TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Chair of Privacy and Security

Master Thesis

Hash-based Digital Signature Schemes

Amelie Wagner

Born on: 17th September 1994
Matriculation number: 3949194

to achieve the academic degree

Master of Science (M.Sc.)

Advisor
M.Sc. Fabio Campos

Supervisors
Dr.-Ing. Elke Franz
Dr.-Ing. Stefan Köpsell

Submitted on: 9th March 2022

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Hash-based Digital Signature Schemes* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 9. März 2022

Amelie Wagner

Abstract

The security of current digital signature systems is in danger because they are not quantum secure. Hash-based digital signature schemes provide a good countermeasure, but they generally perform worse in comparison to classical schemes. Therefore, performance improvement remains a crucial task. In this work, several state-of-the-art hash-based signature systems are analyzed in detail and methods for performance improvement are presented. Using these methods, key generation is 25% faster and signature verification up to 22% faster. However, signature size is increased by up to 29%.

Contents

1. Introduction

In this chapter, the motivation and goals of this work are presented. First, the quantum threat is introduced. Afterwards, the goals of this work are defined. The last section describes the general structure of this thesis.

1.1. Quantum Threat

Quantum computing theory has been researched extensively and is considered the greatest threat to modern cryptography, also referred to as *quantum threat* [1]. It was first mentioned in 1996, when Shor [2] proposed a quantum algorithm for factorization and calculating the discrete logarithm that is exponentially faster than any known classical algorithm. In 1996, Grover [3] proposed a quantum searching algorithm, speeding up search efficiency from classical $\mathcal{O}(N)$ to $\sqrt{N}$. Based on these discoveries, further research shows that present asymmetric cryptographic schemes (whose security is based on the difficulty of factorizing large prime numbers and the discrete logarithm problem) can be broken, once a quantum computer with a sufficient number of quantum bits exists. Symmetric cryptography is also effected by this quantum threat. However, its security can be increased by using larger key spaces. [1] Example quantum attacks on RSA [4] (whose security relies on the prime factorization problem) are shown by Soni et al. [5] and Wang et al. [6].

There exist classes of cryptographic systems considered more resistant against quantum attacks, also denoted as *quantum secure*: Hash-based cryptography, code-based cryptography, lattice-based cryptography, multivariate-quadratic-equations cryptography, symmetric cryptography and isogeny based cryptography. [7, 8] This work focuses on hash-based cryptography used for digital signature systems, referred to as *hash-based signature systems (HBS)*.

1.2. Goals

Quantum secure cryptographic schemes have an overall worse performance in comparison to classical cryptographic systems. This is also a problem for hash-based signature systems. The classical algorithms elliptic curve digital signature algorithm (ECDSA) [9] and Rivest-Shamir-Adleman (RSA [4]) outperform HBS. Noel et al. [10] show that these two classical algorithms outperform the Merkle Signature Scheme (common HBS, see Section 2.4) in key generation, signature generation and verification time. Therefore, finding possible efficiency improvements for existing HBS is a crucial task and also the main goal of this work. For related work on improving the performance of HBS in several ways, see Chapter 3.

1.3. Structure

In Chapter 2 the fundamentals necessary for this work are introduced: After a brief introduction of digital signature schemes in general, the one-time signature schemes Lamport-Diffie One-Time Signature Scheme and Winternitz One-Time Signature Scheme are presented, including a basic example. Afterwards, the Merkle Signature Scheme, Leighton-Micali Signature Scheme and eXtended Merkle Signature Scheme are explained in detail. Chapter 3 summarizes state-of-the-art literature related to the thesis topic. Chapter 4 proposes methods to improve the performance of HBS. Afterwards, these methods are evaluated in Chapter 5. Finally, Chapter 6 discusses the results of this work and provides possibilities for future work. In Appendix A and Appendix B, the T_5-Tree$^+$ concept and generation of the evaluation results are implemented.

2. Background

This chapter introduces the scientific background of hash-based signature systems, which serves as the basis for this work. First, the general concept of digital signature systems and hash functions are elaborated. Afterwards, the most common concepts for hash-based digital signature systems are explained. The presented hash-based schemes are the *Lamport-Diffie one-time signature scheme (LD-OTS)*, the *Winternitz one-time signature scheme (W-OTS)*, the *Merkle signature scheme (MSS)* and the *extended Merkle signature scheme (XMSS)*.

