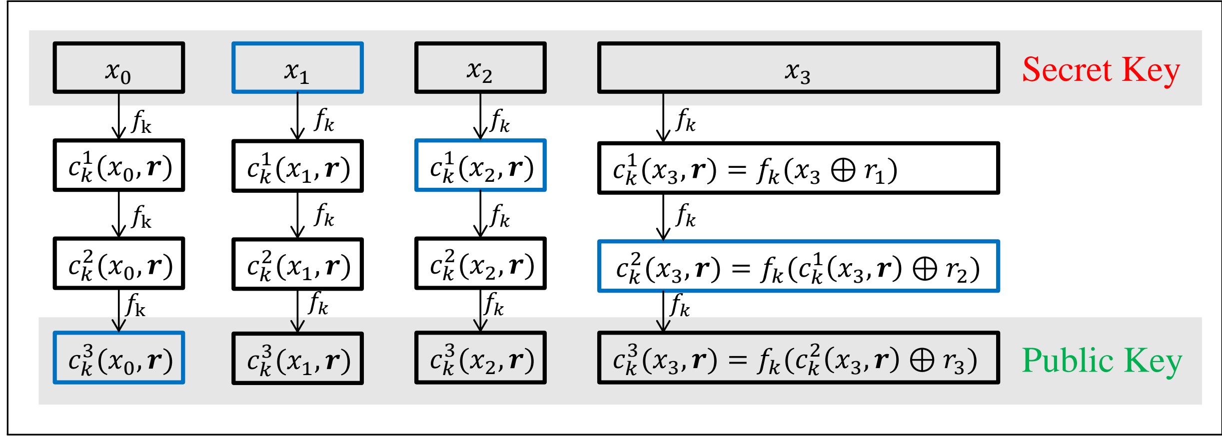


Figure 3.3: Illustration of the W-OTS+ secret key, public key, and signature. The signature $\sigma(m) = c_k^2(x_3,\mathbf{r})||c_k^1(x_2,\mathbf{r})||c_k^0(x_1,\mathbf{r})||c_k^3(x_0,\mathbf{r})$ for the message $d = 10||01$ and the corresponding checksum $C = 00||11$ is highlighted in blue.

3.1.3 W-OTS+

While W-OTS solves most of the disadvantages of LD-OTS is has been further refined by Hülsing et al. in 2013 to W-OTS+ [Hül13]. This extension is also directly used in the Internet Draft [HBGM17]. The main goal of the extension is to produce even smaller signatures without lowering the bit security of the scheme. The paper comes with extensive security proofs which underpin the security of the scheme. Additionally, W-OTS+ has weaker security assumptions about the used one-way function. It is only required to be second-preimage resistant for W-OTS+, while W-OTS requires a collision resistant one.

Since W-OTS+ only differs slightly from W-OTS which was explained in the last section, this section only emphasizes the major differences. Firstly, W-OTS+ does not apply f as a plain hash function to the secret values, but uses a keyed-hash function $f_k : \{0,1\}^n \rightarrow \{0,1\}^n, k \in \{0,1\}^n$. The n-bit key k used for this function is chosen at random at key generation time and is the same for all hash calls. Additionally, W-OTS chooses $w-1$ random n-bit bitmasks $\mathbf{r}$ which are used to mask the values passed to the hash function as illustrated later. Note that the Internet Draft [HBGM17] slightly deviates from the literature in the way the keys and bitmasks are generated since it is important for interoperability. It defines how they are generated from a random seed using a specific pseudorandom function (PRF).

The hash function used for computing the public key from the private key is replaced by the chaining function c_k:

$$c_k^0(x,\mathbf{r}) = x, \qquad c_k^i(x,\mathbf{r}) = f_k(c_k^{i-1}(x,\mathbf{r}) \oplus r_i), \qquad \mathbf{r} = (r_1,...,r_j), j > i$$

The public key can, thus, be computed by $y_i = c_k^{w-1}(x_i,\mathbf{r}), \quad 0 \le i < \ell$. Consequently, the function is also replaced in the signing algorithm (i.e., $\sigma_i = c_k^{b_i}(x_i,\mathbf{r})$) and the verification algorithm:

$$y_i \stackrel{?}{=} c_k^{w-1-b_i}(\sigma_1, \mathbf{r_{b_i+1,w-1}}), \quad 1 \le i \le \ell$$

$\mathbf{r_{b_i+1,w-1}}$ here denotes $(r_{b_i+1}, r_{b_i+2}, ..., r_{w-1})$.

It is important to note that both the key used for the hash function and the randomization bitmasks are part of the public key and, therefore, known to everybody. The public key computation and the signature generation are illustrated in Figure 3.3.

This simple modification greatly improves the security of the scheme, since finding a collision in f is no longer sufficient to forge a signature. This is proven by Hülsing in [Hül13] and he also proves that the bit security of W-OTS+ is $b \ge n - log_2(w^2\ell + w)$.

One additional minor difference between W-OTS and W-OTS+ is that W-OTS uses $C = \sum_{i=\ell-\ell_1}^{\ell-1} (w - b_i)$ for the checksum calculation while W-OTS+ uses $C = \sum_{i=\ell-\ell_1}^{\ell-1} (w - 1 - b_i)$.

3.2 Construction of Many-Time Signature Schemes

In the majority of use cases of digital signatures (e.g., signing of software, web pages, e-mails) an OTS is not suitable, because many (maybe hundreds or even thousands) different messages need to be signed and verified. Using a different key pair for each message introduces an extreme overhead especially because each public key needs to be distributed in an authenticated way.

Therefore, schemes have been proposed to construct many-time signatures using OTS (e.g., LD-OTS or W-OTS) as a building block. One of these schemes is XMSS which forms the basis of the Internet Draft of Hülsing et al. [HBGM17]. XMSS is an extended version of MSS which was introduced by Merkle in 1979 [Mer79]. The original MSS is introduced first in Section 3.2.1, followed by the extensions for XMSS in Section 3.2.2. Additionally, the Internet Draft describes a multi layer version of XMSS called XMSS^{MT} which will be outlined in Section 3.2.3

3.2.1 Merkle Signature Scheme (MSS)

The MSS is a stateful digital signature scheme built upon a one-time signature scheme like LD-OTS. A stateful signature scheme is a digital signature scheme which requires that after each signature generation the secret key is updated. If this update is not carried out properly, the security of the cryptographic scheme degrades or vanishes.

MSS was first mentioned in an report by Merkle in 1979 [Mer79] and described in more detail in 1990 [Mer90]. The following description is mainly based on the work by Buchmann et al. [BDS09].

Given the security parameter n, MSS requires a cryptographic hash function $g : \{0,1\}^* \rightarrow \{0,1\}^n$. In order to be able to sign 2^H messages, it uses 2^H OTS key pairs. The OTS public keys are then combined to a single n-bit MSS public key using a binary tree of height H. MSS is again fully defined by the three algorithms for key generation, signature generation and signature verification.

Key Generation

Given the tree height H, the key generation algorithm first generates 2^H OTS key pairs which yields $(sk_{OTS,i}, pk_{OTS,i})$, where $0 \leq i < 2^H$. The actual OTS scheme is not defined by MSS. Let sk_{OTS} denote all OTS secret keys and pk_{OTS} all OTS public keys. This generation might be done using a PRNG and on-the-fly during the public key computation, such that not the entire key needs to fit in memory (see optimizations section).

MSS then constructs a binary hash tree to compute a single MSS public key from the 2^H OTS public keys. The leaves of this tree are computed using the hash function $g : v_0[i] = g(pk_{OTS,i})$ for $0 \leq i < 2^H$. The inner nodes of the tree are computed by applying the hash function to the child nodes:

$$v_h[j] = g(v_{h-1}[2j] || v_{h-1}[2j+1]), \quad 1 \leq h \leq H, 0 \leq j < 2^{H-h}$$

The construction of the hash tree is illustrated in Figure 3.4. The root node is the MSS public key and denoted by $pk_{MSS} = v_H[0]$. The MSS private key sk_{MSS} consists of all OTS private keys sk_{OTS} and the index s of the next OTS key to use (initially $s = 0$), i.e., $sk_{MSS} = (sk_{OTS}, s)$. Instead of saving all sk_{OTS}, it is also possible to save an n-bit seed of a PRNG that was used to create the keys.

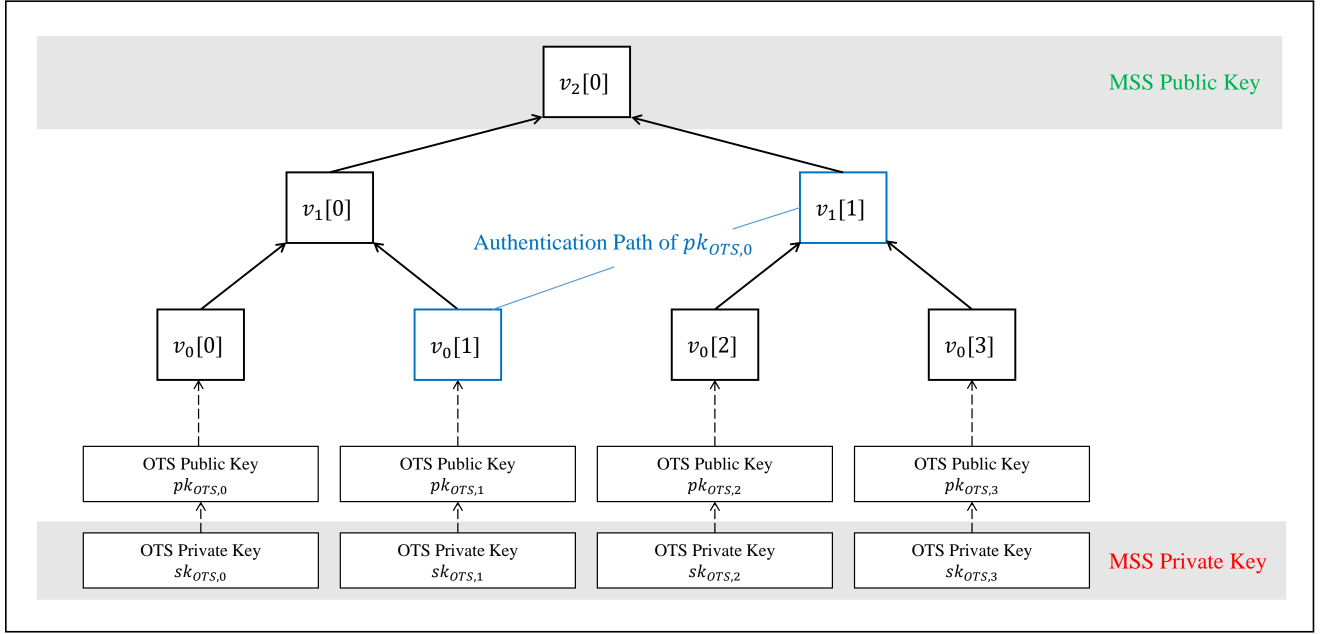


Figure 3.4: Construction of a Merkle tree using an OTS for $H = 2$. The authentication path of the first OTS key pair is highlighted in blue.

Signature Generation

Given a digest $d \in \{0,1\}^n$ of a message m and the MSS secret key $sk_{MSS} = (sk_{OTS}, s)$, first OTS is created using the s-th OTS secret key $sk_{OTS,s}$; the corresponding public key is $pk_{OTS,s}$. This yields σ_{OTS}. It is imperative to increment s in sk_{MSS} to ensure that this one-time key pair is not used again in the future. In addition to $pk_{OTS,s}$ the verifier requires several nodes of the Merkle tree to reconstruct the root of the hash tree. This is achieved by appending the authentication path $A_s = (a_0, ..., a_{H-1})$ to the signature, which contains one node in each layer of the hash tree. The a_h are either left or right neighbors of the nodes in the path from $v_0[s]$ to $v_H[0]$ and chosen using this formula:

$$a_h = \begin{cases} v_h[s/2^h - 1], & \text{if } \lfloor s/2^h \rfloor \equiv 1 \mod 2 \\ v_h[s/2^h + 1], & \text{if } \lfloor s/2^h \rfloor \equiv 0 \mod 2 \end{cases}$$

The authentication path is illustrated in blue in Figure 3.4. The MSS signature is, thus, $\sigma = (s, \sigma_{OTS}, pk_{OTS,s}, A_s)$. It is required to include the index s because the verifier needs to know which node in the authentication path A_s is a left and which is a right neighbor.

Signature Verification

Given the digest $d \in \{0,1\}^n$ of message m, a signature $\sigma = (s, \sigma_{OTS}, pk_{OTS,s}, A_s)$ and the MSS public key pk_{MSS}, the algorithm first verifies σ_{OTS} using $pk_{OTS,s}$. If the signature is valid, the leaf corresponding to $pk_{OTS,s}$ is calculated, i.e., $v_0[s] = h(pk_{OTS,s})$. Then the path from this leaf to the root is calculated using the authentication path A_s given in σ, i.e., $(p_0, ..., p_H)$ with $p_0 = v_0[s]$. The subsequent nodes can be calculated as follows:

$$p_h = \begin{cases} g(a_{h-1} || p_{h-1}), & \text{if } \lfloor s/2^{h-1} \rfloor \equiv 1 \mod 2 \\ g(p_{h-1} || a_{h-1}), & \text{if } \lfloor s/2^{h-1} \rfloor \equiv 0 \mod 2 \end{cases}$$

If the computed root p_H is equal to the given MSS public key pk_{MSS}, the signature is accepted. Otherwise it is rejected.

Optimizations and Discussion

A number of optimizations exist to further improve the practicability and efficiency of MSS. Firstly, as already mentioned, PRNG can be used to generate the OTS keys, such that it is sufficient to store only the seed of the PRNG and not all OTS secret keys. Secondly, Buchmann, Dahmen and Szydlo propose an algorithm, which is referred to as BDS-algorithm in the literature, for efficient authentication path computation which trades off space and time and reuses nodes of the authentication path of the previous signature [BDS09]. Finally, multiple layers of MSS trees can be used to form a hypertree. The leaves of the trees in the upper layers are used to sign the roots of the trees in the lower layers. This concept will be introduced in 3.2.3. For more details on optimizations, we refer to [BDS09], since they are out of scope of this thesis.

The presented scheme solves the problem that each OTS key pair can only be used once, by constructing a (small) single value from a large number of OTS key pairs. Although this results in additional computation overhead, it is still feasible when using appropriate optimizations.

3.2.2 The eXtended Merkle Signature Scheme (XMSS)

XMSS, which is directly used in the Internet Draft [HBGM17], improves MSS and was introduced by Buchmann and Hülsing in 2011 [BDH11]. It has minimal security requirements, since it only requires a second-preimage resistant hash function. Additionally, Buchmann and Hülsing prove that it is forward secure, efficient and provides existential unforgeability under adaptive chosen message attack (EU-CMA). It uses W-OTS (or its optimized version W-OTS+) as a building block. It is reasonable to use W-OTS+ with it because it has similar minimal security requirements. Since XMSS is very similar to MSS, only the main differences are emphasized here. For an extensive description of XMSS we refer to [BDS09] or [HBGM17]. XMSS mainly differs in two points from MSS:

Firstly, the tree construction is slightly modified. XMSS introduces randomization bitmasks that are generated randomly at key generation time. For each tree layer, two n-bit values $b_{l,h}, b_{r,h}$ (left bitmask, right bitmask) are generated. The left (right) bitmask is used to mask the left (right) child node before passing it to the hash function. The tree construction is thus changed to:

$$v_h[j] = h\left((v_{h-1}[2j] \oplus b_{l,h})||(v_{h-1}[2j+1] \oplus b_{r,h})\right)$$

This is a similar idea as in W-OTS+ and lowers the security requirements to the hash function h, since finding a collision is no longer sufficient to attack the scheme. The bitmasks are generated using a PRNG, i.e., a seed is chosen at random and the bitmasks are generated on-the-fly. This minimizes storage requirements. It is important to note, that the bitmasks (or the seed) are part of the public key and, thus, available to the verifier and a potential adversary.

Secondly, the calculation of the leaves of the Merkle tree is modified. While MSS simply applies the hash h to OTS public keys, which makes it vulnerable to collision attacks, XMSS uses a more sophisticated construction called L-trees. An L-tree, which is very similar to the Merkle tree, is a binary tree where the leaves are the n-bit parts of the W-OTS(+) public keys. It uses a similar tree construction rule with randomization bitmasks to construct a single n-bit root from the leaves. Since ℓ (i.e., the number of W-OTS public key blocks) is not a power of 2 in general, it is not always possible to construct a full binary tree. Therefore, the leaves that have no right neighbor are lifted in the tree, until they are a right neighbor of a different node.

There is also a difference between the original XMSS paper [BDH11] and the XMSS Internet Draft [HBGM17]: The paper describes the use of a PRNG to generate W-OTS secret keys, while the Internet Draft leaves this decision to the implementer. However, the bitmasks and keys which are used within W-OTS and XMSS have to be generated using fixed PRF, because they are required for verification as well. Thus, to enable interoperability, they must be generated in the same way.

Additionally, while the original paper [BDH11] is general, the XMSS Internet Draft [HBGM17] specifies allowed parameter sets for n, w, and h: w is always fixed to 16, n may be 256 bits or 512 bits, and h can be either 10, 16 or 20.

3.2.3 Multi-Tree XMSS (XMSS^{MT})

While an XMSS implementation which uses the optimized BDS algorithm provides sufficient performance during signature generation, it is still relatively slow in generating a new key pair, because this requires the construction of the entire hash tree. If this is not acceptable for some reason, the performance of both MSS and XMSS can be further improved by using multiple layers of trees. This method was initially named tree chaining [BDS09] and is referred to as XMSS^{MT} in the Internet Draft of Hülsing et al. [HBGM17]. Hülsing et al. also describe the method in [HRB13].

The idea is to use a hyper tree in which the upper layers are used to sign the roots of the layers below and only the lowest layer is used to actually sign messages. Thus, an XMSS^{MT} hyper trees consists of $T \geq 2$ layers of XMSS trees with heights $h_0, ..., h_{T-1}$, where h_0 is the height of the trees at the lowest level. The Internet Draft further restricts the parameters to $h_0 = h_1 = ... = h_{T-1}$, i.e., all trees have equal height. The OTS key pairs corresponding to the leaves of layer i are used to sign the roots of the trees on layer $i-1$. The root of layer $T-1$ is the XMSS^{MT} public key. Obviously, this is most sensible if a large number of messages is to be signed. In that case the use of a PRNG is highly recommended. Otherwise the required storage and the long time for random number generation outweigh the performance gain of XMSS^{MT}.

XMSS^{MT} has multiple effects on space and time. The impact can be illustrated by thinking of an example where $T = 4$ and $h_0 = h_1 = h_2 = h_3 = 20$. This parameter choice allows to sign a total of 2^{80} messages which is referred to in the literature as a "virtually unlimited" number of messages.

- **Key Generation:** To create the public key it is sufficient to calculate the only XMSS tree on the top layer $(T-1)$ of height 20 and thus requiring 2^{20} node computations in the XMSS tree (plus the leaf computation using L-trees). A single XMSS tree, which can be used to sign the same number of messages, i.e., having a height $h = 80$ would would require 2^{80} node computations, which is infeasible.

- **Signature Generation:** To enable a verifier to validate a signature, it is necessary to include one XMSS signature per tree layer which consists of the W-OTS+ signature, the W-OTS+ public key, the index of the used leaf and the authentication path. If the last leaf on a layer is used, the next tree needs to be generated which is signed by the next unused leaf of the active tree in the layer above. In the worst case this means that $T-1$ trees (all except the top layer) need to be computed to generate a single signature. This means that in the example a signature generation would require at most $3 \cdot 2^{20}$ node computations which is still significantly smaller than 2^{80}.

- **Signature Size:** To be able to verify an XMSS^{MT} signature, one XMSS signature per layer needs to be included, which increases the signature size. However, the authentication paths are shorter compared to a huge tree with $h = 80$. Therefore, the parameter T together with the heights h_i can be seen as space/time trade-off parameters similar to w within W-OTS.

- **Signature Verification:** Verifying an XMSS^{MT} signature requires the verification of d W-OTS+ signatures, the computation of d L-trees and the reconstruction of d XMSS tree roots, which requires a total of h hash computations. Thus, the verification of an XMSS^{MT} signature is significantly slower than an XMSS signature verification, which would only require to verify one W-OTS+ signature, compute one L-tree, and h hash function evaluation.

Thus, XMSS^{MT} can be used to balance the time required to generate a key pair and signature and space occupied by a signature. It is especially useful if a large number of messages have to be signed. It can

and should be used in conjunction with other optimizations like the BDS algorithm and a PRNG. The caching of the authentication paths is essential for the performance of XMSS^{MT}.

3.3 Related Work

Although this thesis is mainly focused on the XMSS standard, important related work needs to be discussed too. This section briefly describes and compares selected work to XMSS. The intention is to give enough detail about the schemes to allow a proper discussion of its side-channel vulnerability in the following chapters without going beyond the scope of this thesis. Firstly, another active IETF Internet Draft for hash-based signatures by McGrew et al. [MCF17] is introduced. Secondly, SPHINCS [BHH+15] is described, which extends XMSS and removes the statefulness, which allows a drop-in replacement of current signature schemes.

3.3.1 McGrew Internet Draft and Leighton-Micali Signatures (LMS)

The IETF has another active Internet Draft for hash-based signatures by McGrew et al. [MCF17]. The authors also state that their scheme "naturally resists side-channel attacks". Similar to XMSS, it is based on the work by Lamport, Diffie, Winternitz and Merkle. However, McGrew et al. call their OTS Leighton Micali one-time signature (LM-OTS) and the many-time signature scheme LMS. Leighton and Micali filed a patent on hash-based signature schemes which was granted in 1995 [LM95] and is now expired. However, their scheme is very similar to XMSS, at least from a side-channel perspective.

Similar to XMSS, they construct LMS using many LM-OTS. We briefly describe both with a focus on the main differences to XMSS.

Leighton-Micali One-Time Signatures (LM-OTS)

LM-OTS is an OTS based on W-OTS+, i.e., requires a second-preimage resistant hash function H. The private key is chosen randomly ($\mathbf{X} \in_R \{0,1\}^{\ell \times n}$), where n is the security parameter in bits and ℓ is the number of n-bit strings in the signature (computed similarly as in W-OTS).

Given a security string $S \in \{0,1\}^{2 \cdot n+32}$ (the same for all y[i]), the public key can be calculated by applying the chaining function $w-1$ times:

$$y[i] = c^{w-1}(S, x[i], i), \quad 0 \leq i < \ell$$

The chaining function is given by

$$c^0(S, x, i) = x \qquad c^j(S, x, i) = H(S || c^{j-1}(S, x, i) || \text{u16str}(i) || \text{u8str}(j-1) || 0x00)$$

where

- u16str(i) is the unsigned 16-bit representation of i
- u8str(j-1) is the unsigned 8-bit representation of j-1

The LM-OTS private key consists of the randomly chosen $x[i]$. The public key includes the computed $y[i]$ and the security string S. The rest of the scheme is very similar to XMSS, besides a slightly different checksum calculation. For more details consider [MCF17]

Leighton-Micali Signatures (LMS)

Similar to XMSS, LMS can be used to sign up to 2^H messages. This is achieved by using 2^H LM-OTS keys that are combined to a single public key using a Merkle tree. The nodes of the tree are addressed using the index r, where $r = 0$ is the root of the tree and $r = 2^{H+1} - 1$ is the rightmost leaf, i.e., the leaves of the tree have indices r with $2^H \leq r \leq 2^{H+1} - 1$.

The leaves of the tree are computed from the LM-OTS public keys using the hash function H:

$$T[r] = H(I||y[r - 2^H]||\text{u32str}(r)||0x03), \quad 2^H \leq r \leq 2^{H+1} - 1$$

where

- $||$ denotes concatenation
- I $\in_R \{0,1\}^{2\times}$ is a random security string (uniformly random, but the same for all leaves)
- u32str(r) is the unsigned 32-bit representation of r

The inner nodes of the tree are constructed by hashing the two child nodes:

$$T[r] = H(I||T[2*r]||T[2*r+1]||\text{u32str}(r)||0x04), \quad 0 \leq r < 2^H$$

As in XMSS the LMS private key consists of all LM-OTS private keys, which are possibly created using a PRNG, and the index q of the next unused key. The public key consists of the security string I and the root node $T[0]$.

The signature generation and verification are now straightforward and the same as in XMSS. Therefore, we skip their description and again refer to [MCF17]. Additional to the single tree variant, they also provide a multi-tree variant which is very similar to XMSS^{MT}.

Comparison

Despite the different naming, LMS is very similar to XMSS. McGrew et al. provide a direct comparison to XMSS [MCF17]:

- XMSS has slightly smaller signatures for the same bit-security level and maximum number of messages.
- LM-OTS is about four times faster than W-OTS+ when using a Merkle-Damgård-based hash function like SHA2, because W-OTS+ requires four calls to the compression function (two are required to determine k and r using a PRF, two are required for the keyed hash function), while LM-OTS only requires one call.

Additionally, a difference is that the Internet Draft of McGrew et al. is limited to SHA-256, while Hülsing et al. allow SHA-256, SHA-512, SHAKE128 and SHAKE256.

3.3.2 SPHINCS

The major downside of both XMSS and LMS is that they are stateful signature schemes, i.e., the signer needs to keep track of the last used OTS key pair to make sure that none is used twice. To solve this problem Bernstein et al. propose an extended scheme called SPHINCS [BHH+15] which is the first and only stateless hash-based signature scheme. Similar to XMSS^{MT} it uses a hyper tree of height h. The

upper layers use XMSS together with W-OTS+ to sign the roots of the child trees, while the lowest layer uses a Merkle tree with HORST for signing messages. HORST is a tree-based version of HORS which was introduced by Reyzin et al. in 2002 [RR02]. HORST and HORS, unlike W-OTS, are few-time signatures, i.e., can be used to sign a few instead of only one message. The more signatures that are generated using one HORST key pair, the more the security of the scheme degrades. We refer to [BHH+15] and [HRS16a] for a more detailed evaluation of the meaning of "few".

To illustrate the idea of SPHINCS, the few-time signature scheme HORST is introduced, but the construction of the Merkle tree which is similar to XMSS is skipped. The overall goal is to eliminate the state, i.e., there is no index s in the secret key.

HORST requires two cryptographic hash functions:

$$F : \{0,1\}^n \rightarrow \{0,1\}^n \qquad H : \{0,1\}^{2n} \rightarrow \{0,1\}^n$$

Additionally, HORST uses the parameters $k, \tau \in \mathbb{N}$. The message digest length is $m = k \cdot \tau$. For example SPHINCS-256 uses $\tau = 16$ and $k = 32$, i.e., $m = 512$.

In the following the key generation, signature generation and signature verification algorithms are described:

HORST Key Generation

Let $t = 2^\tau$. t random n-bit strings are generated which form the secret key:

$$\mathbf{sk} = (sk_1, ..., sk_t) \in_R \{0,1\}^{n \times t}$$

Then, a binary hash tree is constructed whose leaves are $L_i = F(sk_i)$. The inner nodes are computed using the hash function H and additional randomization bitmasks Q (for similar reasons as in W-OTS+ and XMSS). The root of the tree is the HORST public key $\mathbf{pk} \in \{0,1\}^n$. For SPHINCS-256 this means that the private key has a size of $2^{16} \cdot 256$ bits $= 16,777,216$ bits$=$ 2 mebibyte (MiB) and the public key is 256 bits$=$ 32 bytes plus the seed required for generating the randomization bitmasks. A total of 2^{16} calls to F and 2^{16-1} calls to H are required to compute the public key.

HORST Signature Generation

Given a message digest $d \in \{0,1\}^m$ and a HORST secret key $\mathbf{sk} = (sk_1, ..., sk_t)$, the message is split up into k parts, i.e., $d = d_0 || ... || d_{k-1}$, such that each d_i has a length of τ bits. The signature for this message then consists of $k+1$ parts $\sigma = (\sigma_0, ..., \sigma_k)$. The first k parts are computed as $\sigma_i = (sk_{d_i}, Auth_{d_i})$, where $Auth_\alpha$ is the partial authentication path in the HORST tree for leaf α up to a layer $\tau - x - 1$. σ_k then consists of all nodes of layer $\tau - x$. This use of partial authentication paths instead of full ones is decreasing the signature size. The value of x is chosen, such that $k(\tau - x + 1) + 2^x$ is minimal. [BHH+15]

HORST Signature Verification

Given a message digest $d \in \{0,1\}^m$, a signature σ and a HORST public key $\mathbf{pk}$, first the d_i's are computed as above. For each $\mathbf{sk}_{d_i}$ the verifier computes $F(sk_{d_i})$ and uses $Auth_{d_i}$ to compute the corresponding node on layer $\tau - x - 1$. It must be equal to the corresponding node in σ_k. Additionally, the nodes in σ_k are used to compute the root of the HORST tree, which must be equal to pk. If this equivalence holds for all M_i, the signature is accepted, otherwise it is rejected.

SPHINCS

HORST enables the signer to use one key pair more than one time. However, the security vanishes if it is used too often. Therefore, an additional mechanism is required that ensures that each key pair is only used a few times. SPHINCS achieves this by using many HORST key pairs and selecting a “random” one for each signature generation. This random selection is achieved by computing a randomized message digest of the message that is to be signed and then using a part of that hash as an index for the HORST key pair.

For example, SPHINCS-256 uses 2^{60} HORST key pairs. From these key pairs a virtual hypertree with five ayers of Merkle trees is constructed. Each layer has a height of 12. The upper four layers signs the roots of their ancestors, while the lowest layer signs the roots of the corresponding HORST keys.

It is important to note that this huge structure is never fully computed. It is essential to generate all the HORST and W-OTS+ private keys with a PRNG. Thus, for key generation it is sufficient to pick seeds for the PRNG and compute the one tree in the top layer. For signature generation one tree in each layer needs to be computed. By using these optimizations, the computations remain feasible.

Comparison

The goal of SPHINCS was to eliminate the statefulness from XMSS. This is achieved using the few-time signature scheme HORST and random key pair selection using the hash of the message. However, this comes at some additional cost. Firstly, the signature generation is more expensive, since optimizations like the BDS algorithm are no longer suitable, because the key pairs are not used successively but in random order. Secondly, HORST signatures are much larger than W-OTS+ signatures.

4 Side-Channel Analysis of XMSS

This chapter analyzes the side-channel vulnerability of XMSS [HBGM17], which was introduced in Chapter 3. Timing and power side-channel attacks are considered, which were introduced in Chapter 2. Section 4.1 presents an overview of the existing literature about side-channel analysis of hash-based signatures and other post-quantum schemes.

Since we want to keep the analysis as general as possible and are not in possession of a cryptographic device implementing XMSS, we start with a set of assumptions about the implementation in Section 4.2. These assumptions allow us to show that there is no exploitable side-channel under these assumptions and consequently, allows us to reduce the side-channel resistance of an implementation to the gratification of the assumptions. However, to reasonably evaluate the security of an actual implementation against side-channel attacks, it is necessary to analyze if these assumptions hold. A general discussion of the practicability of the assumptions is provided in Section 4.6.

Section 4.3 covers timing-related side-channels and Section 4.4 covers power-related side-channels. For both, we will first discuss the resistance of W-OTS+ and then evaluate if this can be used directly as an implication of the resistance of XMSS. It is shown that this holds under our assumptions, but where more practical ones are used, the resistance of XMSS is much weaker than W-OTS+. The analysis concludes with a discussion as to how our results can be generalized to the other schemes presented in Chapter 3.

4.1 Related Work

Both existing hash-based signature Internet Drafts state that the specified scheme “naturally resists side-channel attacks” without referencing related work [HBGM17, MCF17]. However, to the best of the author’s knowledge, only two publications explicitly cover the side-channel leakage of hash-based signature schemes [EvMY14, TE15]. Both papers conclude that the side-channel leakage is low and mainly relies upon the side-channel leakage of the used hash function and PRNG. While Taha et al. [TE15] only survey other papers, Eisenbarth et al. [EvMY14] perform experiments on the leakage using a hash function built with the block cipher AES. However, both papers provide only a brief side-channel analysis. Furthermore, they do not directly analyze XMSS and, thus, additional consideration is required.

Numerous papers analyze side-channels in implementations of other schemes that are believed to provide quantum-resistance. Silverman and Whyte [SW06] as well as Vizev [Viz07] describe a timing channel in the lattice-based encryption scheme NTRUEncrypt. The side-channel is due to a variable number of hash function calls in the decryption algorithm that depends upon the secret key as well as the message. Additionally, Lee et al. [LSCH10] and Zheng et al. [ZWW13] describe a power side-channel vulnerability in NTRUEncrypt.

The McEliece scheme, which is a code-based scheme, is also attacked by various papers using both timing and power side-channels. The timing side-channels are due to the variable execution time of the extended Euclidean algorithm [STM+08, Str10, SSMS09, AHPT11, Str13], the execution time of large matrix multiplications [AHPT11, vMG14] and variable cache access time to LUT used during matrix multiplication [STM+08, AHPT11]. A SPA attack can be used to recover the execution path which leaks information about the secret key [HMP10, MSSS11, vMG14], i.e., McEliece has conditional branches depending on secret data. Additionally, a sophisticated DPA attack targeting the syndrome computation is possible [CEvMS15].

4.2 Assumptions

To provide a sound analysis of side-channel resistance that is relatively independent of the actual implementation, we make two assumptions about the implementation of XMSS and W-OTS+, which allow for a sound proof of the side-channel resistance. Section 4.6 discusses the practicability of these assumptions for real-world implementations.

Assumption 1: It is assumed that the implementation under attack uses a PRNG to create the W-OTS+ signature keys on-the-fly when creating a signature and calculating the authentication path. The Internet Draft states that an implementation of XMSS *may* do this, but is not required to [HBGM17]. However, when thinking of practical implementations that will be required to sign a lot of messages, not using a PRNG will result in extremely large keys which will be impractical for most implementations, especially on embedded systems. Thus, assuming usage of a PRNG is legitimate.

Assumption 2: Both the implementation of the PRNG and the one-way function are assumed to have no side-channel leakage at all. This assumption allows to reduce the side-channel resistance of XMSS to the side-channel resistance of these two cryptographic primitives.

Having defined these assumptions, we can now start analyzing the schemes in the next sections.

4.3 Timing Side-Channels

Various timing side-channels have been described in Chapter 2, which can all be put down to a variation in the execution time depending on secret data. There are some reasons for timing variation like conditional branches that are easy to detect and others that require a more sophisticated analysis like cache access time.

The assumptions defined in the previous section imply that at least the PRNG and the one-way function do not contain a timing side-channel, which means that they do not leak information about their input. To determine if secret information might be leaked through a timing-side channel in W-OTS+ and XMSS, we need to look at all parts of the algorithms that process secret data.

4.3.1 W-OTS+

As illustrated in Section 3.1.3, the only secret data that is processed inside W-OTS+ are the secret key parts x_i. The used randomization elements r and the keys k are public values and, thus, are of no interest for a potential attacker. Figure 4.1 shows the secret parts of the W-OTS+ public key computation and signature generation. It is important to note, that not all intermediate values of the hash chains are secret values, since some of them are part of the generated signature ($c_k^{b_i}(x_i, r)$) or can be computed from them ($c_k^{\alpha}(x_i, r), \alpha > b_i$). Thus, all intermediate values $c_k^{\alpha}(x_i, r)$ for $\alpha < b_i$ are secret. Since the b_i's are not known before the actual message is signed, all intermediate values have to be considered secret.

The x_i are only used as input to the chaining function c_k. The chaining function applies the hash function f_k multiple times to the secret key parts x_i. During key generation, the number of hash calls is fixed ($w - 1$) and during signature generation, the number of hash calls solely depends on the message blocks (b_i). The blocks b_i only depend upon the message digest which is again of no interest to a potential attacker. According to *assumption 1*, the hash function itself does not contain a timing side-channel, therefore, evaluating $c_k(x_i, r)$ cannot leak anything about x_i with respect to timing.

Additionally, the output of the chaining function is directly part of the signature and is, therefore, considered to be known to the adversary. If $b_i = 0$ this means that the adversary knows part of the secret key. However, this does not undermine the security of W-OTS+ due to its one-time nature and the checksum, i.e., partial leakage is an intrinsic characteristic of hash-based signature schemes. Even if the entire message digest that is being signed would equal 0, the checksum would still ensure that some b_i's are greater than zero.

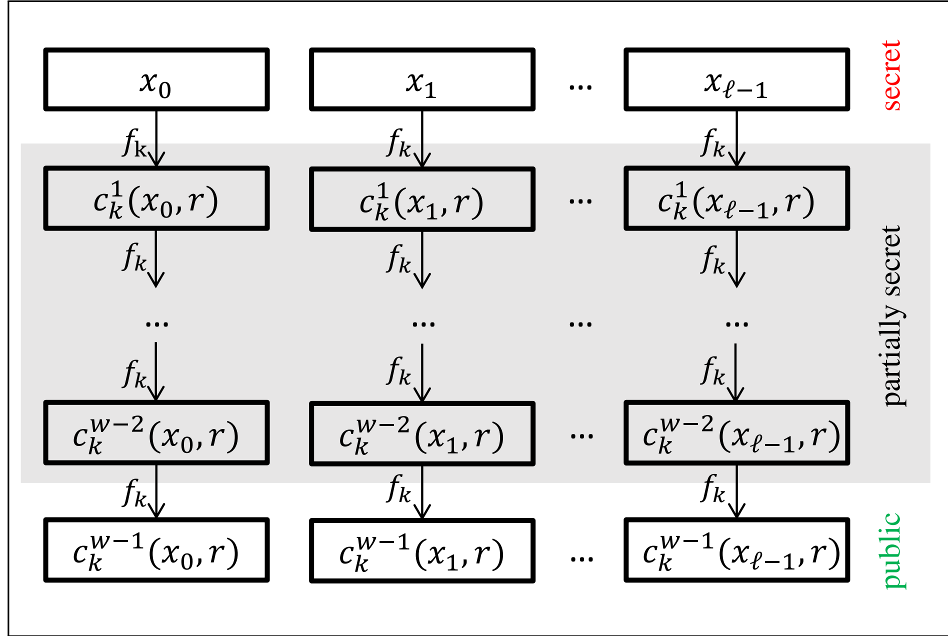


Figure 4.1: Parts of W-OTS relevant for side-channel analysis

4.3.2 XMSS

The last section concluded that W-OTS+ does not leak any information about the secret key parts via timing side-channels. Since XMSS is built using many W-OTS+ keys, intuitively XMSS provides this resistance as well.

Figure 4.2 summarizes the parts of XMSS which are relevant for side-channel analysis. The entire XMSS tree is public and, thus, leakage agnostic, i.e., even if it is leaked entirely, the adversary does not learn anything secret. This includes the W-OTS+ public keys and the intermediate values in the L-trees, which are used to compute the XMSS tree leaves. The relevant parts are shown in the lower part of the figure and include the seed used for the pseudorandom W-OTS+ secret key generation and the W-OTS+ secret key itself. As already mentioned, the intermediate values in the W-OTS+ hash chains need to be considered secret if they are below the parts contained in the W-OTS+ signature. If W-OTS+ has not yet been used to sign a message, all intermediate values need to be considered secret. Thus, the secret key in XMSS either consists of many W-OTS+ private keys or a random seed used to create them. When using the first option, the timing resistance of XMSS directly follows from the resistance of W-OTS+. When using the latter option, we also need to make the assumption that the PRNG does not leak anything about the random seed. This ensures that at least the W-OTS+ part of the XMSS signature generation and public key computation does not have an exploitable timing side-channel.

4.3.3 Discussion

We have shown that timing side-channels in W-OTS+ can only be used to extract information about the b_i's, which the adversary either already knows (known message attack) or chooses himself (chosen-message attack). The additional computation in XMSS may leak some information about the internal state s, but this value is already contained in a valid signature and cannot be used to mount an attack. Therefore, we conclude that XMSS does not have a timing side-channel, if the assumptions from Section 4.2 hold.

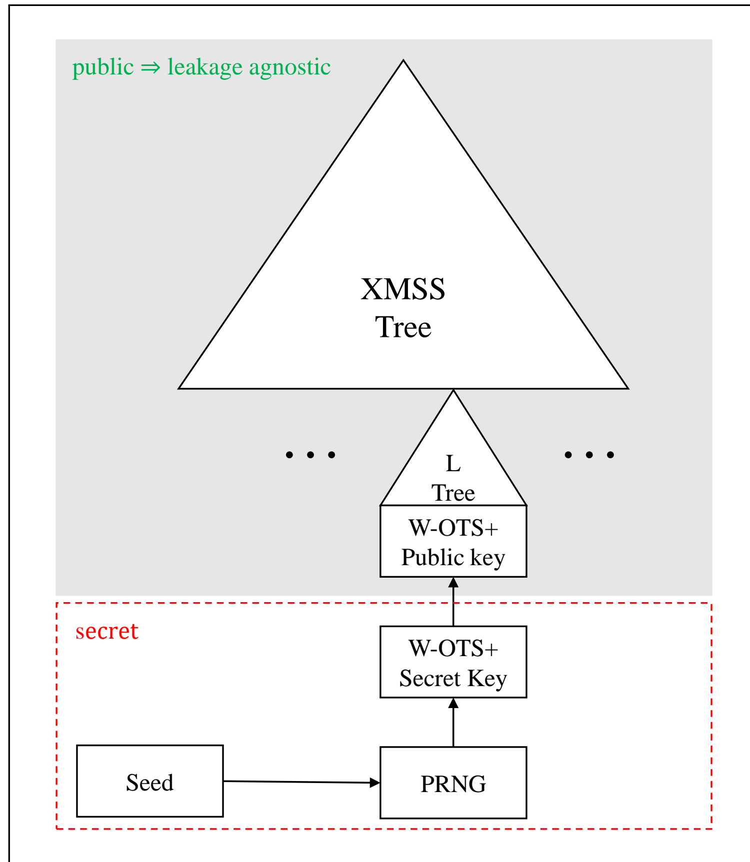


Figure 4.2: Parts of XMSS relevant for side-channel analysis

4.4 Power-related Side-Channels

For mounting a power analysis attack, it is required to find a function that is evaluated somewhere in the cryptographic implementation and depends on a part of the secret key and some known variable input data. It is important that this function can be calculated for all possible values of the key, i.e., using a function on the entire key is not feasible. Usually, this is done for small blocks of the key (e.g., 8 bits).

To analyze XMSS with respect to such power side-channels, we need to look for potential target functions that could be used. To keep the analysis sound and simple, we use the same set of assumptions as for the timing attacks, i.e., the hash function and PRNG are not vulnerable. This is a very unrealistic assumption, since every function has some leakage through power side-channels, even if it is very small.