2.1. Digital Signature Schemes

A *digital signature scheme* uses a set of rules and a set of parameters to verify the identity of the originator, the integrity of data and non-repudiation. [11] In this section, for explanatory reasons, the term data refers to a message sent from a sender to a receiver across a network (e.g. LAN). The sender is the person signing the message, the recipient usually wants to verify the received message. Therefore they are referred to as signer and verifier. Notably, the verifier can also be a third party, not just the recipient. The *digital signature* σ of a message, generated by a digital signature scheme, is a value dependent on some secret known only to the signer (usually the private key X) and on the content of the message being signed. The corresponding public key Y can be used to verify the authenticity of the signature without requiring access to the signers private key X. This ensures that the message actually belongs to the signer - for example to detect a lying signer trying to repudiate their signature, a fraudulent claimant arguing the message is theirs, or a message that has been tampered with. To ensure the above mentioned properties of digital signatures a digital signature scheme consists of the following parts (see also Figure 2.1) [12] :

1. The *key generation algorithm* creates a private key X used for signature generation, and a public key Y used for signature verification. Both keys are mathematically dependent on each other, the way being determined by the specific type of the signature scheme (e.g. the Winternitz signature scheme, see Section 2.3.2).
2. The *signing algorithm* creates the digital signature σ of a message depending on the private key X of the signer and the content of the message.
3. The *verification algorithm* is used by the verifier to check the validity of the signature σ and the corresponding message with the public key Y.

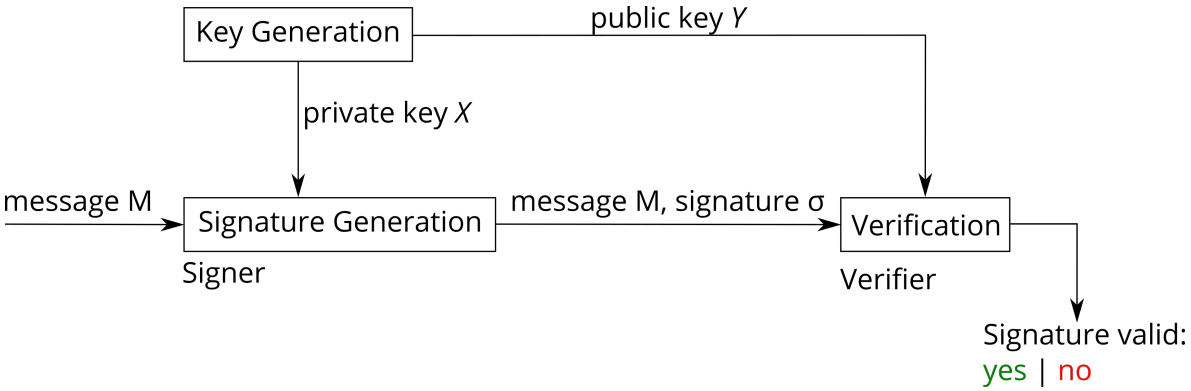
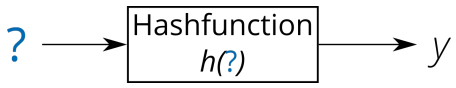


Figure 2.1.: The general structure of a digital signature system.

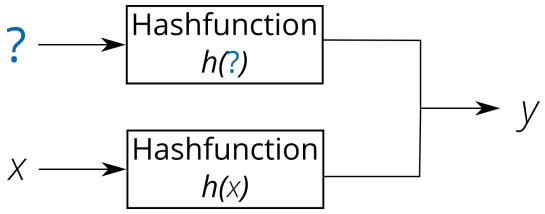
2.2. Definition of Hash Functions

The security of the one-time signature methods presented in Section 2.3 is based on cryptographically secure hash functions. A hash function is a function that can be computed efficiently and maps strings of arbitrary length to strings of fixed length [13]. Therefore, a hash function h is defined as any function $h : \{0,1\}^* \rightarrow \{0,1\}^n$ [14]. A hash function is considered *cryptographically secure* if it has the following properties [15]:

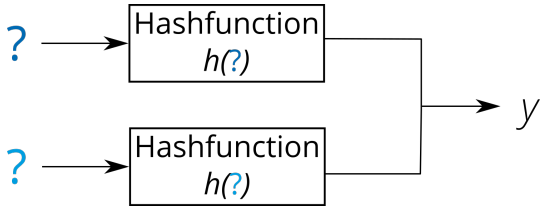
1. **Preimage-Resistance / One-wayness** A hash function h is preimage-resistant if given an output value y, it is computationally infeasible to find any input which generates this output, i.e. finding any preimage x such that $h(x) = y$, when given any y.