4.4.1 W-OTS+

The interesting function within W-OTS+ is the chaining function

$$c_k^0(x,r) = x, \qquad c_k^i(x,r) = f_k(c_k^{i-1}(x,r) \oplus r_i), \qquad r = (r_1,...,r_j), j > i.$$

At first sight this seems to be a perfect fit for a power analysis attack, because for $i = 1$ the signer calculates $x \oplus r$, where x is some secret key block and r_i is a randomization bitmask which is known to the verifier and attacker. However, this function is only called twice: once during key generation and once during signature generation. Additionally, the r_i is the same for both evaluations. This prevents differential attacks like DPA, which rely upon different inputs to the attacked primitive. Additionally, SPA attacks can, in the best case scenario, recover the HW of the processed values. If it is assumed that the

Table 4.1: Guessing entropy for HW-leakage per byte

HW	# values	left entropy (bits)	leakage (bits)
0	1	0	8
1	8	3	5
2	28	4.81	3.19
3	56	5.81	2.19
4	70	6.13	1.87
5	56	5.81	2.19
6	28	4.81	3.19
7	8	3	5
8	1	0	8

$\oplus$ is computed per byte, the approximate number of bits leaked can be if there is no noise at all can be computed:

If the adversary knows $HW(x_i) = 0$, this directly implies that $x_i = 0$ and he knows all the bits of this sub-key. If $HW(x_i) = 1$, the possible values for x_i are $\{1,2,4,8,16,32,64,128\}$, which leads to an entropy of $log_2(8) = 3$ bits, i.e., the adversary successfully recovered 5 of 8 bits of information. This calculation is shown in Table 4.1 for all other Hamming weights as well. Since every key byte value should have equal probability, we can derive a formula for the average leakage:

$$leak_{avg} = \sum_{i=0}^{8} \frac{1}{256}\binom{8}{i}\left(8 - \log_2\binom{8}{i}\right) = 2.5 \text{ bits}$$

Assuming a 256 bit key, this would leak approximately 80 bits. Since then still around 176 bits of guessing entropy are left, this is not enough for a successful attack. Furthermore, if only the HW of 32-bit words is leaked, which is more likely, the leakage reduces to approximately 3.5 bits per word, i.e., 28 bits per 256 bit key.

This shows that the first iteration of the chaining function does not provide sufficient leakage to recover the key or restrict the remaining search space enough to find the key by guessing. In the next step, the hash function is called upon the intermediate result, which according to our assumption, does not leak additional information.

The following iterations of the chaining function behave exactly the same. Each time a few bits of the intermediate value $c_k^{i-1}(x,r) \oplus r_i$ are leaked during the $\oplus$ computation.

Note on leakage resilience: When analyzing the leakage resilience of cryptographic schemes, it is usually assumed that the leakage function is chosen by an attacker and only bounded by the amount of bits leaked (referred to as λ). Standaert et al. introduce a "future computation attack" that uses this assumption to attack an iterative PRNG [SPY+10], where the leakage of each iteration is combined to reveal some result of future computation (a pseudorandom value). The idea is that, if in every iteration λ bits are leaked, and the cryptographic device executes the function often enough, the adversary eventually knows the final value. This trivially breaks an iterative PRNG. However, Standaert et al. find that it is very impractical, since (1) an attacker is usually not able to choose the leakage function and (2) an iteration cannot leak anything about computations in the future. While we could modify this attack to break W-OTS+ as well by assuming that each chaining function iteration leaks λ bits from x_i, this is not reasonable. When a cryptographic hash function is used to construct the chaining function, it should behave like a random oracle, i.e., the output of the hash function is entirely random for a previously unseen input. This implies that if an adversary knows λ bits of x_i and λ bits of $h(x_i)$ is not equivalent to knowing $2 \cdot \lambda$ bits of x_i.

4.4.2 XMSS

One might think that if W-OTS+ is resistant to side-channels, this would directly imply the resistance of XMSS as well. However, one major difference that makes XMSS more vulnerable than W-OTS+, is that the W-OTS+ key generation is called much more often during authentication path computation. To calculate the authentication path of the leftmost W-OTS+ public key, all other keys are required as well. If it is assumed that the keys are always computed on-the-fly and nothing is cached in between signatures, the W-OTS+ public key computation and the PRNG need to be executed for each single signature for all keys. This was found by Eisenbarth et al. [EvMY14] for MSS, but it directly applies to XMSS as well. It was found in the previous section that the leakage resistance of W-OTS+ can mainly be guaranteed because both the key generation and the signature generation are only executed once and this alone prevents power analysis attacks. However, if the key generation is executed more often, a more elaborate analysis is required.

Let the XMSS tree have height H, i.e., having 2^H leaves. For a signature using the W-OTS+ private key at index s, the signer needs to first compute the W-OTS+ signature using $sk_{OTS,s}$ and then the authentication path for $v_0[s]$. The signature is not a real problem, but the authentication path calculation requires that all other $v_0[i]$ are computed as well. While some nodes of the authentication path can be reused, some need to be recomputed. If it is assumed that the signer does not reuse nodes at all, we know that at the time of the signature generation for index s, the $sk_{OTS,s}$ already leaked a few times before. To be more specific, it leaked once during initial key generation and once for each signature that was created before, which means that including the current leakage, it leaked $s+2$ times and will also leak for all signature generations in the future (2^H+1 times if all keys are used).

At first sight it might seem useless to recover a W-OTS+ key that was already used, since it is forbidden to use it again. However, an attacker will not care that a key was already used. He can just create an arbitrary number of valid signatures using this one key and the same authentication path used in the legitimate signature.

When again assuming the most powerful side-channel adversary [SPY+10] who can choose the leakage function arbitrarily and adaptively change it for each signature generation, this leads to a leakage of $2^{H+1} \cdot \lambda$ bits, where λ is a bound for the bits leaked per W-OTS+ key generation and signature generation. Even a small bound λ trivially breaks the security of XMSS for any reasonable choice of H and n. However, in practice such an adversary does not exist. When considering a real-world leakage model, the attack becomes infeasible: During each signature generation the W-OTS+ chaining function is called with the exact same inputs to produce the same W-OTS+ public keys. While this is useful for filtering out noise which is inevitable in every power analysis attack, the leaked information is still very limited. An adversary cannot hope to recover more than the HW of each intermediate result. We already found for timing attacks, that the actual computations in the Merkle tree are not interesting for a potential attacker and, thus, their power side-channel vulnerability analysis is skipped here. When combining this finding with our assumptions and the result of the last section, we find that XMSS has the same leakage as W-OTS+, but the adversary can use the multiple computations to reduce the noise, which is useful.

This can be easily generalized to the hyper tree variant XMSS^{MT}, since an XMSS^{MT} signature generated using a hypertree with T layers can be viewed as T independent XMSS signatures from a side-channel perspective. One major difference is that the W-OTS+ signature generations on the upper layers are executed more than once (if no caching is implemented). Intuitively this seems to provide more leakage than the single tree variant of XMSS. However, as we have seen, in the straightforward XMSS the public key computation is executed many times, which provides the same or even more leakage than a signature generation. Actually, W-OTS+ key generation and signature generation are equivalent, since for the signature generation the signer also needs to calculate the W-OTS+ public key $c_k^{w-1}(x_i, r)$ because it is

not cached. The only difference is that during signature generation, the signer additionally outputs the intermediate results $c_k^{b_i}(x_i, r)$. Since the computations are the same, the leakage cannot be any different.

Chapter 6 shows, that while this equivalent resistance of XMSS and XMSS^{MT} applies for passive side-channel attacks, it is very different when considering more active attackers that are able to insert faults into the computations.

4.4.3 Discussion

Both W-OTS+ and XMSS provide strong side-channel resistance under our assumptions. We showed that W-OTS+ is resistant to the majority of attacks due to its one-time nature alone, which limits the number of traces that can be obtained. This robustness is somehow weakened by the on-the-fly key generation within XMSS.

Thus, we come to the conclusion, that XMSS cannot be attacked by power analysis attacks, if the used hash function and the PRNG are leakage resistant. This is a similar result as related work found for other hash-based signature schemes [EvMY14, TE15].

Although this result is very promising, it is crucial to ensure that the assumptions hold in an actual implementation.

4.5 Generalization to Other Hash-Based Schemes

The above analysis was limited to XMSS which is also the main focus of this thesis. However, the other schemes are so similar that everything found in this analysis is directly applicable to them as well. Yet, for some key differences additional analysis is required. This section provides a very brief comparison from a side-channel perspective starting from the predecessors of W-OTS+ and XMSS (Section 4.5.1), the McGrew Internet Draft (Section 4.5.2) and SPHINCS (Section 4.5.3). The same set of assumptions is used without explicitly mentioning them.

4.5.1 LD-OTS, W-OTS and MSS

The side-channel attack vectors of LD-OTS, W-OTS and MSS seem even smaller than the ones of W-OTS+ and XMSS:

Timing Attacks: The public key computations of both LD-OTS and W-OTS have a constant number of hash function evaluations. Therefore, a timing side-channel does not exist. The signature generation for LD-OTS has a fixed number of hash calls (half of the public key computation) which also means that no timing attack can be mounted. The runtime of a W-OTS signature generation only depends on the message to be signed and therefore does not leak any valuable information to an attacker. For MSS, the same reasoning as for XMSS can be used.

Power Analysis Attacks: Both LD-OTS and W-OTS do nothing but apply the hash function to the private key parts. The bitmasks in W-OTS+ which introduced a minor power side-channel are not present in the previous schemes. Thus, a side-channel resistant hash function is sufficient for guaranteeing the side-channel resistance of LD-OTS and W-OTS. MSS performs no computations on secret data other than the underlying OTS. The on-the-fly key generation might be useful for an attacker to reduce noise, however, an attack seems equally unlikely as for XMSS. Obviously, if a PRNG is used, it must be side-channel resistant.

Thus, we conclude that they also naturally resist the considered side-channel attacks under our assumptions.

4.5.2 LMS

Since LM-OTS it is based on W-OTS the differences are very small. The side-channel vulnerability seems equal as well:

Timing Attacks: Looking at the chaining function $c^j(S, x, i)$, we see that it only applies the hash function H. Obviously, the timing of this chaining function does not leak any valuable information since it only depends on j which is not secret. The construction of LMS is similar to XMSS with only small variations in the use of constants and randomization elements. Since they are all public, no additional timing side-channels can occur.

Power Analysis Attacks: The included S (constant randomization string), i (index of the hash chain), j (index in the hash chain) and D_ITER (constant 0x00) seem to provide some additional power side-channel at first sight when LM-OTS public keys are computed on-the-fly and multiple times. However, the values are fixed and are therefore not suitable for a power analysis attack. Additionally, they are only passed as input to the hash function which is assumed to be leakage resistant in our analysis. The same reasoning applies to LMS: All intermediate values inside the Merkle trees are public and, thus, side-channel agnostic.

Thus, we conclude that LMS is inherently secure against timing and power side-channel attacks if the PRNG and hash function implementations are side-channel resistant.

4.5.3 SPHINCS

Unlike the other three schemes, SPHINCS is using HORST to sign messages. However, we conclude SPHINCS is equally side-channel resistant as XMSS under our assumptions.

Timing Attacks: The number of hash function evaluations for both HORST key generation and HORST signature generation is independent of the secret key. Thus, if the hash function does not contain a timing side-channel, we are safe to conclude that HORST is timing side-channel resistant. The rest of the SPHINCS construction is only using W-OTS+ and XMSS and can thus not contain a timing side-channel.

Power Analysis Attacks: The leaves of the HORST tree are constructed from the randomly generated secret key parts sk_i by applying the hash function F. Therefore, HORST is side-channel resistant if F is side-channel resistant, since everything else can be considered public. Combining this resistance with the reasoning of W-OTS+ and XMSS, we are confident that SPHINCS has only negligible side-channel leakage if the used hash functions and the PRNG are secure.

Thus, we conclude that both HORST and SPHINCS can be considered side-channel resistant if the building blocks (hash function F and PRNG G) are properly protected.

4.6 Pseudorandom Number Generator (PRNG) and Hash Function Side-Channel Resistance

The previous subsections concluded that W-OTS+, XMSS, and XMSS^{MT} provide strong side-channel resistance under the assumption that the used hash function and the PRNG are side-channel resistant. Although the actual fulfillment of this requirement is implementation specific, this section presents a discussion of general side-channel resistance of the used building blocks.

4.6.1 Hash Function Side-Channel Resistance

The hash function evaluation within the chaining function of W-OTS+ is the only computation that is performed on the W-OTS+ secret key and, therefore, presents the only target for a side-channel adversary. However, a hash function per se cannot be vulnerable or resistant to side-channel attacks, since it can be used in numerous ways which do not necessarily involve a secret key. Thus, it depends on

how the hash function is used and it is only sensible to analyze the side-channel resistance for an actual scheme using the hash function in a certain way.

For the reader's convenience the definition of the W-OTS+ chaining function is repeated:

$$c_k^0(x,r) = x, \qquad c_k^i(x,r) = f_k(c_k^{i-1}(x,r) \oplus r_i), \qquad r = (r_1, ..., r_j), j > i$$

The keyed hash function f_k is implemented using either a hash function of the SHA2 or SHA3 function family using the following construction [HBGM17]:

$$f_k(x) = \texttt{SHA-256}(\texttt{toByte}(0,32)||k||x)$$

$$f_k(x) = \texttt{SHA-512}(\texttt{toByte}(0,64)||k||x)$$

$$f_k(x) = \texttt{SHAKE-128}(\texttt{toByte}(0,32)||k||x, 256)$$

$$f_k(x) = \texttt{SHAKE-256}(\texttt{toByte}(0,64)||k||x, 512)$$

where $\texttt{toByte}(a,b)$ is the big-endian b-byte encoding of a. SHA-256 and SHA-512 are in the SHA2 function family [Nat15a], whereas SHAKE-128 and SHAKE-256 are the variable output length SHA3 functions [Nat15b]. The second argument of SHAKE-128 and SHAKE-256 corresponds to the hash digest length. Note that the key k, which is used within f_k is a public randomization element, is computed from the public seed and, thus, known to the adversary and fixed for a certain hash chain.

Several side-channel attacks, which are all DPA attacks, have been proposed on both SHA2 and SHA3 hash function in the context of Hash-based message authentication code (HMAC) [ZKSH12, TS13, BBD+13, MTMM07]. When computing an HMAC, both the key and the message are passed to the hash function. Since the key is fixed and the message is variable for successive computations of HMAC, an adversary can attack the key by creating multiple HMAC for different messages and analyze the differences in the computations.

However, this is not applicable for the W-OTS+ chaining function, since it does not process variable data for the same key. Thus, we conclude that it is not vulnerable to timing and power analysis attacks:

Timing Attacks: Both SHA2 and SHA3 create digests for inputs of arbitrary length by splitting them up into several blocks and processing them iteratively. Since the input, which is passed to the hash function, has a fixed length in the W-OTS+ chaining function, the number of iterations used to process it is constant and, thus, cannot present a timing side-channel. For SHA2, which is based on the Merkle-Damgård construction, the message blocks are combined to a digest by iteratively calling a compression function [Nat15a]. The number of compression function calls is fixed for a fixed input size. For SHA3, which is a Sponge construction, these function calls are called the "absorb" phase and are followed by the "squeeze" phase which produces the digest [Nat15b]. The number of "absorb" and "squeeze" steps is constant for a fixed input size. Thus, a timing side-channel can only be located within the SHA2 compression function and the SHA3 "absorb" and "squeze" procedure. None of these functions contains conditional branches depending on the input passed to the function. Additionally, all computation within these functions is based on simple arithmetic which has a constant execution time. The compression function of SHA2 only consists of bitwise AND, bitwise OR, XOR, addition modulo 2^{32}, negation, cyclic shift, and non-cyclic shift operations. The SHA3 "absorb" and "squeeze" steps only require bitwise AND, XOR, rotation, and negation operations. Thus, we conclude that both SHA2 and SHA3 run in fixed time for a fixed input size, which prevents all categories of timing attacks for W-OTS+ and consequently for XMSS and XMSS^{MT}.

Power Analysis Attacks: We already concluded that DPA attacks are not applicable to the hash function within W-OTS+, since the input to the hash function is always the same for a certain key. Additionally, SPA cannot be used to successfully attack W-OTS+ because there are no conditional branches depending upon the input passed to the hash function. Thus, we conclude that the hash function as it is used within W-OTS+ is resistant to power analysis attacks.

4.6.2 PRNG Side-Channel Resistance

There are both, leakage resilient PRNG [SPY+10, TRS16, YSPY10] and side-channel resistant implementations of PRNG [CDK+10, BSVS16]. However, all the schemes proposed and implemented in these publications are fundamentally different from the PRNG which is recommended for XMSS. Thus, additional consideration is required. The XMSS Internet Draft recommends the use of one of the following constructions to produce a pseudorandom value from a random seed and index i:

$$\texttt{SHA-256}(\texttt{toByte}(3,32)||SEED||\texttt{toByte}(i,32))$$

$$\texttt{SHA-512}(\texttt{toByte}(3,64)||SEED||\texttt{toByte}(i,32))$$

$$\texttt{SHAKE-128}(\texttt{toByte}(3,32)||SEED||\texttt{toByte}(i,32),256)$$

$$\texttt{SHAKE-256}(\texttt{toByte}(3,64)||SEED||\texttt{toByte}(i,32),512)$$

The security of the PRNG must match the security of the used hash function used within W-OTS+ and XMSS, i.e., if the implementer decides to use $n = 256$ and the SHA2 function family, the first PRNG must be used. Their side-channel resistance is analyzed next, building upon the conclusions already drawn for hash functions:

Timing Attacks: The length of the input passed to the hash function is fixed, thus, the constant execution time of the hash function implies timing side-channel resistance of both the SHA2 and SHA3 PRNG.

Power Analysis Attacks: The non-existence of conditional-branches depending upon the input of the hash function implies that no SPA can be mounted upon any of the recommended PRNG. However, all four constructions present a good candidate for a DPA attack, since the hash functions are evaluated for the same seed with different indices. The full Chapter 5 is devoted to the DPA attack vulnerability of the PRNG and, therefore, skip it for now.

Discussion

We have shown that the hash functions (SHA2 and SHA3), which are used within the W-OTS+ chaining function, and the recommended PRNG, which can be used to generate the W-OTS+ secret keys, are not vulnerable to timing side-channel attacks. Additionally, the hash functions within the W-OTS+ chaining function cannot be attacked by either SPA or DPA attacks. The recommended PRNG cannot be attacked by SPA, but is vulnerable to a DPA, which will be covered in Chapter 5.

It is very important to note that the hash functions used may be replaced in future versions of XMSS, e.g., if SHA2 and SHA3 become insecure due to the discovery of new attacks or computational advances. It is imperative that the replacement is chosen such that it provides similar side-channel resistance.

Additionally, note that XMSS Internet Draft [HBGM17] does not standardize which PRNG must be used, but does only recommend one. Thus, an implementation may choose a different one which then requires additional side-channel resistance consideration.

5 Power Analysis Attack on the PRNG in XMSS

In the previous chapter we reached the conclusion that an implementation of XMSS can provide strong side-channel resistance only of the underlying hash function and PRNG are side-channel resistant. However, to the best of the author's knowledge there are no power analysis attacks on hash functions alone, but only on HMAC. This chapter extends one of these attacks to hash-based PRNG.

Section 5.1 describes a DPA attack on SHA2 HMAC upon which our proposed attack is based. Section 5.2 introduces a vulnerable SHA2 PRNG and describes the necessary steps to adopt the attack, including the assumptions being made about the adversary. Our implementation consisting of a power consumption simulator and the actual DPA is described in Section 5.3. Section 5.4 presents the experimental results of our attack simulations. Section 5.5 discusses how the results found by our experiments apply to practical PRNG, e.g., the one described in the XMSS Internet Draft. [HBGM17]

The source code that can be used to simulate this attack can be found at `https://github.com/mkannwischer/xmss-prng-dpa` and is published under a 2-clause BSD license.

5.1 A DPA Attack on SHA2 HMAC

Since the PRNG, which is suggested by the Internet Draft [HBGM17] to create the W-OTS+ secret keys within XMSS, is based on the hash function families SHA2 and SHA3, it is sensible to look at several publications that analyze their side-channel resistance. To the best of our knowledge, there are only publications that cover hash function side-channel resistance in the context of HMAC. McEvoy et al. propose a DPA attack on a SHA2 HMAC [MTMM07] which is further elaborated by Belaïd et al. [BBD+13]. Additionally, similar DPA attacks on SHA3 HMAC are proposed by Zohner et al. [ZKSH12] and Taha et al. [TS13]. This chapter focuses on SHA2 but the attacks can be adapted for SHA3 as well.

We briefly summarize the attack on SHA2 HMAC upon which our attack is based following the description of Belaïd et al. [BBD+13]. An HMAC for the hash function H, which is used to authenticate a message m, can be computed using the key k, by applying the hash function twice [KBC97]:

$$\texttt{HMAC}(m,k) = H\left((k \oplus opad)||H\left((k \oplus ipad)||m\right)\right)$$

The bitmasks $opad$ and $ipad$ denote to constant values $0x5c5c...5c$ and $0x3636...36$. When using a Merkle-Damgård-based hash function, the key is padded to the block length of the hash function (e.g.,

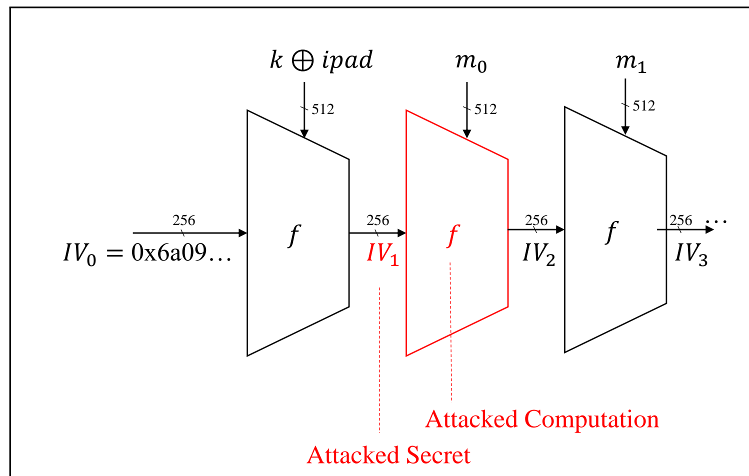


Figure 5.1: DPA on SHA-256 HMAC (simplified from [BBD+13])

512 bits for SHA-256), such that each evaluation of H results in at least 2 evaluations of the compression function f. The message block passed to the first evaluation consists of the masked key only, while the second evaluation processes the first part of the message itself. The process of the HMAC calculation is illustrated in Figure 5.1. First, the compression function f is called with the masked key ($k \oplus ipad$) and the fixed initialization vector (IV). Then for each block in the message m, an additional call to f, iteratively combines the resulting IV from the previous iteration with 512 bits of the IV. The construction is similar for other SHA2 hash lengths, but uses different block sizes.

The goal of an adversary is to be able to create a valid message authentication code (MAC) for an arbitrary message m. It is assumed that an adversary is able to query a cryptographic device with usually several thousand messages $m' \neq m$ (either chosen or known), while recording the power consumption of the device. Additionally, an HW leakage model is assumed.

Since the first evaluation only processes the key, but no variable data, it is not possible to mount a DPA attack on the computations inside. Instead, Belaïd et al. target the second evaluation of f, which processes the first block of m and the result of first evaluation of f, which we let denote to IV_1. The computations inside f can be used to entirely recover IV_1 which is enough to forge the inner part of the HMAC.

Algorithm 5.1 SHA-256 compression function f [Nat15a]

1: Input: IV (256 bit), m_i (512 bit)
2: $W_t \leftarrow m_i^{(t)} \quad 0 \leq t \leq 15$
3: $W_t \leftarrow \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-16}) + W_{t-15} \quad 16 \leq t \leq 63$
4: $A \leftarrow IV_0; B \leftarrow IV_1; C \leftarrow IV_2; D \leftarrow IV_3; E \leftarrow IV_4; F \leftarrow IV_5; G \leftarrow IV_6; H \leftarrow IV_7;$
5: **for** $t = 0; t < 64; t{+}{+}$ **do**
6: $\quad T1 \leftarrow H + \Sigma_1(E) + Ch(E,F,G) + K_t + W_t$
7: $\quad T2 \leftarrow \Sigma_0(A) + Maj(A,B,C)$
8: $\quad H \leftarrow G; G \leftarrow F; F \leftarrow E;$
9: $\quad E \leftarrow D + T1$
10: $\quad D \leftarrow C; C \leftarrow B; B \leftarrow A$
11: $\quad A \leftarrow T1 + T2$
12: **end for**
13: **return** $[IV_0 + A, IV_1 + B, IV_2 + C, IV_3 + D, IV_4 + E, IV_5 + F, IV_6 + G, IV_7 + H]$

$$Ch(x,y,z) := (x \wedge y) \oplus (\neg x \wedge z) \qquad Maj(x,y,z) := (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\Sigma_0(x) := ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \qquad \Sigma_1(x) := ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$

$$\sigma_0(x) := ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \qquad \sigma_1(x) := ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

The actual attack is based upon intermediate values inside the SHA2 compression function shown in Algorithm 5.1. The entire function uses arithmetic on unsigned 32-bit words including bitwise-and ($\wedge$), xor ($\oplus$), negation ($\neg$), addition modulo 2^{32} ($+$), circular-shift ($ROTR$) and non-circular-shift (SHR). In lines 2 and 3 the message block m, consisting of 16 512-bit words, is expanded to 64 words $\mathbf{W_t}$. Then the algorithm iterates 64 times combining the input vector IV with one word $\mathbf{W_t}$ per iteration. It is important to note, that the adversary wants to recover IV, i.e., the values of $A, ..., H$ before line 5, while he knows the message blocks and, thus, the words $\mathbf{W_t}$.

Notation: Let $D^{(i)}$ denote to the value of D before iteration $t = i$, thus $D^{(0)} = IV_0$. Similarly $T1^{(i)}$ is the value of $T1$ that was computed during iteration $t = i - 1$. Additionally, let values that are different for each HMAC generation denote to bold letters (e.g., $\mathbf{W_t}$), while values that are the same for all generations are in standard letters (e.g., $T1$)

Using this information, the adversary now mounts several DPA attacks building upon each other to recover $A^{(0)}, ..., H^{(0)}$. The first value the adversary recovers is the initial value of D. This is done by first recovering $\delta^{(1)} := H^{(0)} + \Sigma_1(E^{(0)}) + Ch(E^{(0)}, F^{(0)}, G^{(0)}) + K_0$ using the computation $\mathbf{T1^{(1)}} \leftarrow \delta^{(1)} + \mathbf{W_0}$ from line 6. Since $\mathbf{W_t}$ is known and variable and $\delta^{(1)}$ is fixed and secret, a DPA can be used to recover $\delta^{(1)}$. Once the adversary knows $\delta^{(1)}$, he can compute $\mathbf{T1^{(1)}}$ for each known word $\mathbf{W_0}$. The second DPA attack then recovers $D^{(0)}$ from $\mathbf{E^{(1)}} \leftarrow D^{(0)} + \mathbf{T1^{(1)}}$ using known values for $\mathbf{T1^{(1)}}$.

Building upon the recovered values of $\mathbf{T1^{(1)}}$, another 6 DPAs in the first and second iteration can be used to recover the values of $E^{(0)}, F^{(0)}, A^{(0)}, B^{(0)}$ and $G^{(0)}$:

- DPA 3: $T2^{(1)}$ is recovered from $\mathbf{A^{(1)}} \leftarrow \mathbf{T1^{(1)}} + T2^{(1)}$ (line 11) in iteration $t = 0$
- DPA 4: $E^{(0)} \left(= F^{(1)}\right)$ is recovered from $\mathbf{E^{(1)}} \wedge F^{(1)}$ in Ch (line 6, $t = 1$)
- DPA 5: $F^{(0)} \left(= G^{(1)}\right)$ is recovered from $\neg\mathbf{E^{(1)}} \wedge G^{(1)}$ in Ch (line 6, $t = 1$)
- DPA 6: $A^{(0)} \left(= B^{(1)}\right)$ is recovered from $\mathbf{A^{(1)}} \wedge B^{(1)}$ in Maj (line 7, $t = 1$)
- DPA 7: $B^{(0)} \left(= C^{(1)}\right)$ is recovered from $\mathbf{A^{(1)}} \wedge C^{(1)}$ in Maj (line 7, $t = 1$)
- DPA 8: $G^{(0)} \left(= H^{(1)}\right)$ is recovered from $T1^{(2)} \leftarrow H^{(1)} + \Sigma_1(\mathbf{E^{(1)}}) + Ch(\mathbf{E^{(1)}}, F^{(1)}, G^{(1)}) + K_t + \mathbf{W_t}$ (line 6, $t = 1$)

Finally, the adversary only misses $C^{(0)}$ and $H^{(0)}$. $H^{(0)}$ can be computed easily from line 6 (t=0), since everything except $H^{(0)}$ is known. Belaïd et al. state that $C^{(0)}$ can be computed from line 7 (t=0), since $A^{(0)}$, $B^{(0)}$ and $T2^{(1)}$ are known. However, since Maj consists of two bitwise AND operations, this will not work for all values of $A^{(0)}$ and $B^{(0)}$. Therefore, we instead propose to attack line 9 in iteration t=1, where $C^{(0)} \left(= D^{(1)}\right)$ is combined with a known and variable $\mathbf{T1^{(1)}}$. Thus, the full attack requires 9 DPA, which depend on each other.

5.2 Attack Design and Adversary Model

To the best of the author's knowledge there is currently no attack available on hash-based PRNG. However, the HMAC construction above looks very similar to the PRNG suggested by the XMSS Internet Draft [HBGM17] for the generation of W-OTS+ secret keys: To generate 2^H W-OTS+ secret keys, which each consists of $\ell \cdot n$ bits from a secret n bit seed, we use two layers of PRNG. First, we generate a n-bit intermediate secret value for each leaf of the XMSS tree:

$$SEED_{W-OTS+,j} = PRNG_{XMSS}(SEED, j) \quad 0 \leq j < 2^h$$

These intermediate values are then used to calculate the actual W-OTS+ secret key parts:

$$sk_{W-OTS+,i} = PRNG_{XMSS}(SEED_{W-OTS+,j}, i) \quad 0 \leq i < \ell$$

For the SHA2 function family and $n = 256$,the Internet Draft recommends the following construction for $PRNG_{XMSS}$:

$$PRNG_{XMSS}(SEED, i) = \texttt{SHA-256}(\texttt{toByte}(3, 32)||SEED||\texttt{toByte}(i, 32))$$

where $\texttt{toByte}(x, y)$ corresponds to the big-endian encoding of x to y bytes. It is similarly defined for $n = 512$ and the SHA3 hash function family. As in the SHA2 HMAC construction the first 512-bit message block consists of the seed and a padding ($\texttt{toByte}(3, 32)||SEED$). The second message block solely consists of the index i and the padding and length of the message (as defined in [Nat15a]). Trying

to apply the attack of Belaïd et al. [BBD+13], we notice that the message words W_0 and W_1, which were used to mount the DPA attack, are always zero for any reasonable parameter choice. Due to the big-endian encoding, these 32-bit words would only change if more than 2^{448} keys would be generated, which will never happen. If the known values are fixed, a DPA attack does not work.

Since the XMSS Internet Draft does not specify which PRNG must be used because it does not affect interoperability, an implementation might as well use a different method of pseudo secret key generation. We propose the following vulnerable PRNG:

$$sk_{W-OTS+,i} = PRNG_{vuln}(SEED, i) = \texttt{SHA-256}(\texttt{toByte}(3, 32)||SEED||\texttt{SHA-256}(i))$$

This PRNG does provide similar cryptographic security in the standard model, but can be attacked if physical attacks are considered. Due to the hashing of the index i, the message words W_0 and W_1 in the second evaluation of the compression function are uniformly distributed which allows an adversary to mount the DPA attack proposed by Belaïd et al. [BBD+13].

We first implement the attack on $PRNG_{vuln}$ in Section 5.3 and then analyze how the attack can be adapted to attack the original $PRNG_{XMSS}$ under some (unrealistic) assumptions in Section 5.5.

The goal of the adversary is to recover IV_1, which is the result of the first evaluation of the compression function f. Having recovered IV_1 the adversary can compute $sk_{W-OTS+,i}$ for any choice of i, which enables him to forge signatures for arbitrary messages (i.e., universal forgery), thus, it entirely breaks the security of XMSS.

The attack on $PRNG_{vuln}$ uses the following adversary model and assumptions: We assume that the scheme is implemented on a cryptographic device which leaks the HW of the processed values. The adversary is able to collect D power traces for different indices i, which most likely means that he must be in possession of the cryptographic device at least for a short period of time. The number of required traces is to be found by the experiments conducted in this thesis. It is important to note, that XMSS uses the PRNG extensively, i.e., a single signature generation leaks traces for many executions of PRNG. If the implementation does not use optimized authentication path computation (e.g., using the BDS algorithm), each signature generation calls the PRNG 2^H times, although it might be difficult to locate the single executions in the trace recorded during an entire signature generation. Additionally, note that the values used for attacking the IV_1 do not depend upon the message signed by XMSS. Thus, the adversary is not required to choose or know the signed messages and they are not required to be variable.

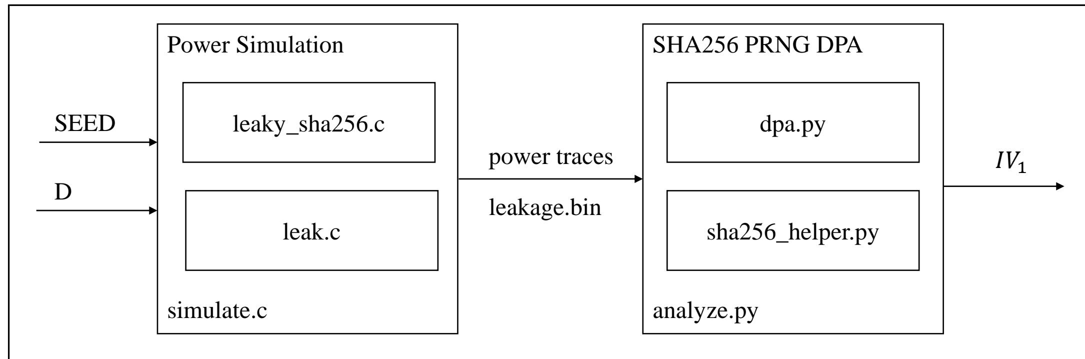


Figure 5.2: Simulation of a DPA attack on a SHA2-based PRNG

5.3 Implementation

To validate that our attack indeed can be used to recover IV_1 and in consequence generate all W-OTS+ secret keys, we created a proof-of-concept implementation of the attack. The source code of our implementation is available at `https://github.com/mkannwischer/xmss-prng-dpa`. It includes scripts that can be used to reproduce the results presented in this thesis.

Figure 5.2 illustrates the general architecture of our implementation. Since an actual hardware implementation was not available and is beyond the scope of this thesis, we implemented a power simulator which is capable of creating power traces in the hamming weight leakage model. It was written in the C programming language and consists of a leakage library (leak.c) and a custom implementation of SHA-256 (leaky_sha.c), which leaks intermediate values using the leakage library. The details of the power simulator are described in Section 5.3.1. The SHA-256 implementation was cross-checked with the OpenSSL implementation [Ope] to ensure it works correctly.

The resulting power traces are then passed to the analysis code (analyze.py) written in Python. It contains a more general DPA library (dpa.py) and some supportive functions specific to SHA-256 (sha256_helper.py).

All modules, including the libraries, are created as a part of this thesis and are meant to be reused in future work.

5.3.1 Power Simulation

The first part of the proof-of-concept implementation is a power simulator used to create the traces and shown as the left part of Figure 5.2. It mainly consists of three modules which are explained briefly:

leak.c: We created a general library which is responsible for leaking to a file. It implements different leakage modes: `HW`, `HW_BYTE`, `HD_R`, `HD_R`. In the `HW`-mode the HW of the unsigned 32-bit integer result of each computation is leaked. The `HW_BYTE` mode works similarly, but leaks the HW of each of the four bytes in each word. The `HD_L`/`HD_R`-mode leaks the HD of the left/right operand and the result (both 32-bit unsigned ints). Since the maximum HW or HD for all modes is 32 we use a single byte representation and leak them to a binary file. The module provides four procedures, which are self-explanatory: `leak_start(enum leakage_type type, const char *filename)`, `leak_end()`, `leak_pause()`, `leak_resume()`. Additionally, for each arithmetic operation (and, plus, left shift, right shift, xor), a separate function is provided which does the calculation and leaks the values corresponding to the leakage mode. The function returns the result, such that the computation only needs to be done once. For example, a bitwise-and computation would be performed by calling

```
result = leak_uint_and(a, b);
```

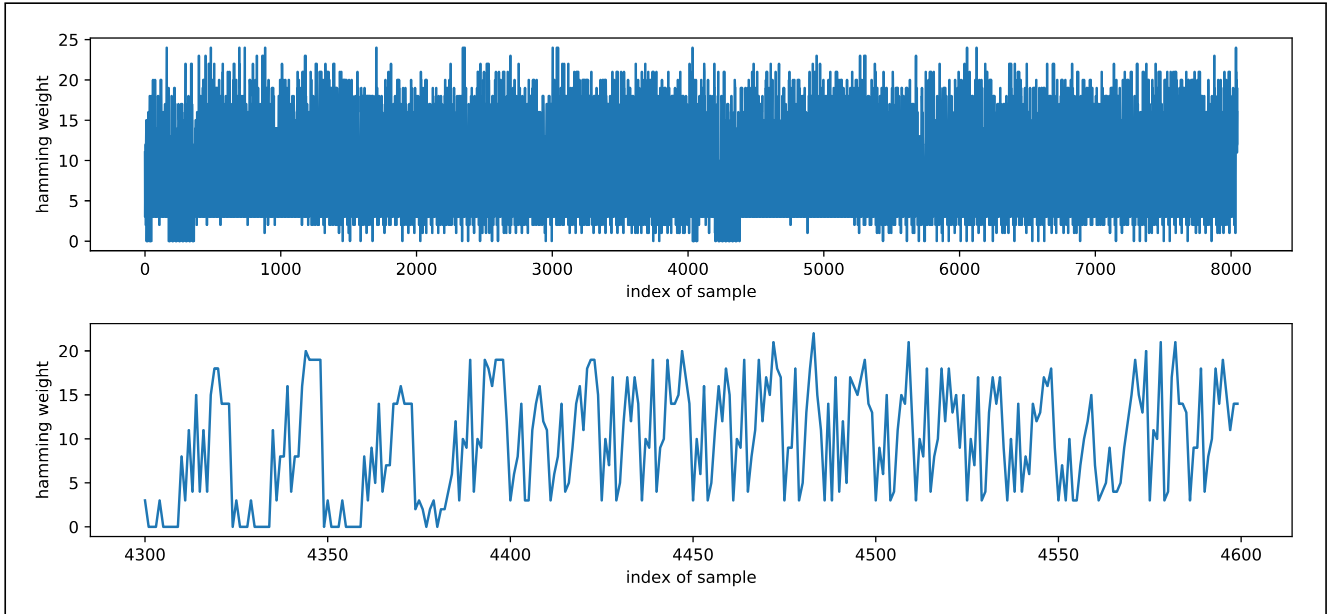


Figure 5.3: Simulated power trace for $PRNG_{vuln}$ in the HW model for 32-bit words. The upper plot shows the full trace, while the lower plot shows a zoomed view on the beginning of the second compression function evaluation

leaky_sha256.c: We created a straightforward implementation of SHA-256 following the NIST specification [Nat15a] using 32-bit unsigned integers. Next, we replaced each arithmetic operation with a call to one of our leaking functions.

simulate.c: The entry point of the simulation is the main-procedure in *simulate.c*. As input it requires the number D of different inputs for which power traces should be generated. Additionally, a secret seed can be provided as a hexadecimal string. If it is not provided, a random one is generated. The module initializes the leakage library and calls the $PRNG_{vuln}$ for indices $0 \leq i < D$. To keep the traces short, leakage is only activated during the outer hash computation.

Figure 5.3 shows a plot of a power trace simulated by our power simulator using the 32-bit HW leakage mode. The full trace, which is shown in the upper plot, has a length of around 8000 samples. The beginning of each of the both calls to the compression function f can be clearly identified at sample 0 and around sample 4000, because there is a long period of low power consumption (HW is 0). This is caused by the message schedule computation (compare Algorithm 5.1 line 3), where a lot of values of W_i are zero due to the padding in both blocks. The lower plot shows a zoomed view on the same trace starting at the beginning of the second compression function execution, which will be used to recover the secret IV_1.

Naturally, the simulation introduces several huge simplifications for this attack:

- *All our traces are perfectly aligned*, i.e., the same sample of two traces corresponds to the same computation. The alignment is required for a DPA to properly find the correlations and can be tedious to achieve if they are measured physically. [MOP07]

- *Our leakage is noise free*. In an actual attack there will be noise both introduced by the measurement setup and the physical properties of the cryptographic device. Noise can usually be mitigated by using more traces. [MOP07]

- *We know the implementation*. Since we created the SHA-256 implementation and possess the source-code, we know how the algorithm is implemented. We also know which sample in a trace corresponds to which computation, which helps for debugging.

5.3.2 DPA

The traces generated by the power simulator are fed into a Python script that implements the DPA proposed by Belaïd et al. [BBD+13] following the strategy described in 2.1.2. We decided to use Python for the DPA, since NumPy[1] allows the efficient and easy-to-read implementation of the required matrix operations. Since a DPA requires the computation of hypothetical power consumption values for each possible key hypothesis, our implementation recovers each byte of IV_1 separately. At first we assume that we have a byte-wise leakage of the HW (i.e., using the `HW_BYTE` mode in the simulator), which allows the recovery of the key with very few traces. However, since this is not realistic, we extend this later to work with the leakage of the HW per 32-bit word using partial DPA.

We separated three different modules:

- *dpa.py*: Implementing the general DPA attack on addition and bitwise-and.
- *sha256_helper.py*: Providing SHA256 specific helper functions, e.g., the computation of $Maj, Ch, \Sigma_0, \Sigma_1, \sigma_0$, and σ_1.
- *analyze.py*: Entry point of the DPA and containing the part of the attack that is specific for this attack, i.e., it implements the 9 DPA.

```
1  # analyze.py
2  # Given d: Dx4, T: DxT
3  carry = np.zeros([len(T)])
4  delta_0 = dpa.dpa_addition(T, d[:, 0], carry)
5
6  carry = (delta_0, d[:, 0]) // 256
7  delta_1 = dpa.dpa_addition(T, d[:, 1], carry)
8
9  # dpa.py
10 # T: DxT, d: Dx1, carry: Dx1
11 def dpa_addition(T, d, carry):
12 # H: Dx256
13 H = np.zeros([len(T), 256])
14 for hyp in range(0,256):
15 H[:, hyp] = hw(d + hyp + carry)
16 # R: Tx256
17 R = correlate(T, H)
18 return R.max(0).argmax()
```

Listing 5.1: DPA attack on modular addition

Listing 5.1 briefly sketches the implementation of the very first DPA used to recover $\delta^{(1)}$ in $T1^{(1)} \leftarrow \delta^{(1)} + W_0$. We assume in line 1, that we have a matrix T ($D \times T$) containing all D simulated power traces of length T and the already computed known data block W_0 for each index i ($0 \leq i < D$) and stored each byte in d ($D \times 4$), where `d[:, 0]` corresponds to the least significant bytes. Since we are attacking addition, we need to take care of the carry bit between the attacked bytes. For the least significant byte there is no carry. Therefore, it is initialized to zeros (line 3). Line 4 then calls the DPA library using the entire matrix T, the least significant bytes of d and the zero-carries. The actual DPA is implemented by the `dpa_addition()` function, for each data point d and each hypothetical key ($0 \leq hyp < 256$), it computes the hypothetical power consumption value using the hamming weight of the sum of d, hyp and the carry bit, resulting in a matrix H of dimension $D \times 256$. Matrices H and T are then used to compute the correlation coefficient R ($T \times 256$) in line 17. The actual correlation coefficient computation is not shown here, but was implemented efficiently using matrix operations and the equation for R in Section 2.1.2. Once R is computed, we simply find the index of the maximum value within R which

[1] http://www.numpy.org/

corresponds to the key candidate with the highest probability. After the least significant byte of $\delta^{(1)}$ is recovered, the adversary computes the carry bits by adding the recovered value to each known value d and applying integer division by 256. Having recovered the carry, the recovery of the more significant byte is straightforward. Similarly, the other bytes of δ are recovered. The implementation of the DPA for a bitwise AND is even more straightforward, since no carries are needed.