2. **Second Preimage-Resistance** A hash function h is second preimage-resistant, if given any input value and the corresponding output, it is computationally infeasible to find another distinct input that produces the same output, i.e. given any x finding a second preimage $x' \neq x$ such that $h(x') = h(x)$.



3. **Collision Resistance** A hash function h is called collision-resistant, if it is computationally infeasible to find a pair of different inputs x, x' that map to the same output value, such that $h(x) = h(x')$.



2.3. One-Time Signature Schemes

This section is based on the work of Buchmann et al. [7]. The two signature schemes Lamport-Diffe and Winternitz are both *one-time signature schemes (OTS)*, meaning the public and private key can be used **once**, for signing a single message. If they are used for generating more than one signature, the signature can be forged. The following types of functions are used for the Lamport-Diffie OTS and the Winternitz One-Time Signature Scheme: The cryptographic hash function h is preimage resistant, second preimage-resistant and collision resistant. It is applied to the original message and generates the message digest.

$$\text{Cryptographic hash function h: } \{0,1\}^* \rightarrow \{0,1\}^n \tag{2.1}$$

The one-way function f is a hash function that is at least preimage-resistant and takes a fixed input length because it is applied to the message digest.

$$\text{One-way function f: } \{0,1\}^n \rightarrow \{0,1\}^n \tag{2.2}$$

2.3.1. Lamport-Diffie One-Time Signature Scheme (LD-OTS)

The Lamport–Diffie One-Time Signature Scheme (LD-OTS) was first proposed by Leslie Lamport in 1979 [16].

LD-OTS Key Generation

The private key X consists of $2n$ bit strings of length n chosen at random. Because the keys and the signature size depend on n, it is also referred to as the security parameter

$$X = (x_0[0], x_0[1], x_1[0], x_1[1], \cdots, x_{n-1}[0], x_{n-1}[1]) \tag{2.3}$$

The public key Y is created from the private key X. For each $x_i[j] \in X, 0 \leq i \leq n-1, j \in \{0,1\}$, the one-way function f (see Equation 2.2) is applied.

$$y_i[j] \in Y = f(x_i[j]), 0 \leq i \leq n-1, j \in \{0,1\} \tag{2.4}$$

$$Y = (y_0[0], y_0[1], y_1[0], y_1[1], \cdots, y_{n-1}[0], y_{n-1}[1]) \tag{2.5}$$

LD-OTS Signature Generation

Before signing, the public key Y has to be published. The private key X (see Equation 2.3) is used to sign the message $M \in \{0,1\}^*$. The cryptographic hash function h (see Equation 2.1) is applied to M in order to get the hash digest m of fixed length n.

$$m = h(M) = (h_0, \cdots, h_{n-1}) \tag{2.6}$$

For each bit $h_i \in m$, the corresponding $x_i[h_i]$ is chosen from the private key X, resulting in the signature σ for the message m.

$$\sigma = (x_0[h_0], x_1[h_1], \cdots, x_{n-1}[h_{n-1}]) = (\sigma_0, \cdots, \sigma_{n-1}) \tag{2.7}$$

LD-OTS Verification

After receiving a message M with the corresponding signature σ, the verifier calculates the message digest $h(M) = m$. To verify the given signature σ, it is necessary to check the following condition.

$$(f(\sigma_0), \cdots, f(\sigma_{n-1})) = (y_0[h_0], \cdots, y_{n-1}[h_{n-1}]) \tag{2.8}$$

If the condition is true, the signature is valid.