One problem that occurs when attacking addition or AND, is that some keys cannot be recovered easily. For example, when attacking an AND operation, the secret value 0 cannot be recovered, since this will always result in a zero HW. However, constant values in the trace cannot be used to find correlations. Yet, an adversary might still detect that there is no high correlation for any key candidate and, thus, deduce that the key must be zero.

Partial DPAs

Up to this point, we assumed that the implementation leaks the HW of each byte separately, such that we can mount independent DPA upon them. However, since SHA2 only involves unsigned 32-bit arithmetic, a byte-wise implementation is highly unrealistic. Most implementations will use 32-bit words and, thus, only leak the HW of the entire words. Luckily, the strategy can be adapted and still be used to recover each byte separately, although requiring a much higher number of traces. The adapted technique is called partial DPA and also evaluated by Belaïd et al. [BBD+13]

For the least significant byte, we use the exact same code as illustrated in Listing 5.1. This works because the HW of the 32-bit words still correlates with the hypothetical power consumption values calculated on a per byte basis. Although the correlation is much lower, it is still possible to recover the byte if enough traces are available. For the second-least significant byte, we can use the knowledge about the HW of the result of the least significant byte, which leads to higher correlation values and a higher success probability. Similarly, the correlation values increase for the more significant bytes. Usually, we have a maximum correlation value of 0.4 for the least significant byte and 0.5, 0.7 and 1.0 for the more significant bytes. This is due to the much more precise prediction of the HW if we already know the HW of other bytes of the same word. For the most significant byte, we can perfectly predict the HW of the result for each possible key candidate. Since the simulated trace is noise free, we have a perfect correlation and, thus, a correlation coefficient of 1.0.

Belaïd et al. [BBD+13] extensively studied partial DPA and concluded that they are practical. We managed to reproduce their experimental results without any problems.

5.4 Results

We validated that our proposed attack works by performing experiments using the implemented power simulator. We evaluated the success probabilities for both, the 8-bit HW leakage model and the 32-bit HW leakage model. The results are summarized next.

5.4.1 8-Bit Hamming Weight Leakage Model

We started evaluating our proposed attack in the 8-bit HW leakage model which allows the attack of each byte operation separately. Figure 5.4 illustrates two columns of the correlation matrix $\mathbf{H}$ for a DPA attack on a single 8-bit addition operation which, in this case, is the computation of the least significant byte of $\mathbf{T1}^{(1)}$. The upper plot shows the correlation values over time for the correct key hypothesis (34), while the lower plot illustrates those of an arbitrary wrong key hypothesis (66). Note that the correlation for all key candidates is very low for most of the time, but contains several peaks at the beginning of the first round of the second compression function evaluation (around sample 21000). The first peak denotes the operation we are actually targeting and the following smaller peaks are computations on the

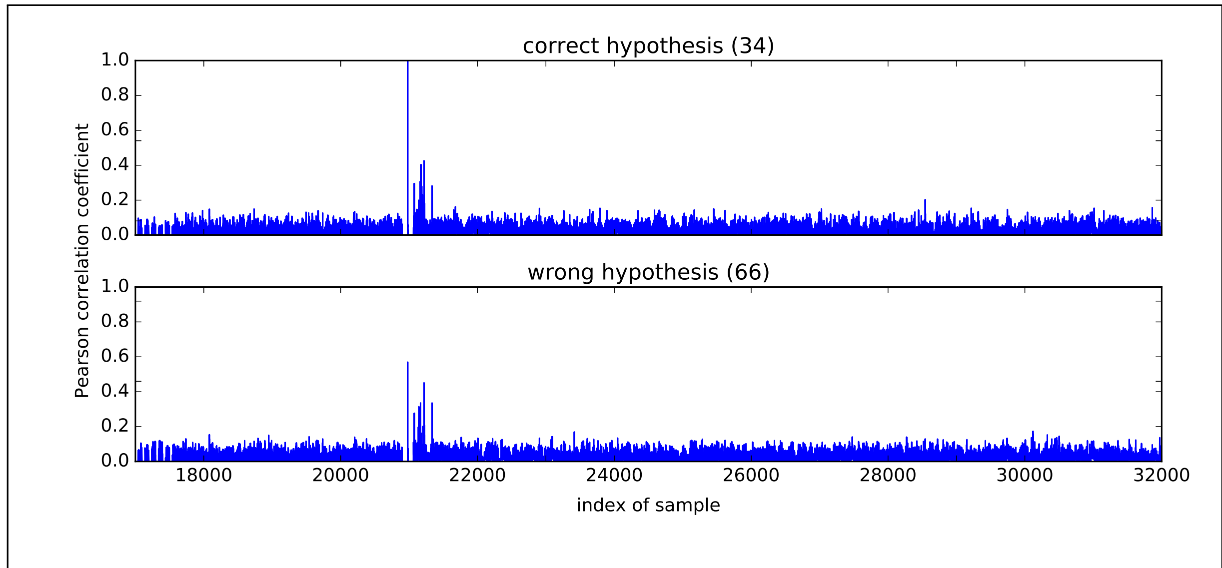


Figure 5.4: Correlation values of correct and wrong key hypothesis with simulated power traces over time in the 8-bit HW leakage model.

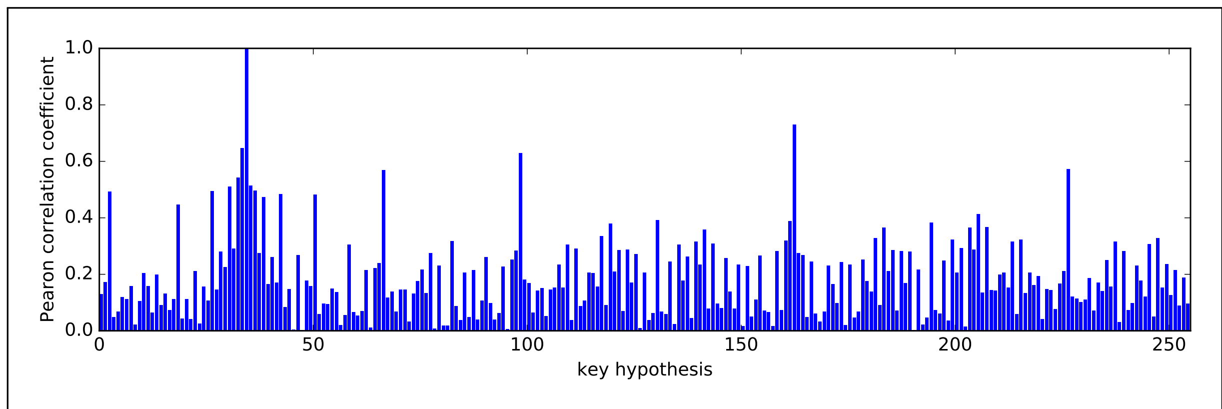


Figure 5.5: Maximum correlation of all possible key hypotheses in the 8-bit HW leakage model. The correct sub-key (34) can be detected easily

result of our targeted computations which, consequently, also lead to smaller correlations. The correct hypothesis results in much higher correlation values than the other key candidate. Since our simulated traces contain no noise at all, the correlation of the correct key hypothesis is exactly 1.0.

Figure 5.5 illustrates the maximum correlation values of each possible key hypothesis for the same computation. The correct hypothesis results in a correlation of 1.0, which is significantly higher than any other correlation, which allows the recovery of the least significant byte of $\delta^{(1)}$. Note that the correlation values when using physically measured traces will be smaller than 1.0 due to noise, such that the detection of the correct sub-key will be harder and in consequence may require more traces. Figure 5.5 also shows that the correlation values are small (< 0.4) for most of the key candidates and only higher for 16 key candidates in this experiment. Thus, even if the noise is too high to successfully require the correct sub-key, it still allows a drastic reduction in the search space which can then be easily iterated to find the correct key.

The previous experiment showed that the DPA is able to recover a single key byte. Next, we wanted to evaluate the success probability of the entire attack, which includes 9 DPA on 32-bit operation, i.e., 36 DPA when using the 8-bit HW leakage model. The success rates of the single DPA are not independent of each other due to two reasons: Firstly, when attacking addition, the higher significant bytes can only be

recovered reliably if the lower significant byte key guesses are correct, since only then can we correctly calculate the carry bits. Secondly, the attacked operations depend on each other, e.g., DPA 2 requires that DPA 1 successfully recovered $\delta^{(1)}$. Thus, it is certain that the success rate of the entire attack is significantly smaller than for each individual DPA.

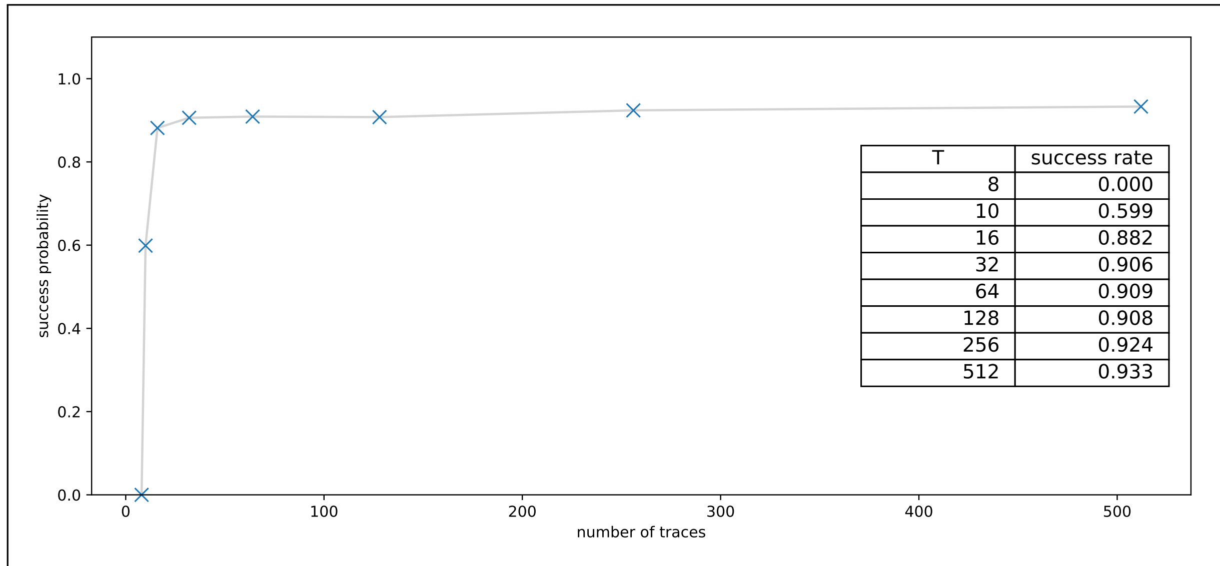


Figure 5.6: Success rate of the full DPA key recovery attack on the vulnerable PRNG in the 8-bit HW leakage model

Figure 5.6 shows the results of the full key recovery attack using different numbers of traces. To produce reliable results, we repeated the experiment for each number of traces 3000 times. When using only $T = 8$ traces per experiment, the recovery failed for 100% of our trials, whereas using $T = 10$ already resulted in a success probability of almost 60%. This further increased to 93.3% for $T = 512$. However, as already mentioned, the DPA on *AND* operations always failed to recover the key if the subkey is equal to zero. Thus, for all experiments in which one of the attacked bytes in DPA 4, 5, 6, and 7 equals zero, the attack fails. Since the attacked values are approximately uniformly distributed, the probability for this is $\frac{4 \cdot 4}{256} = 6.25\%$. Thus, we conjecture that the best achievable success rate in this setup is 93.75%, no matter how many traces are used. Note that this problem can be mitigated by an adversary, as discussed earlier. However, in our experiments we did not implement that optimization.

Note that the actual numbers found in these experiments can only provide a lower bound, since our traces contain no noise at all. In real-world measurements, the required number of traces is significantly larger depending on the amount of noise.

5.4.2 32-Bit Hamming Weight Leakage Model

Since an 8-bit leakage is not very likely for actual implementations of SHA2, we additionally performed experiments which are using the more realistic 32-bit HW leakage model and partial DPA as introduced by Belaïd et al. [BBD+13].

Since partial DPA require more traces than normal DPA and the run-time of our analysis drastically increases with the number of traces, we evaluated the recovery of single 32-bit words, instead of the full 256-bit IV. Figure 5.7 illustrates how the number of traces affects the maximum Pearson correlation coefficient found for each key hypothesis, when the least significant byte of $\delta^{(1)}$ is attacked without any knowledge about the three more significant bytes. For a relatively small number of traces (<100), the correlation values appear random and do not allow inference of the correct key hypothesis. If 200 or more traces are used, the correlation value of the correct key hypothesis is slightly below 0.5, which is

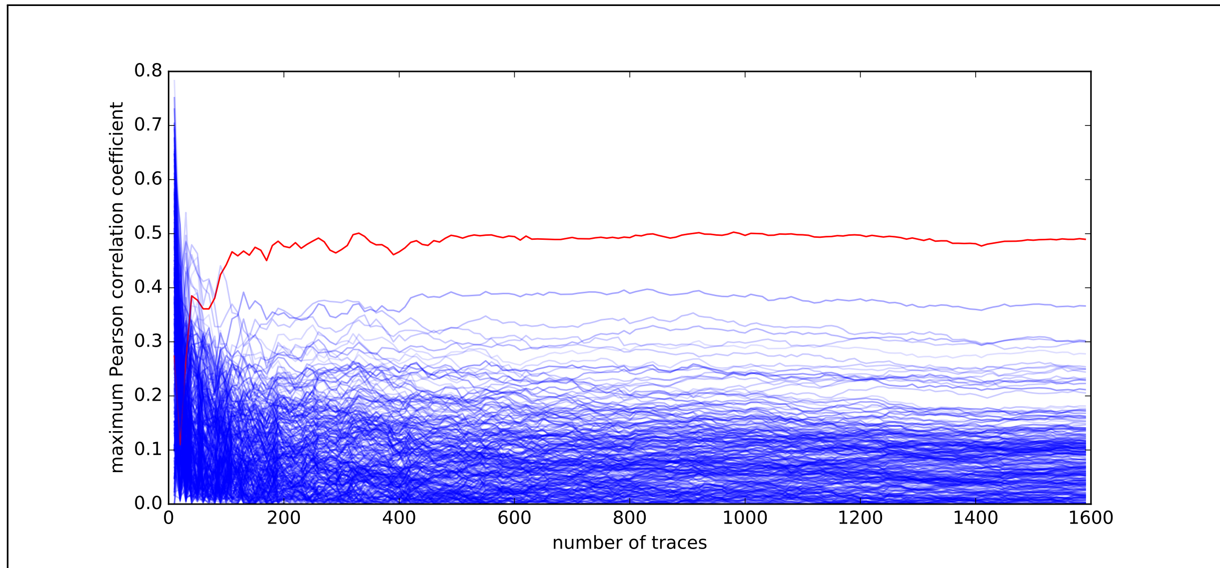


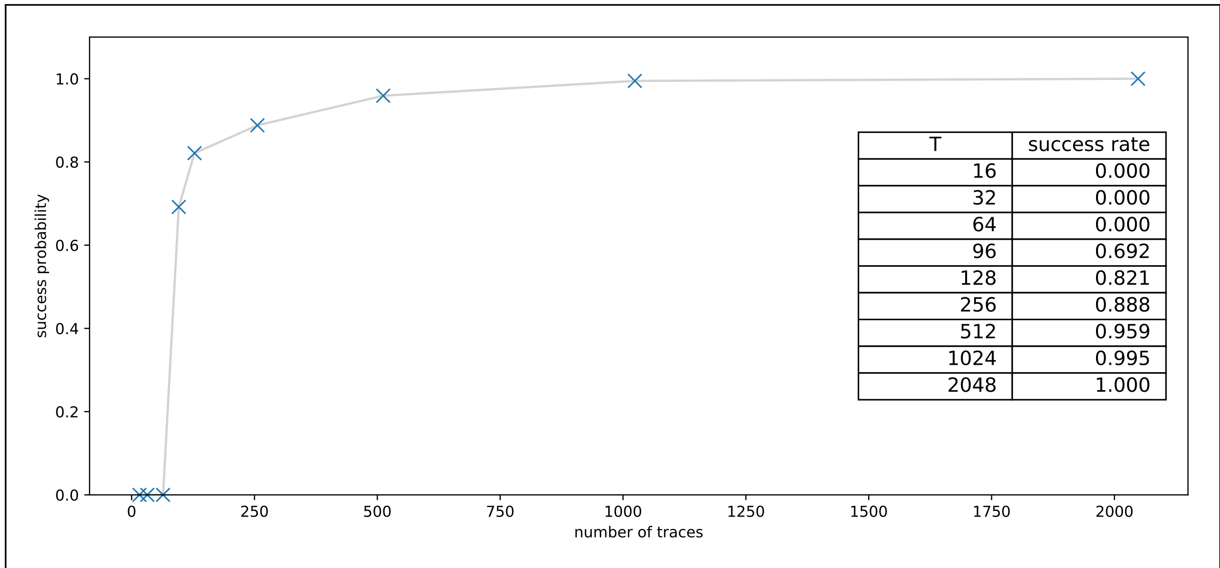
Figure 5.7: Maximum partial correlation values of correct hypothesis (red) and wrong hypotheses (blue) when attacking modular addition in the 32-bit HW leakage model

significantly larger than for any other key hypothesis. Note that there is a small number of hypotheses that also have moderate correlations in the range of 0.3 and 0.4. However, the majority of key hypothesis result in low correlations (<0.3) and can, thus, be be ruled out. At some point, further increasing the number of traces results in very little change in the correlation values. Note that in this experiment we attacked the least significant byte only, which is the hardest one. Then, when attacking the more significant ones, we can utilize the knowledge of the correct hamming weight of the less significant ones, which improves the prediction of the hamming weight. This leads to much higher correlation values and, thus, a significantly higher success rate of recovering the key bytes.

We performed another two experiments to evaluate how these correlation values translate into actual success rates of our DPA. Figure 5.8 shows the results for the success rate of recovering $\delta^{(1)}$, i.e., 4 partial DPA on modular 32-bit addition to recover each byte separately. For small values of $T \in \{16, 32, 64\}$, the success rate was 0%, i.e., not a single trial successfully recovered all four bytes of $\delta^{(1)}$. When increasing the number of traces, the success rate increases up to 100% for $T = 2048$. Thus, for a 32-bit addition operation a reliable recovery is possible if enough traces are available. However, the results again just provide an optimistic lower bound, i.e., the actual number of required traces in a physical setup will be higher depending on the amount of noise.

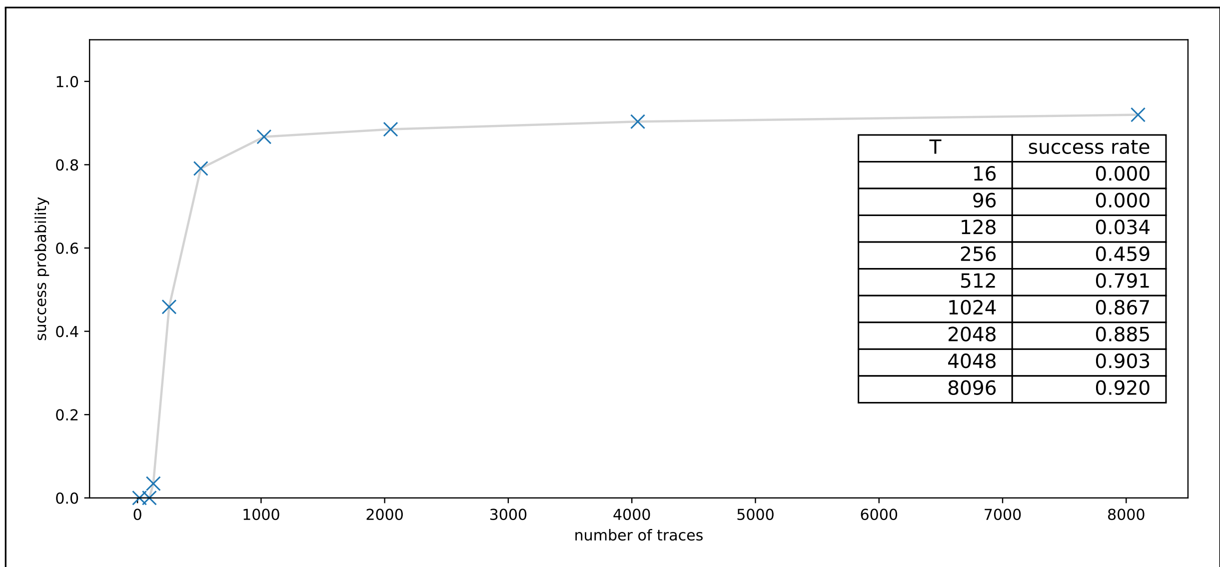
The results of the same experiment for a 32-bit bitwise AND operation are plotted in Figure 5.9. It evaluates the success rate of the first DPA on bitwise AND, which is DPA 4 under the assumption that DPA 1 and DPA 2 were successful. This assumption is required, because DPA 4 only works if we know $\mathbf{E}^{(1)}$, which is computed from $D^{(0)}$ (recovered in DPA 2) and $\mathbf{T1}^{(1)}$ (computable from $\mathbf{W_0}$ and $\delta^{(1)}$, which was recovered in DPA 1). To simulate this, we included the values for $\delta^{(1)}$ and $D^{(0)}$ in the simulation results, such that the analysis script can use them. The differences compared to the addition operation are twofold: Firstly, the success rate stays at around 0% for $T \leq 128$ and then increases a lot slower, i.e., more traces are required for a successful DPA. Secondly, the success rate does not increase to 100% for reasons already discussed.

Our experiments showed that the proposed attack does allow recovery of the IV to the second compression function, which can then be used to recover the pseudorandom values for an arbitrary index i. If these pseudorandom values are used as W-OTS+ secret keys, this entirely compromises the security of the signature scheme. However, it is still unclear if this attack can be adapted to attack the PRNG recommended in the XMSS Internet Draft [HBGM17], which is discussed in the next section.



T	success rate
16	0.000
32	0.000
64	0.000
96	0.692
128	0.821
256	0.888
512	0.959
1024	0.995
2048	1.000

Figure 5.8: Success rate of recovering a single 32-bit value using a DPA on modular addition in the 32-bit HW leakage model



T	success rate
16	0.000
96	0.000
128	0.034
256	0.459
512	0.791
1024	0.867
2048	0.885
4048	0.903
8096	0.920

Figure 5.9: Success rate of recovering a single 32-bit value using a DPA on bitwise AND in the 32-bit HW leakage model

5.5 Applicability to Practical PRNGs

We have shown that under certain conditions SHA2 PRNG are vulnerable to DPA attacks. We proposed a new PRNG, which is cryptographically secure, but is relatively easy to attack using power-based side-channel attacks. We then evaluated in our experiments how likely the attack is to succeed. This proved that if a weak PRNG is chosen, an implementation can be broken. Since the XMSS Internet Draft [HBGM17] leaves the choice of the PRNG to the implementer, we want to emphasize that this is the key point when creating implementations of XMSS which need to resist side-channel attacks. The Internet Draft suggests that the use of a PRNG does not affect the scheme security as long as the PRNG provides similar cryptographic security. However, we emphasize that while this might be the case when analyzing XMSS as a black box, it is not true when side-channel attacks are considered.

Thus, it is imperative to look at less artificial PRNG and evaluate whether our attack can be adapted to break them as well. Both Hülsing et al. [HBGM17] and Buchmann et al. [BDH11] provide the same suggestion on how to implement the PRNG for W-OTS+, XMSS and XMSS^{MT}. Although already introduced, we repeat them here for convenience. The PRNG suggested for the SHA2 function family and $n = 256$ is

$$PRNG_{XMSS}(SEED, i) = \texttt{SHA-256}(\texttt{toByte}(3, 32)||SEED||\texttt{toByte}(i, 32))$$

and $\texttt{toByte}(i, 32)$ is the big-endian 32-byte (256 bit) encoding of i. We already found that the attack is applicable if the values of i were random and distributed over the full range of possible values, i.e., $0 <= i < 2^{256}$. Thus, it is essential to evaluate for which values it is called in practice.

W-OTS+ uses the PRNG to generate $\ell \cdot n$ bits from a n-bit seed with s

$$sk_{W-OTS+,i} = PRNG_{XMSS}(SEED_{W-OTS+,j}, i) \quad 0 \leq i < \ell$$

ℓ can only have the values $\ell = 67$ for $n = 256$ and $\ell = 131$ for $n = 512$.

When using W-OTS+ inside of XMSS the $SEED_{W-OTS+,j}$ are generated with

$$SEED_{W-OTS+,j} = PRNG_{XMSS}(SEED_{XMSS}, j) \quad 0 \leq j < 2^h$$

Thus, the range of the indices i is determined by the parameter h. In the current version of the Internet Draft [HBGM17], the maximum value of h is 20, i.e., $0 \leq i < 2^{20}$.

Consequently, since i is encoded big-endian byte order, the message block passed to the second evaluation of the compression function, which is $\texttt{toByte}(i, 32)$, only changes in bytes 29, 30, and 31, which are all part of $\mathbf{W_7}$. Bytes $0, ..., 28$ are always zero. Bytes $32, ..., 63$ only contain padding and the length of the full message, which is always constant. Therefore, the DPA on the first iteration of the compression function aiming to recover $A^{(0)}, ..., H^{(0)}$ is not possible due to the fixed words $W_0 = 0$ and $W_1 = 0$. However, since they are fixed for all evaluations, it is possible to attack later iterations. For example, if $W_t = 0$ for all $t < \alpha$, we can still attack $A^{(\alpha)}, ..., H^{(\alpha)}$ since they are the same for all i.

Note that our attack used exactly two 32-bit words, $\mathbf{W_0}$ and $\mathbf{W_1}$. Thus, if $\alpha \leq 14$ and the words $\mathbf{W}_\alpha$ and $\mathbf{W}_{\alpha+1}$ contain enough entropy, the attack does work. For XMSS it holds that $W_\alpha = 0$ for all $\alpha < 15$ and only $\mathbf{W_7}$ is variable. W_8 contains the fixed padding pattern `0x80000000`. Consequently, this implies that our attack does not work since $\mathbf{W_7}$ is variable and W_8 is fixed. Later iterations cannot be used to mount a DPA attack since the values of $A, ..., H$ then depend on the index i.

Thus, we conclude that the PRNG recommended by the XMSS Internet Draft [HBGM17] is vulnerable to the attack presented in general, but the parameters of XMSS prevent a successful attack. If h were to be much larger (≥ 64), the attack would work. However, this is not a practical parameter choice, since the public key computation is then infeasible.

Nonetheless, the attack could still apply to the hypertree variant XMSS^{MT}, since the overall structure is much larger. However, the XMSS^{MT} uses an additional layer of PRNG in which the $SEED_{XMSS}$ is computed for the x-th tree at layer y by evaluating

$$seed_{XMSS,x,y} = PRNG_{XMSS}(PRNG_{XMSS}(SEED, \texttt{toByte}(y, 32)), \texttt{toByte}(x, 32)) \quad 0 \leq y < d, 0 \leq x < 2^{h/d}$$

Since the values of x and y are very small, this method of producing seeds is also not vulnerable, which makes XMSS^{MT} resistant to our attack.

6 Fault Attack on XMSS^{MT}

The second physical attack on XMSS, which is proposed and implemented as a part of this thesis, is a fault attack. It attacks the hypertree variant of XMSS and exploits the one-time nature of the W-OTS+ signatures used to sign the roots of the lower XMSS trees. Our attack is based on a very recent thesis by Genet on fault attacks against hash-based signature schemes and was orginally proposed and implemented for SPHINCS [Gen17]. Section 6.1 introduces the work upon which the attack is based. Section 6.2 describes the modifications and improvements we implemented to apply the attack to XMSS^{MT}. Our implementation and simulation of the attack are described in Section 6.3. We present the experimental results in Section 6.4 and propose several countermeasures in Section 6.5 that can be used to prevent the attack entirely. The source code that can be used to simulate this attack can be found at `https://github.com/mkannwischer/xmss-fault-attack` and is published under a 2-clause BSD license.

6.1 Fault Attack on SPHINCS

As introduced in Section 3.3.2 SPHINCS uses a hypertree with the few-time signature scheme HORST to sign the messages and W-OTS+ to sign the roots of the lower Merkle trees. The hash of the message to be signed is used to "randomly" select a HORST key pair in the giant hypertree structure. A SPHINCS signature consists of the HORST signature, d W-OTS+ signatures and their corresponding authentication paths in the Merkle trees. Since for each different message, a different HORST key pair will be used, all W-OTS+ signatures and the authentication paths need to be recomputed every time. This is feasible due to the hypertree structure, but presents a severe vulnerability when faults attacks are considered.

In Section 3.1.2 it was already described that if an adversary manages to force different messages to be signed with the same W-OTS+ secret key, the security degrades and eventually vanishes [BH16]. This can be turned into an attack exploiting that SPHINCS creates multiple W-OTS+ signatures using the same secret key. In normal operation the signatures are generated for the same messages (roots of the Merkle trees) every time, but if natural or malicious faults occur, this is no longer guaranteed.

Given a security parameter n (hash length), the total hypertree height h, the number of layers d and the Winternitz parameter w, the attack of Genet [Gen17] works as follows: Suppose we use SPHINCS to sign the same message q times, but during each signature generation a fault occurs while the computation of the subtree at layer i ($0 \leq i < d-1$, i.e., not the top layer). This leads to a different root of the corresponding Merkle tree, and, thus a different W-OTS+ signature σ_{i+1}. By combining the chain values contained in each σ_{i+1} the adversary is able to recover chain values for different b_i's. Eventually, the adversary recovered enough values to be able to create an existential W-OTS+ forgery. Then, the adversary can replace all trees below by trees computed from secret W-OTS+ and HORST keys chosen by himself. This enables the adversary to create universal forgeries.

Genet practically implements this attack for SPHINCS-256 on an Arduino Due board and uses voltage glitching to inject faults during the Merkle tree computations. He successfully creates a practical forgery for SPHINCS using $q = 20$ faulty signatures. However, his results suggest that for a success rate of nearly 100% about 100 faulty signatures are required.

This attack does not directly apply to XMSS, because a main difference is that SPHINCS uses a pseudorandom leaf selection, while XMSS uses the leaves in strict order. While signing the same message multiple times with SPHINCS will result in the same leaf selection and the same authentication path, this is not the case for XMSS. When using XMSS, the authentication path will change for every new message and, thus, the attack is not directly applicable. However, just a slight modification would enable an attack on XMSS also, as described in the next section.

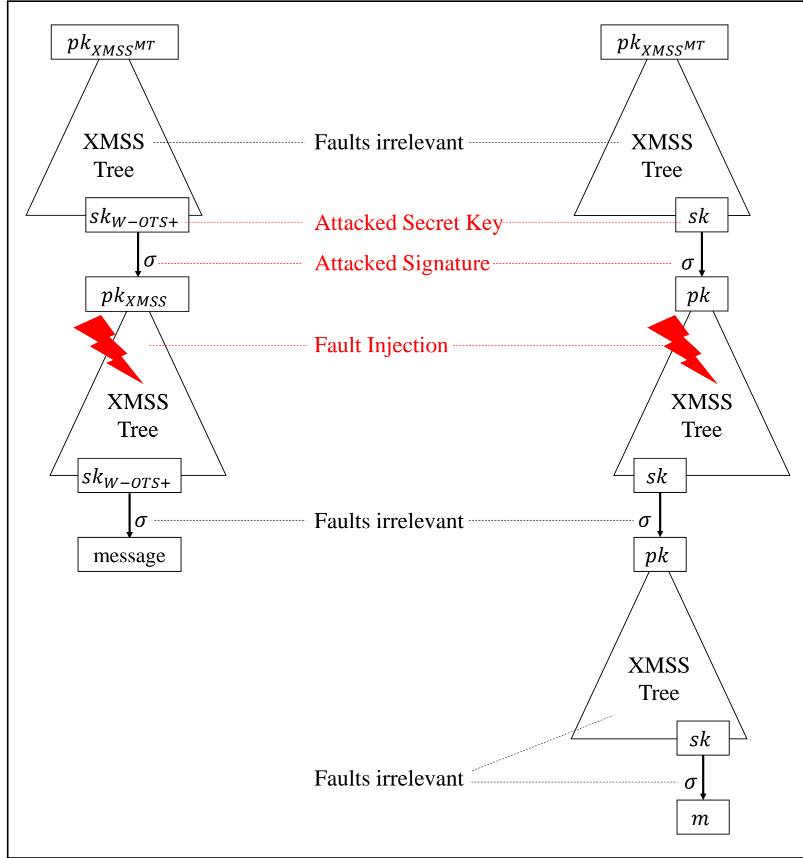


Figure 6.1: Proposed fault attack on XMSS^{MT} for $d = 2$ (left) and $d = 3$ (right)

6.2 Attack Design and Adversary Model

Due to the similarity of SPHINCS and XMSS^{MT}, the attack proposed by Genet [Gen17] can be adapted to attack XMSS^{MT} as well. Figure 6.1 shows the general attack scenario for two XMSS^{MT} hypertrees ($d = 2$ and $d = 3$). Straightforward implementations of XMSS^{MT} will recompute each layer for each signature and, thus, create d W-OTS+ signatures per signature generation. By injecting faults at layer $d-2$, we can force the cryptographic device to sign different values with the W-OTS+ secret key at layer $d-1$. However, different from SPHINCS, for each consecutive signature generation a new W-OTS+ key pair at the bottom layer will be used, such that the authentication path to the XMSS^{MT} public key will be different for every signature. Since each key pair at layer $d-1$ is used $2^{h-(h/d)}$ times, we can still attack it but $2^{h-(h/d)}$ is the maximum number of faulty signatures than can be used to recover it. Therefore, different from Genet's attack, we do target the top W-OTS+ signature and not an arbitrary one to increase the number of obtainable faulty signatures. Additionally, since XMSS uses the leaves on the bottom layer in strict order, we do not require that each faulty signature is generated for the same message as it is required for the attack on SPHINCS.

We extend the work of Genet by the following:

- We adapt the attack for the use of XMSS and provide an extensive description on how this attack can be implemented.
- We improve the attack to require less faulty signatures by attempting the tree grafting p times, which will be explained in more detail later
- We simulate the attack to experimentally determine the effect of the security parameter n and the parameter p.

- We propose countermeasures specific to XMSS and show that it is easy to completely prevent this attack in XMSS, while it is infeasible to apply the same countermeasure to SPHINCS.

Before describing our attack, we outline our *adversary model*:

- The goal of an adversary is to be able to create signatures for arbitrary messages (i.e., universal forgeries) that can be verified with a given XMSS^{MT} public key (consisting of the root and a public seed used to generate the bit masks).
- The adversary is able to get a limited number of faulty and correct signatures.
- The adversary can control where and when the fault occurs. He can ensure that during the computation of the XMSS tree at layer $d-2$, a fault occurs in the registers holding the intermediate values. Which computation inside the tree is faulty does not matter, since even small errors will propagate through the hash tree resulting in a random root node due to the properties of the hash function.
- The adversary can ensure that the computation of the W-OTS+ signature at layer $d-1$ is computed correctly, i.e., no fault occurs. Faults during the other W-OTS+ computations may occur and are irrelevant for this attack.

For the following explanation we assume that at the start of the attack a fresh XMSS tree at layer $d-2$ is used, i.e., $s \equiv 0\ (mod\ 2^{h-(h/d)})$. If this assumption does not hold in practice the adversary has two options: He either creates signatures until the assumption holds or he tries to perform the attack anyway which further limits the maximum number of faulty signatures he can generate before all leaves of this tree are used up and XMSS^{MT} switches to the next tree.

Algorithm 6.1 Proposed fault attack on XMSS^{MT}

```
 1: Input: pk_{XMSS^MT}, p
 2: σ_valid ← sign(m)
 3: sk^partial_{W-OTS+} ← null
 4: for i=0; i<2^{h-(h/d)} - 1; i++ do
 5:     σ_fault ← faulty_sign(m')
 6:     sk^partial_{W-OTS+} ← extract_and_merge(σ_fault, sk^partial_{W-OTS+}, σ_valid)      ▷ Algorithm 6.2
 7:     for j = 0; j < p; j++ do                                                            ▷ Algorithm 6.3
 8:         (isForgeable, σ_forge, sk_seed) ← forge_signature(sk^partial_{W-OTS+}, σ_valid, m'', pk_{XMSS^MT})
 9:         if isForgeable = True then
10:             return (sk_seed, σ_forge)
11:         end if
12:     end for
13: end for
14: return Fail
```

Algorithm 6.1 outlines our attack. Details (sub-routines) will be described in Section 6.3. Firstly, a valid signature for an arbitrary message is created (line 2). The actual message does not matter, thus, it is sufficient to eavesdrop a message-signature pair. This message is required to recover the W-OTS+ public key at layer $d-2$, which we denote $pk^{d-2}_{W\text{-}OTS+}$. The adversary then repeatedly creates faulty signatures until either enough information is recovered or the maximum number is reached, i.e., XMSS^{MT} switches to next tree at layer $d-2$. With each faulty signature (line 5), the adversary retrieves additional information about the secret key which is merged to the partial W-OTS+ secret key (line 6). The message used in each iteration does not matter. The adversary not even needs to know it. At some point the adversary has enough information to forge a signature. He doesn't need all secret key parts to

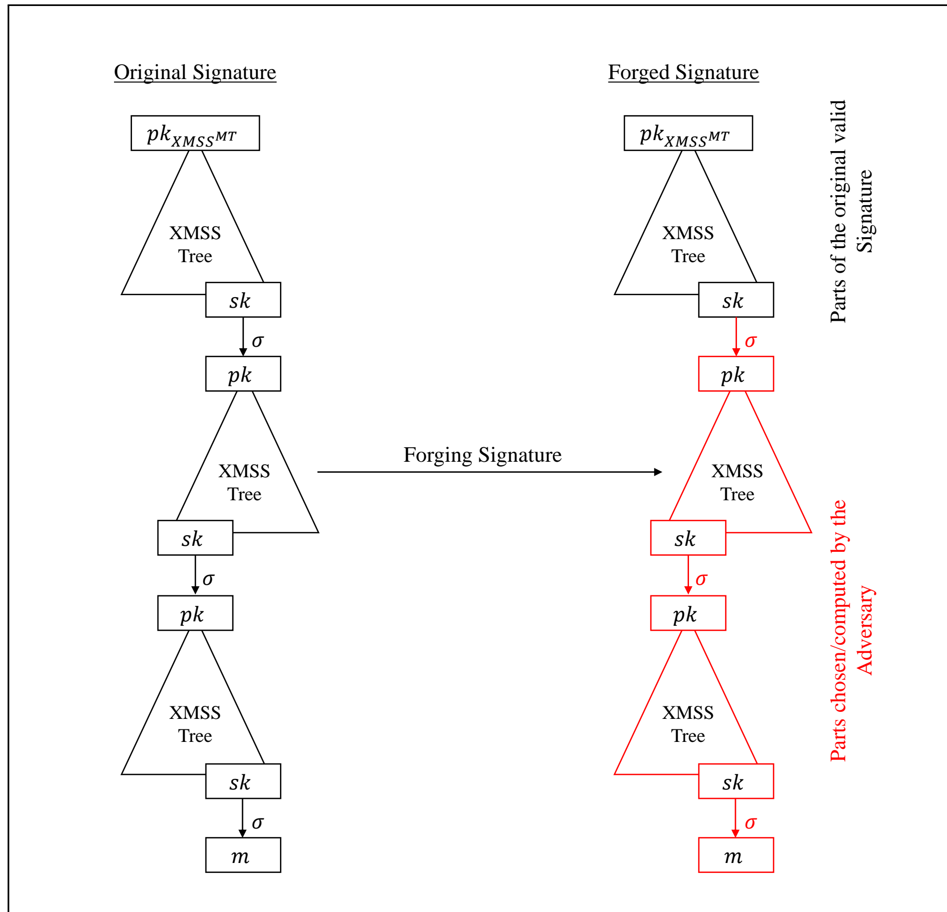


Figure 6.2: Forging an XMSS^{MT} signature by replacing the lower XMSS trees and signing the root with the recovered (partial) secret W-OTS+ key

forge an W-OTS+ signature. As long as he has $c^j(x_i, r)$ and $j <= b_i$ an W-OTS+ signature can be forged (see Section 3.1.3). This results in much fewer required faulty signatures to create a forgery. Therefore, the adversary tries to forge a signature after each signature generation to evaluate how many signatures will be required. Additionally, given a set of recovered values $sk^{partial}_{W\text{-}OTS+}$ it might be possible to sign some messages, while it fails for others. To study this, the adversary attempts p forgeries for different seed values in each iteration. The higher the parameter p is chosen, the less faulty signatures are required.

Note that since the forging is a relatively expensive operation, a real adversary will follow a different strategy instead: He will collect as many faulty signatures as possible and then try to forge the signature later. However, for our experiments we wanted to determine the number of required signatures.

Once the adversary has successfully forged one signature, he can use the exact same key to sign arbitrary messages, i.e., an universal forgery is as hard to create as an existential forgery. Figure 6.2 shows how the forgery works for a hypertree with $d = 3$. The adversary entirely replaces the XMSS trees at all layers below the top layer. The root of the XMSS tree at layer $d-2$, which the adversary computed from the chosen secret keys, is signed using the recovered (partial) W-OTS+ secret key. Combining this with the authentication path from the initial valid signature, a valid signature for an arbitrary message can be forged without the possession of the actual entire secret key.

6.3 Implementation

We created a proof-of-concept simulation of our attack in C based on the reference implementation of the Internet Draft [GB17]. The source code of our implementation is available at `https://github.com/mkannwischer/xmss-fault-attack`. It includes scripts that can be used to reproduce the results presented in this thesis.

Algorithm 6.2 Fault attack: `extract_and_merge`

1: Input: $\sigma, sk^{partial}_{W\text{-}OTS+}, \sigma_{valid}$
2: Extract W-OTS+ public key $pk^{d-1}_{W\text{-}OTS+}$ at layer $d-1$ from σ_{valid}
3: **if** $sk^{partial}_{W\text{-}OTS+} =$ `null` **then** ▷ Initialize if initial call of this procedure
4:     **for** i=0; i<ℓ; i++ **do**
5:         $sk^{partial}_{W\text{-}OTS+}[i] \leftarrow [w, pk^{d-1}_{W\text{-}OTS+}[i]]$
6:     **end for**
7: **end if**
8: Extract $\sigma^{d-1}_{W\text{-}OTS+}$ from σ
9: **for** i=0; i<ℓ; i++ **do**
10:     **for** j=0; j<w; j++ **do**
11:         **if** $j >= sk^{partial}_{W\text{-}OTS+}[i][0]$ **then**
12:             **break** ▷ already have a better value for this chain
13:         **end if**
14:         **if** $c^{w-1-j}(\sigma^{d-1}_{W\text{-}OTS+}[i], r) = pk^{d-1}_{W\text{-}OTS+}$ **then**
15:             $sk^{partial}_{W\text{-}OTS+}[i][0] \leftarrow j$ ▷ found better chain value
16:             $sk^{partial}_{W\text{-}OTS+}[i][1] \leftarrow \sigma^{d-1}_{W\text{-}OTS+}[i]$
17:         **end if**
18:     **end for**
19: **end for**
20: **return** $sk^{partial}_{W\text{-}OTS+}$

Algorithm 6.2 presents the `extract_and_merge` step of our attack which is used to merge partial information about our target secret W-OTS+ key. It takes an XMSS^{MT} signature σ (valid or faulty), the already known values $sk^{partial}_{W\text{-}OTS+}$ and a valid XMSS^{MT} signature σ_{valid}, which is the same for all iterations. The adversary first extracts the W-OTS+ public key from the valid signature (line 1). If this is the first call of the routine, $sk^{partial}_{W\text{-}OTS+}$ is initialized (lines 3-7): For each chain, the adversary creates a 2-element array where the first element represents the index k of the value in the chain (16: public key, 0: secret key) and the second element is the chain value, i.e., $(c^k(x_i, r))$. Initially, this is set to the public key.

The routine then extracts the targeted W-OTS+ signature $\sigma^{d-1}_{W\text{-}OTS+}$ from σ. Since it is part of the XMSS^{MT} signature, this is only a copy operation. The adversary then checks for each chain i if the chain value does provide some information which he does not already know. This is done by hashing the new value $w-1-j$ times and comparing it to the public key part. If they are equal, the adversary found $c^j(x_i, r)$. If j is smaller than the index of the already known value, the adversary recovered a new chain value and replaces it in $sk^{partial}_{W\text{-}OTS+}$.

Algorithm 6.3 describes the routine that is used to find out if forging an XMSS^{MT} signature is possible and provide a forgery if possible. The adversary first chooses a random seed (line 2) which is used to create a temporary XMSS^{MT} key pair (line 3). This key pair is used to sign the message for which the adversary wants to forge a signature (line 5). It is important to note that this created signature as it is, cannot be verified using the given public key, but replacing certain parts will result in a valid signature.

The adversary extracts the root of the XMSS tree at layer $d-2$, which is done by running part of the XMSS^{MT} verification algorithm which reconstructs the authentication path. The adversary now needs

Algorithm 6.3 Fault attack: `forge_signature`

1: Input: $sk^{partial}_{W\text{-}OTS+}$, σ_{valid}, m, $pk_{XMSS\hat{}MT}$
2: Choose random seed sk_seed
3: Create a new XMSS^{MT} key pair using sk_seed and the public seed from $pk_{XMSS^{MT}}$
4: $\rightarrow \widehat{sk}_{XMSS^{MT}}, \widehat{pk}_{XMSS^{MT}}$ ▷ Note: $\widehat{pk}_{XMSS^{MT}} \neq pk_{XMSS^{MT}}$
5: $\sigma_{forge} \leftarrow sign_{XMSS^{MT}}(m, \widehat{sk}_{XMSS^{MT}})$ ▷ Note: verify($m, \sigma_{forge}, pk_{XMSS\hat{}MT}$) = `False`
6: Extract pk^{d-2}_{XMSS} from σ_{forge}
7: Calculate $b_0, ..., b_\ell$ from pk^{d-2}_{XMSS} ▷ try to sign the root with the recovered partial secret key
8: Initialize $\sigma_{W\text{-}OTS+}$
9: **for** i=0; i<ℓ; i++ **do**
10: $k \leftarrow sk^{partial}_{W\text{-}OTS+}[i][0]$ ▷ index of our known chain value
11: $\hat{x} \leftarrow sk^{partial}_{W\text{-}OTS+}[i][1]$ ▷ known chain value
12: **if** $b_i < k$ **then**
13: **return** (`False`, `null`, `null`) ▷ forging failed
14: **else**
15: $\sigma_{W\text{-}OTS+}[i] \leftarrow c^{b_i - k}(\hat{x}, r)$
16: **end if**
17: **end for**
18: Replace W-OTS+ signature at layer d-1 in σ_{forge} with $\sigma_{W\text{-}OTS+}$
19: Copy authentication path for layer d-1 from σ_{valid} to σ_{forge}
20: **return** (`True`, σ_{forge}, sk_seed)

to forge an W-OTS+ signature for this value using the recovered (partial) secret key. Therefore, he calculates the W-OTS+ blocks b_i including the checksum. If for each b_i the adversary has a chain value $c^j(x_i, r)$ for which $j <= b_i$, the forgery succeeds (line 12). The forgery is then straightforwardly implemented by applying the chaining function $b_i - j$ times (line 15).

In the end the adversary needs to replace the W-OTS+ signature in σ_{forge} with the forged W-OTS+ signature. Additionally, the authentication path for the corresponding W-OTS+ public key needs to be copied from the initial valid XMSS^{MT} signature σ_{valid}. The returned signature σ_{forge} is valid when verified with the given public key $pk_{XMSS^{MT}}$. Additionally, the used seed together with a valid signature (either the initial or the forged) can be used to create more signatures for arbitrary messages.

Our attack simulation can be executed as follows:

```
./attack n h d p [--silent]
```

The parameters are as follows

- **n**: security parameter / hash size in bytes, i.e., 32 or 64
- **h**: total height of the XMSS^{MT} tree
- **d**: number of layers ($h \equiv 0 \mod d$ is required)
- **p**: number of different seeds tried per iteration
- **--silent**: turns off logging, only outputs result

The simulation outputs if the attack was successful and how many signatures were required.

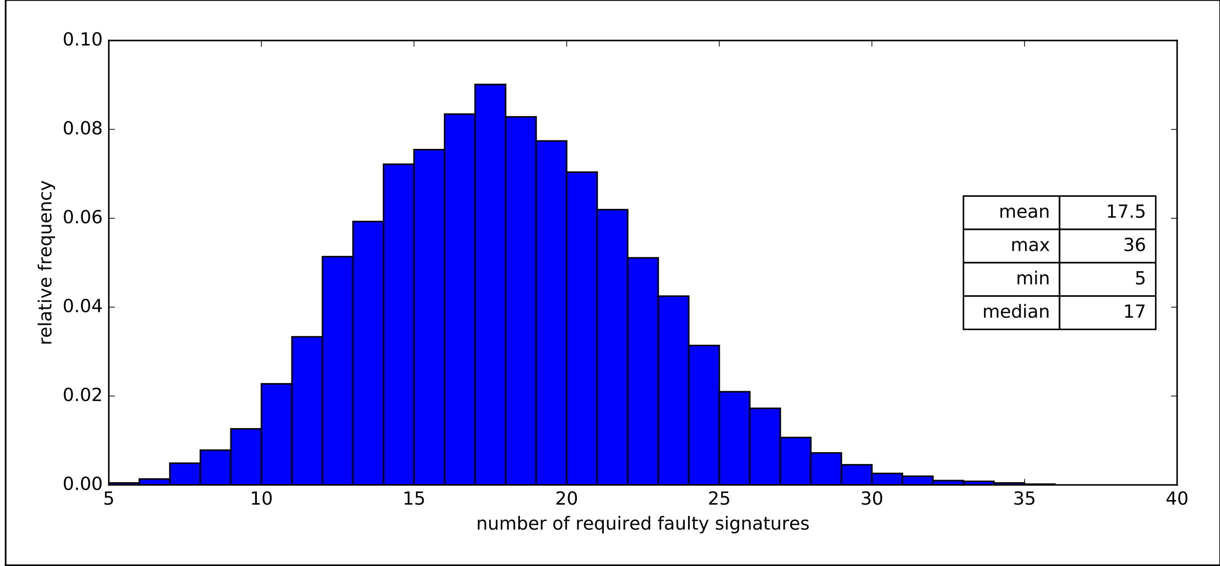


Figure 6.3: Experimental results for the fault attack simulation for $n = 256$ and $p = 1$

The objective of our experiments is threefold. Firstly, we want to create a proof-of-concept to show that the attack actually works for XMSS. Secondly, we want to determine the number of required faulty signatures for similar parameters as in [Gen17]. Thirdly, we simulate how the security parameter n and the number of forgery trials p affects this number. n can be 256 or 512 for XMSS [HBGM17]. p, the number of forgery trials, can be chosen by the adversary. The higher p, the more likely an attack is to succeed. However, the runtime of the experiment linearly increases with the value of p.

Figure 6.3 shows the number of required faulty signatures for $n = 256$. We used $h = 8$ and $d = 4$, such that the number of faulty signatures is limited to $2^6 - 1 = 63$. This limit was never reached in our experiments, i.e., in all experiments the forgery succeeded. To produce statistically representative results, we repeated the experiment 10,000 times. The maximum number of faulty signatures required was 36 and the minimum was 5. However, more extreme values are possible in theory with lower probabilities. The median number of required faulty signatures was 17, i.e., 17 faulty signatures are enough for the attack to succeed in over 50% of cases.

Note that 17 is a lot lower than the results of Genet [Gen17], which finds that around 30 faulty signatures are required for the same parameters and a 50% success probability. This is not due to an improvement we implemented, but due to the experimental setup. Genet determined how many signatures were required to forge a signature for an arbitrary message. However, we repeatedly try to forge a signature for a different XMSS public key generated from a new seed in every iteration until the forgery succeeds. Consequently, this leads to a bias towards lower numbers of required signatures, since in each iteration there is a certain chance for the attack to succeed. This effect will be further studied later by altering the parameter p.

Genet derives a formula which can be used to determine the success probability of the attack given the Winternitz parameter w, the number of hash chains ℓ (calculated from n and w) and the number of faulty signatures [Gen17]:

$$Pr[success] = \frac{1}{w^{\ell}} \left(\sum_{x=0}^{w-1} \left(1 - \left(\frac{w - (x+1)}{w} \right)^{q+1} \right) \right)^{\ell}$$

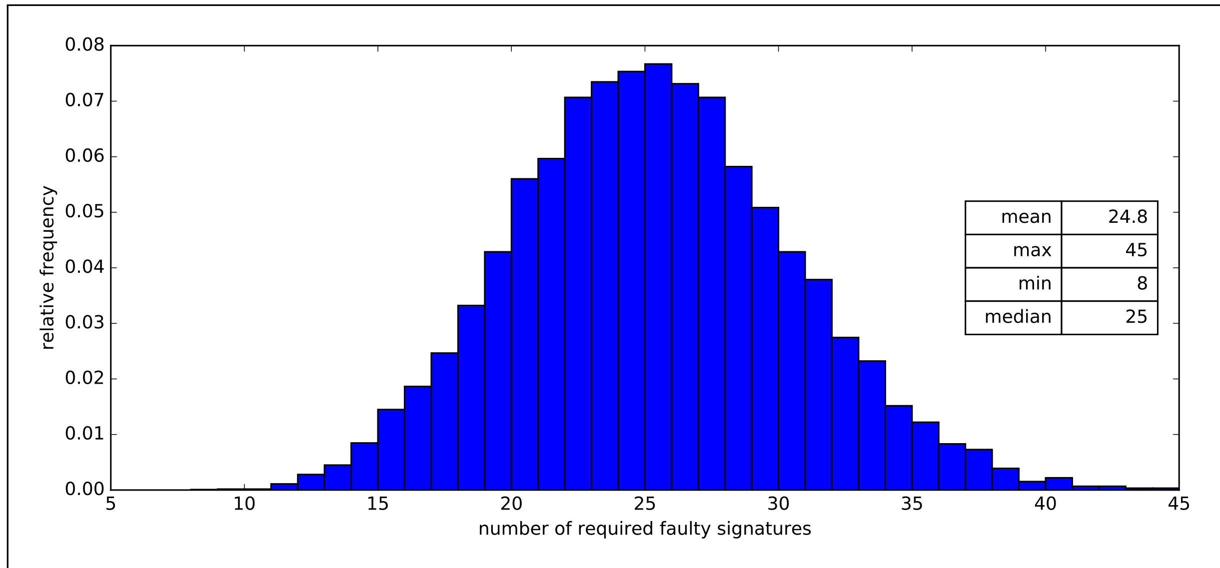


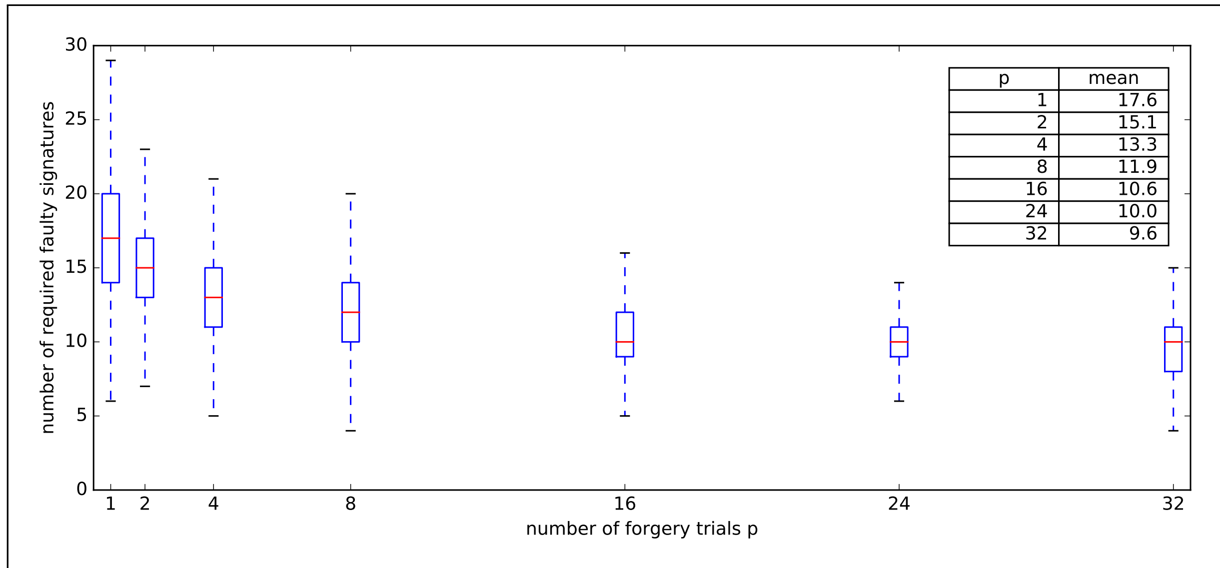
Figure 6.4: Experimental results for the fault attack simulation for $n = 512$ and $p = 1$

This formula assumes that all b_i's are uniformly distributed, which is the case for all message blocks, but not for the checksum blocks. However, Genet showed that the approximation is pretty good in practice.

We repeated the same experiment for $n = 512$ to see how the security parameter affects the number of required signatures. Figure 6.4 shows that doubling n, does increase increase the mean number of required faulty signatures to 24.832. This can be explained with the approximative formula derived by Genet. With a higher n, W-OTS+ uses more hash-chains. For $n = 256$ and $w = 16$, W-OTS+ uses $\ell = 67$ chains while for $n = 512$ $\ell = 131$ chains are used. When inserting the value of ℓ in the formula, we see that for achieving a similar success probability, more faulty signatures are needed. Again our results are slightly more optimistic than the formula suggests due to reasons already discussed.

Note that the parameters h and d have no effect on the number of required faulty signatures other than limiting the number of signatures that can be obtained. Additionally, note that practical parameter choices according to [HBGM17] are $h \in \{20, 40, 60\}$ and $d \in \{2, 3, 4, 6, 8, 12\}$ and, thus, in practice the parameter choice that limits the number of obtainable faulty signatures the most is $h = 20, d = 2$, which still allows $2^{10} - 1$ faulty signatures. Thus, we conclude that in practice this limitation is not relevant and an attack is very likely to succeed.

Our biased results in the previous experiments suggested, that the adversary can drastically improve the attack success probability by simply trying the forgery several times for different seeds. To investigate how this affects the overall success probability, we introduced the parameter p, which is the number of forgery trials and then repeated the experiment for $p = \{1, 2, 4, 8, 16, 24, 32\}$. For $p = 1$ this is the same experiment as before. Figure 6.5 shows the results of the experiments. We can see that, indeed, the number of required signatures decreases with the parameter p. The mean required number is plotted as a red line, while the boxes depict the first and third quartiles and the dashed lines show the minimum and maximum values. For $p \geq 24$ it suffices to collect 10 faulty signatures for an attack success rate of 50%. Note that again this result is slightly optimistic, since we do p forgery trials per iteration, i.e., if we use $p = 32$ and require α faulty signatures, this means that we attempted a total of $p \cdot \alpha$ forgeries. However, our results prove that the adversary can drastically improve his chances by investing more computational effort.



p	mean
1	17.6
2	15.1
4	13.3
8	11.9
16	10.6
24	10.0
32	9.6

Figure 6.5: Experimental results for the fault attack simulation for $n = 256$ and $p = \{1, 2, 4, 8, 16, 24, 32\}$

6.5 Countermeasures

Several countermeasures can be applied to prevent the attack presented. A straightforward countermeasure to detect faults is to do the computation multiple times as suggested, e.g., in [Ott04] and check if the result differs. Since an adversary is not able to reproduce the same fault in practice, it can never happen that the signatures are equal and at least one is faulty. Another trivial way of checking for faults in the context of digital signatures is to validate the signature after it has been generated. Faulty XMSS signatures will always be invalid and, thus, can be easily detected. If such a faulty signature is detected, the device should output no signature, but an error instead.

However, both these countermeasures are impractical, since they result in a lot of computational overhead. An easier way of completely preventing this attack on XMSS^{MT} is to ensure that each W-OTS+ secret key is used only once. This is achieved by caching the signatures on the upper layers. Since the leaves of the hypertree are used in strict order, caching one signature per layer is sufficient. Once XMSS^{MT} switches to a new leaf on an upper layer, the corresponding W-OTS+ signature is no longer needed and can be deleted. Thus, this method is very practical and additionally improves the performance of the signature generation, since it reduces the number of W-OTS+ signatures that need to be generated. The XMSS Internet Draft [HBGM17] *recommends* this optimization for performance. However, we conclude that it is also essential for physical attack security.

Note that this countermeasure is not feasible for SPHINCS since the leaves are not used in order there, which subsequently would require storing all W-OTS+ signatures generated forever.

7 Discussion

This chapter concludes this thesis. Section 7.1 summarizes the contributions of this thesis, which are the results of Chapter 4, Chapter 5, and Chapter 6. Section 7.2 discusses the conclusions that can be drawn from our results and the relevance for the XMSS standard. Section 7.3 provides recommendations for implementers of XMSS in a brief fashion with an emphasis on practical aspects rather than scientific research. Section 7.4 presents potential future work which either arises from the results of this thesis or related topics that have been excluded from this thesis but are essential for physical attack security

7.1 Summary

After Chapter 1, Chapter 2, and Chapter 3 introduced the motivation and required theoretical background on physical attacks and hash-based signature schemes, Chapter 4 presented the first contribution of this thesis that is the extensive analysis of XMSS for side-channel vulnerabilities. To provide a precise analysis, we started with the assumption that the used hash function and PRNG do not leak any information about the secret data processed. This showed that XMSS is resistant under these assumptions, which was done from the bottom up, i.e., starting with W-OTS+ and continuing with XMSS and XMSS^{MT}.

The only part of W-OTS+ that is processing secret data is the chaining function. Since the chaining function is only applying the hash function to the secret key parts, it is trivially side-channel resistant, both against timing and power analysis attacks, under the assumption that the hash function is side-channel resistant.

The same argumentation can be applied to XMSS: The only secret information processed within XMSS is the secret seed which is used for pseudorandom number generation of W-OTS+ secret keys and the secret keys itself. Thus, if the PRNG and W-OTS+ are side-channel resistant, so is XMSS. However, it needs to be emphasized that while this equal resistance holds if there is no leakage at all, it does not hold if there is very small leakage. XMSS re-computes the W-OTS+ public keys several times, because they are required for the authentication path computation. Thus, if a minor leakage occurs during these computations, it might be the case that though W-OTS+ is side-channel resistant, XMSS is not. Nonetheless, we conclude that XMSS provides strong side-channel resistance under the used assumptions. Additionally, we found that XMSS^{MT}, which is the hypertree variant of XMSS, has equal side-channel resistance as XMSS.

Chapter 4 concluded with the analysis of our assumptions, namely the side-channel resistance of the hash function and PRNG. We have shown that the hash function specified by the XMSS Internet Draft, as it is used within the W-OTS+ chaining function, is not susceptible to known timing and power analysis attacks. However, we found that while the PRNG provides resistance against timing attacks, it is vulnerable, in theory, to DPA attacks. This is mainly the case because the method of obtaining pseudorandom W-OTS+ secret keys can be chosen by the implementer. Thus, if a vulnerable one is chosen, the entire scheme is vulnerable.

The found vulnerability is further elaborated in Chapter 5. We proposed a SHA2 PRNG, which is cryptographically secure but vulnerable to DPA attacks. We proposed and implemented the DPA attack using power traces generated by our own power simulator. We started by creating an implementation that leaked the HW of each byte of the intermediate values and showed that if there is no noise at all, only 32 traces are sufficient to recover an intermediate state with over 90% success probability. The recovered state can be used to recover all W-OTS+ secret keys and, thus, create universal forgeries. However, this result can only provide a lower bound of required traces, since physically measured traces will always contain noise, which is caused by both measurement errors and physical properties of the attacked cryptographic device. Additionally, the 8-bit HW leakage model is highly unrealistic for SHA2 implementations, since all computations inside the SHA2 compression function perform 32-bit arithmetic, i.e., a

vast majority of implementations is using 32-bit integers. However, we implemented a partial DPA attack which allows the recovery of the key from the leaked HW of 32-bit words. We found in our simulations that a single modular addition operation can be attacked with a success probability of over 95% with around 512 traces, whilst using 2048 traces yields a 100% probability of recovering a single 32-bit word. For bitwise AND we found that to achieve a success rate of over 90%, around 4048 traces are required, which is more than for modular addition. Additionally, we found that the success probability for the bitwise AND operation never reaches 100%, which is caused by the fact that the key hypothesis zero can never be recovered. This limitation is caused by our attack setup and can be mitigated by looking at the Pearson correlation values found. If these values are lower than a certain thresholds, i.e., no key hypothesis correlates with the given traces, it is likely that zero is the correct sub-key.

The second attack presented was a fault attack on $XMSS^{MT}$, which is the hypertree version of XMSS. We adapted a very recent fault attack on SPHINCS for use with $XMSS^{MT}$. As simulation of the attack was implemented to determine how many faulty signatures were required. A relatively small number of around 10 faulty signatures is enough to recover a partial W-OTS+ key, which allows the creation of an existential forgery for the root of an XMSS tree in more than 50% of cases. This consequently allows the creation of universal $XMSS^{MT}$ forgeries by replacing the lower XMSS trees with forged ones. We emphasized that the attack is possible with significantly fewer faulty signatures than previous work suggested, if the adversary is capable of attempting the forgery several times for different seeds. Chapter 6 concluded with the discussion of countermeasures against the presented fault attack. Caching all W-OTS+ signatures generated presents a feasible countermeasure that entirely prevents the attack.

7.2 Conclusion

Secure implementations of XMSS are required to resist side-channel and fault attacks. Although the XMSS Internet Draft [HBGM17] claims their natural side-channel resistance, until now there was no extensive analysis available. This thesis provides this necessary analysis and we confirm the conjecture that XMSS provides very strong resistance against passive timing and power analysis attacks. The only component that might be attackable by DPA is the PRNG, since it can be chosen by the implementer and processes secret data which enables universal forgeries when recovered. However, we conclude that the method of generating pseudorandom numbers proposed by the XMSS Internet Draft is not susceptible when used with the parameters of the current Internet Draft. Nonetheless, if the implementer chooses to use a different PRNG, additional side-channel analysis will be required.

Additionally, we conclude that XMSS, while being resistant to passive attacks, is vulnerable to fault attacks when used in a hypertree setup. Straightforward implementations, which do not cache generated W-OTS+ signatures of XMSS tree roots, can be attacked by injecting faults into the tree computations and consequently forcing the cryptographic device to sign different values with the same W-OTS+ secret key. If an adversary manages to retrieve about 10 signatures for different faulty roots, he can achieve at least a 50% success rate of universal forgeries. This attack can be fully prevented by caching all W-OTS+ signatures generated. Thus, we conclude that a secure implementation is required to use this optimization, which also improves performance.

We further conclude that if the PRNG is side-channel resistant and the implementation uses caching of W-OTS+ signatures, there are currently no attacks vectors of which we are aware that can be used to create XMSS forgeries - neither existential nor universal.

7.3 Recommendations for Implementers

To create implementations resistant to side-channel and fault attacks, implementers need to pay special attention to the following three points:

- **Caching of W-OTS+ signatures**: The recomputation of W-OTS+ signatures for the XMSS roots in the XMSS^{MT} scheme presents a severe vulnerability to fault attacks. If an implementation recomputes the W-OTS+ signatures for each XMSS^{MT} signature generation, an adversary can inject faults into the XMSS root computations which consequently allows him to recover a partial W-OTS+ secret key, enabling him to create existential W-OTS+ forgeries. Replacing parts of the XMSS hypertree enables enables adversaries to create universal XMSS^{MT} forgeries. Caching the W-OTS+ signatures entirely prevents this attack and, thus, we conclude that it is imperative for physical attack security. Additionally, the caching significantly improves performance and should be implemented anyway. Due to the strictly ordered use of leaves in XMSS, it is sufficient to cache one signature per hypertree layer excluding the top layer, i.e., exactly d-1 signatures, which should be feasible for every practical implementation.

- **Side-channel resistant PRNG**: We presented a DPA on a vulnerable PRNG, which allows recovery of all W-OTS+ secret keys and, thus, to create universal XMSS^{MT} forgeries. Since the choice of the PRNG is left to the implementer, an implementation is insecure if a vulnerable one is chosen. We conclude that the PRNG, which is recommended by the XMSS Internet Draft, is not vulnerable to any known attack and recommend using it as it is. If a different method of pseudorandom key generation is used, its resistance to side-channel and fault attacks needs to be extensively verified since its security is central to the resistance of XMSS.

- **Optimized authentication path computation**: Optimized algorithms, like the BDS algorithm [BDS09], minimize the number of W-OTS+ public key recomputations. While this is mainly a performance optimization, it also limits the accesses to the secret keys and consequently also the use of the PRNG. If they are accessed more seldom, less leakage can occur which mitigates the overall susceptibility to side-channel attacks. Thus, even if new attack vectors (e.g., in the PRNG) are discovered, an implementation may still retain security, because an adversary cannot obtain enough traces for a successful attack.

If these recommendations are implemented, the overall side-channel and fault resistance of XMSS is very strong and we conclude that, unless a totally new attack category is found, adversaries are unlikely to succeed. However, it must be emphasized that this does not apply to stateless alternatives like SPHINCS.

7.4 Future Work

We mainly see three interesting areas of research emerging from the results of this thesis:

- **Side-channel analysis of other PRNG**: Our results suggest that the side-channel resistance of the PRNG is essential for overall side-channel resistance of XMSS. However, very little research is available on side-channel resistance of different methods of obtaining pseudorandom numbers. Analyzing their resistance in general will help to identify appropriate candidates for the use in hash-based cryptography in the future. Additionally, discovering vulnerabilities may also enable detection of weaknesses of other cryptographic schemes that use a PRNG. While there is some work available on leakage resilient PRNG, the schemes proposed there are highly specialized and seem to be used very rarely in practice. Thus, additional analysis of practical PRNG is required.

- **Advanced side-channel analysis of SHA2 and SHA3**: While we showed that existing attacks on HMAC based upon SHA2 and SHA3 cannot be adapted to attack the hash function within W-OTS+, there might be other powerful side-channel attacks that are applicable.
- **Countermeasures for stateless hash-based signature schemes**: We have shown that the presented fault attack can be easily prevented for XMSS, but the same countermeasures do not apply to the stateless alternative SPHINCS. However, in general, stateless schemes are much more attractive for practical deployment, because they allow a drop-in replacement of current signature schemes like DSA and RSA. Therefore, finding an effective countermeasure against fault attacks upon stateless hash-based signature schemes is imperative.

Bibliography

[AHPT11] R. Avanzi, S. Hoerder, D. Page, and M. Tunstall. Side-channel attacks on the McEliece and Niederreiter public-key cryptosystems. *J. Cryptographic Engineering*, 1(4):271–281, 2011.

[BB03] D. Brumley and D. Boneh. Remote Timing Attacks Are Practical. In *Proceedings of the 12th USENIX Security Symposium, 2003*. USENIX Association, 2003.

[BBD+13] S. Belaïd, L. Bettale, E. Dottax, L. Genelle, and F. Rondepierre. Differential power analysis of HMAC SHA-2 in the Hamming weight model. In *2013 International Conference on Security and Cryptography (SECRYPT)*, pages 1–12. 2013.

[BDH11] J. Buchmann, E. Dahmen, and A. Hülsing. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In B.-Y. Yang, editor, *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011. Proceedings*, pages 117–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[BDL97] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In W. Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques. Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[BDL01] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Eliminating Errors in Cryptographic Computations. *J. Cryptology*, 14(2):101–119, 2001.

[BDS09] J. Buchmann, E. Dahmen, and M. Szydlo. Hash-based Digital Signature Schemes. In D. J. Bernstein, J. Buchmann, and E. Dahmen, editors, *Post-Quantum Cryptography*, pages 35–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Ber05] D. J. Bernstein. Cache-timing attacks on AES, 2005. `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[BH16] L. G. Bruinderink and A. Hülsing. "Oops, I did it again" – Security of One-Time Signatures under Two-Message Attacks. Cryptology ePrint Archive, Report 2016/1042, 2016. `http://eprint.iacr.org/2016/1042`.

[BHH+15] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn. SPHINCS: Practical Stateless Hash-Based Signatures. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques. Proceedings, Part I*, pages 368–397. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[BHLV17] D. J. Bernstein, N. Heninger, P. Lou, and L. Valenta. Post-quantum RSA. In T. Lange and T. Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017. Proceedings*, volume 10346 of *Lecture Notes in Computer Science*, pages 311–329. Springer, 2017.

[BIS+16] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, J. M. Martinis, and H. Neven. Characterizing quantum supremacy in near-term devices. *arXiv preprint arXiv:1608.00263*, 2016.

[BMM00] I. Biehl, B. Meyer, and V. Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In M. Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference. Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2000.

[BOS06] J. Blömer, M. Otto, and J. Seifert. Sign Change Fault Attacks on Elliptic Curve Cryptosystems. In L. Breveglieri, I. Koren, D. Naccache, and J. Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006. Proceedings*, volume 4236 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2006.

[BSVS16] V. Bahadur, D. Selvakumar, Vijendran, and P. M. Sobha. Reconfigurable side channel attack resistant true random number generator. In *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*, pages 1–6. 2016.

[BT11] B. B. Brumley and N. Tuveri. Remote Timing Attacks Are Still Practical. In V. Atluri and C. Díaz, editors, *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.

[Buc02] J. A. Buchmann. *Introduction to Cryptography*. Springer, 2002. ISBN 0-387-95034-6.

[Buc16] J. Buchmann. Digitale Signaturen. In *Einführung in die Kryptographie*, pages 245–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[CDK+10] S. Chari, V. V. Diluoffo, P. A. Karger, E. R. Palmer, T. Rabin, J. R. Rao, P. Rohatgi, H. Scherzer, M. Steiner, and D. C. Toll. Designing a Side Channel Resistant Random Number Generator. In D. Gollmann, J. Lanet, and J. Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010. Proceedings*, volume 6035 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2010.

[CEvMS15] C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt. Differential Power Analysis of a McEliece Cryptosystem. In T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, editors, *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015. Revised Selected Papers*, volume 9092 of *Lecture Notes in Computer Science*, pages 538–556. Springer, 2015.

[CSW17] E. Carmon, J. Seifert, and A. Wool. Photonic side channel attacks against RSA. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017*, pages 74–78. IEEE Computer Society, 2017.

[DSS05] C. Dods, N. P. Smart, and M. Stam. Hash Based Digital Signature Schemes. In N. P. Smart, editor, *Cryptography and Coding, 10th IMA International Conference, 2005. Proceedings*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer, 2005.

[EvMY14] T. Eisenbarth, I. von Maurich, and X. Ye. Faster Hash-Based Signatures with Bounded Leakage. In T. Lange, K. Lauter, and P. Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013: 20th International Conference. Revised Selected Papers*, pages 223–243. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[FLRV09] P. Fouque, G. Leurent, D. Réal, and F. Valette. Practical Electromagnetic Template Attack on HMAC. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop. Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2009.

[GB17] S.-L. Gazdag and D. Butin. A very basic reference implementation for the Internet-Draft "XMSS: Extended Hash-Based Signatures" (Version 9), 2017. `http://www.square-up.org/downloads/xmss_2016-07-26.tar.gz` (Accessed on 08/28/2017).

[Gen17] A. Genet. Hardware Attacks against Hash-based Cryptographic Algorithms, 2017. Master Thesis, École polytechnique fédérale de Lausanne, School of Computer and Communication Sciences, Lausanne, Switzerland.

[GGH97] O. Goldreich, S. Goldwasser, and S. Halevi. Public-Key Cryptosystems from Lattice Reduction Problems. In B. S. K. Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference. Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 1997.

[GMO01] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop. Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[GST14] D. Genkin, A. Shamir, and E. Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference. Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2014.

[HBGM17] A. Hülsing, D. Butin, S.-L. Gazdag, and A. Mohaisen. XMSS: Extended Hash-based Signatures, July 2017. Work in Progress - https://datatracker.ietf.org/doc/draft-irtf-cfrg-xmss-hash-based-signatures/.

[HMP10] S. Heyse, A. Moradi, and C. Paar. Practical Power Analysis Attacks on Software Implementations of McEliece. In N. Sendrier, editor, *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010. Proceedings*, volume 6061 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2010.

[HPS98] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In J. Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, 1998. Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.

[HRB13] A. Hülsing, L. Rausch, and J. Buchmann. Optimal Parameters for XMSS MT. In A. Cuzzocrea, C. Kittl, D. E. Simos, E. Weippl, and L. Xu, editors, *Security Engineering and Intelligence Informatics: CD-ARES 2013 Workshops: MoCrySEn and SeCIHD. Proceedings*, pages 194–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[HRS16a] A. Hülsing, J. Rijneveld, and P. Schwabe. ARMed SPHINCS. In C.-M. Cheng, K.-M. Chung, G. Persiano, and B.-Y. Yang, editors, *Public-Key Cryptography – PKC 2016: 19th IACR International Conference on Practice and Theory in Public-Key Cryptography. Proceedings, Part I*, pages 446–470. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[HRS16b] A. Hülsing, J. Rijneveld, and F. Song. Mitigating Multi-target Attacks in Hash-Based Signatures. In C.-M. Cheng, K.-M. Chung, G. Persiano, and B.-Y. Yang, editors, *Public-Key Cryptography – PKC 2016: 19th IACR International Conference on Practice and Theory in Public-Key Cryptography. Proceedings, Part I*, pages 387–416. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[Hül13] A. Hülsing. W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes. In A. Youssef, A. Nitaj, and A. E. Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013: 6th International Conference on Cryptology. Proceedings*, pages 173–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[IBM17] IBM. IBM Builds Its Most Powerful Universal Quantum Computing Processors. `https://www-03.ibm.com/press/us/en/pressrelease/52403.wss`, May 2017. (Accessed on 08/28/2017).

[JF11] D. Jao and L. D. Feo. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In B. Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), 1997.

[KJJ99] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference. Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KJJR11] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *J. Cryptographic Engineering*, 1(1):5–27, 2011.

[KNSS13] J. Krämer, D. Nedospasov, A. Schlösser, and J. Seifert. Differential Photonic Emission Analysis. In E. Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013. Revised Selected Papers*, volume 7864 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[Koc96] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference. Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[KPG99] A. Kipnis, J. Patarin, and L. Goubin. Unbalanced Oil and Vinegar Signature Schemes. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques. Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 1999.

[Krä15] J. Krämer. *Why cryptography should not rely on physical attack complexity*. Ph.D. thesis, Berlin Institute of Technology, 2015.

[Lam79] L. Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International Palo Alto, 1979.

[LM95] F. Leighton and S. Micali. Large provably fast and secure digital signature schemes based on secure hash functions, 1995. US Patent 5,432,852.

[LSCH10] M.-K. Lee, J. E. Song, D. Choi, and D.-G. Han. Countermeasures against Power Analysis Attacks for the NTRU Public Key Cryptosystem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E93.A(1):153–163, 2010.

[McE78] R. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, 1978.

[MCF17] D. McGrew, M. Curcio, and S. Fluhrer. Hash-Based Signatures, June 2017. Work in Progress - https://datatracker.ietf.org/doc/draft-mcgrew-hash-sigs/.

[Mer79] R. C. Merkle. *Secrecy, authentication, and public key systems.* Ph.D. thesis, Stanford University, 1979.

[Mer90] R. C. Merkle. A Certified Digital Signature. In G. Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 218–238. Springer New York, New York, NY, 1990.

[MI88] T. Matsumoto and H. Imai. Public Quadratic Polynominal-Tuples for Efficient Signature-Verification and Message-Encryption. In C. G. Günther, editor, *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques. Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 419–453. Springer, 1988.

[MMN04] S. Moriai, M. Matsui, and J. Nakajima. A Description of the Camellia Encryption Algorithm. RFC 3713, April 2004.

[MOP07] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MSSS11] H. G. Molter, M. Stöttinger, A. Shoufan, and F. Strenzke. A simple power analysis attack on a McEliece cryptoprocessor. *Journal of Cryptographic Engineering*, 1(1):29–36, 2011. ISSN 2190-8516.

[MTMM07] R. P. McEvoy, M. Tunstall, C. C. Murphy, and W. P. Marnane. Differential Power Analysis of HMAC Based on SHA-2, and Countermeasures. In S. Kim, M. Yung, and H. Lee, editors, *Information Security Applications, 8th International Workshop, WISA 2007. Revised Selected Papers*, volume 4867 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2007.

[Nat99] National Institute of Standards and Technology. FIPS PUB 46-3 - Data Encryption Standard (DES), October 1999. `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

[Nat01] National Institute of Standards and Technology. FIPS PUB 197 - Advanced Encryption Standard (AES), November 2001. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`.

[Nat15a] National Institute of Standards and Technology. FIPS PUB 180-4: Secure Hash Standard, August 2015. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`.

[Nat15b] National Institute of Standards and Technology. FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, August 2015. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`.

[Nat16a] National Institute of Standards and Technology. NIST Special Publication 800-57: Recommendation for Key Management Part 1 Revision 4, January 2016. `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf`.

[Nat16b] National Institute of Standards and Technology. POST-QUANTUM CRYPTO STANDARDIZATION. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/cfp-announce-dec2016.html`, December 2016. (Accessed on 08/28/2017).

[Ope] OpenSSL Implementation of SHA256. `https://github.com/openssl/openssl/blob/master/crypto/sha/sha256.c` (Accessed on 08/28/2017).

[Ott04] M. Otto. *Fault attacks and countermeasures*. Ph.D. thesis, Paderborn University, 2004.

[QS01] J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In I. Attali and T. P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001. Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[Rey17] M. Reynolds. Quantum simulator with 51 qubits is largest ever. `https://www.newscientist.com/article/2141105-quantum-simulator-with-51-qubits-is-largest-ever/`, July 2017. (Accessed on 08/28/2017).

[RMB15] C. Rebeiro, D. Mukhopadhyay, and S. Bhattacharya. *Timing channels in cryptography : a micro-architectural perspective*. Springer International Publishing, Cham, 2015.

[RR02] L. Reyzin and N. Reyzin. Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying. In L. Batten and J. Seberry, editors, *Information Security and Privacy: 7th Australasian Conference, ACISP 2002. Proceedings*, pages 144–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[Sho97] P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[SPY+10] F. Standaert, O. Pereira, Y. Yu, J. Quisquater, M. Yung, and E. Oswald. Leakage Resilient Cryptography in Practice. In A. Sadeghi and D. Naccache, editors, *Towards Hardware-Intrinsic Security - Foundations and Practice*, Information Security and Cryptography, pages 99–134. Springer, 2010.

[SSMS09] A. Shoufan, F. Strenzke, H. G. Molter, and M. Stöttinger. A Timing Attack against Patterson Algorithm in the McEliece PKC. In D. H. Lee and S. Hong, editors, *Information, Security and Cryptology - ICISC 2009, 12th International Conference. Revised Selected Papers*, volume 5984 of *Lecture Notes in Computer Science*, pages 161–175. Springer, 2009.

[STM+08] F. Strenzke, E. Tews, H. G. Molter, R. Overbeck, and A. Shoufan. Side Channels in the McEliece PKC. In J. Buchmann and J. Ding, editors, *Post-Quantum Cryptography: Second International Workshop, PQCrypto 2008. Proceedings*, pages 216–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Str10] F. Strenzke. A Timing Attack against the Secret Permutation in the McEliece PKC. In N. Sendrier, editor, *Post-Quantum Cryptography: Third International Workshop, PQCrypto 2010. Proceedings*, pages 95–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Str13] F. Strenzke. Timing Attacks against the Syndrome Inversion in Code-Based Cryptosystems. In P. Gaborit, editor, *Post-Quantum Cryptography: 5th International Workshop, PQCrypto 2013. Proceedings*, pages 217–230. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[SW06] J. H. Silverman and W. Whyte. Timing Attacks on NTRUEncrypt Via Variation in the Number of Hash Calls. In M. Abe, editor, *Topics in Cryptology – CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007. Proceedings*, pages 208–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[TE15] M. Taha and T. Eisenbarth. Implementation Attacks on Post-Quantum Cryptographic Schemes. Cryptology ePrint Archive, Report 2015/1083, 2015. `http://eprint.iacr.org/2015/1083`.

[TRS16] M. M. I. Taha, A. Reyhani-Masoleh, and P. Schaumont. Keymill: Side-Channel Resilient Key Generator. *IACR Cryptology ePrint Archive*, 2016:710, 2016.

[TS13] M. M. I. Taha and P. Schaumont. Differential Power Analysis of MAC-Keccak at Any Key-Length. In K. Sakiyama and M. Terada, editors, *Advances in Information and Computer Security - 8th International Workshop on Security, IWSEC 2013. Proceedings*, volume 8231 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2013.

[Viz07] N. V. Vizev. Side Channel Attacks on NTRUEncrypt, 2007. Bachelor's thesis, Technical University of Darmstadt, Germany, 2007. Available on http://www.cdc.informatik.tu-darmstadt.de/reports/reports/Nikolay_Vizev.bachelor.pdf.

[vMG14] I. von Maurich and T. Güneysu. Towards Side-Channel Resistant Implementations of QC-MDPC McEliece Encryption on Constrained Devices. In M. Mosca, editor, *Post-Quantum Cryptography: 6th International Workshop, PQCrypto 2014. Proceedings*, pages 266–282. Springer International Publishing, Cham, 2014.

[YSPY10] Y. Yu, F. Standaert, O. Pereira, and M. Yung. Practical leakage-resilient pseudorandom generators. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*, pages 141–151. ACM, 2010.

[ZKSH12] M. Zohner, M. Kasper, M. Stöttinger, and S. A. Huss. Side channel analysis of the SHA-3 finalists. In W. Rosenstiel and L. Thiele, editors, *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012*, pages 1012–1017. IEEE, 2012.

[ZWW13] X. Zheng, A. Wang, and W. Wei. First-order collision attack on protected NTRU cryptosystem. *Microprocessors and Microsystems - Embedded Hardware Design*, 37(6-7):601–609, 2013.