2.3.2. Winternitz One-Time Signature Scheme (W-OTS)

LD-OTS signatures are efficient to calculate but have a large size. The Winternitz one-time signature scheme (W-OTS) generates signatures with substantially shorter size. W-OTS uses the same hash function (Equation 2.1) and one-way function (Equation 2.2) as LD-OTS. To counter adaptive chosen-message attacks, a W-OTS contains a checksum, an example calculation is shown at the end of Section 2.3.2.

W-OTS Key Generation

First, two parameters are selected: The Winternitz-Parameter $w \geq 2$ and the security parameter n. As n is the length of the message digest, increasing it leads to higher security because it increases the collision resistance of the hash function. The Winternitz parameter w enables space-time trade-offs (for a detailed explanation see W-OTS Verification in Section 2.3.2).

After selecting the parameters w and n, the values t_1, t_2 and t are calculated. The value t_1 determines the amount of blocks the message digest m will be separated into (see Equation 2.15):

$$t_1 = \left\lceil \frac{n}{w} \right\rceil \tag{2.9}$$

The value t_2 determines the amount of blocks the checksum c will be separated into (see also Equation 2.16):

$$t_2 = \left\lceil \frac{\lfloor \log_2 t_1 \rfloor + 1 + w}{w} \right\rceil \tag{2.10}$$

The value t determines the total amount of blocks (see Equation 2.18) as well as the amount of elements in the private and public keys (see Equation 2.12, 2.13) and the signature (see Equation 2.19):

$$t = t_1 + t_2 \tag{2.11}$$

The private key X consists of t randomly chosen bit strings of length n.

$$X = (x_0, \cdots, x_{t-1}) \tag{2.12}$$

The public key Y is generated by applying the one-way function f to each element $x_i \in X$ consecutively $2^w - 1$ times.

$$y_i \in Y = f^{2^w-1}(x_i), 0 \leq i \leq t-1 \tag{2.13}$$

$$Y = (y_0, \cdots, y_{t-1}) \tag{2.14}$$

Each $y_i \in Y$ is a bit string of length n. The public key Y has to be published before the signature can be generated. One value of the public key corresponds to one full *Winternitz chain*.

W-OTS Signature Generation

For signing a message $M \in \{0,1\}^*$, the cryptographic hash function h (see Equation 2.1) is applied to M (see Equation 2.6). The resulting hash digest m is split into t_1 bit strings of length w. If m is not divisible by w, it is necessary to add leading zeros to m before splitting.

$$m = m_0 \parallel m_1 \parallel \cdots \parallel m_{t_1-1} \tag{2.15}$$

Each bit string $m_i \in m$ is converted to its decimal representation in order to calculate the checksum c. A detailed example why the checksum is necessary is shown at the end of Section 2.3.2 in W-OTS Checksum Example.

$$c = \sum_{i=0}^{t_1-1} (2^w - m_i) \tag{2.16}$$

The checksum c is divided into t_2 bit strings of length w. In order to divide c this way, it may be necessary to add leading zeros to c as a padding.

$$c = c_0 \parallel c_1 \parallel \cdots \parallel c_{t_2-1} \tag{2.17}$$

Afterwards, m and c are concatenated to one block B. This leads to t bit strings of length w in total, as $t = t_1 + t_2$.

$$\begin{aligned} B &= m \parallel c \\ &= m_0 \parallel \cdots \parallel m_{t_1-1} \parallel c_0 \parallel \cdots \parallel c_{t_2-1} \\ &= b_0 \parallel \cdots \parallel b_{t-1} \end{aligned} \tag{2.18}$$

The signature σ is calculated by applying the one-way function f to each part of the private key X (see Equation 2.12) several times: The element $b_i \in B$ determines the amount of times the hash function f is applied to the corresponding $x_i \in X$. One element of the signature is also referred to as one *Winternitz chain*.

$$\sigma = (f^{b_0}(x_0), f^{b_1}(x_1), \cdots, f^{b_{t-1}}(x_{t-1})) = (\sigma_0, \cdots, \sigma_{t-1}) \tag{2.19}$$

W-OTS Verification

Given a signature σ and message M, the hash digest m is generated (see Equation 2.6). Afterwards, the block B is generated out of m as shown in the previous section (see Equations 2.15 to 2.18). To check if the given signature is valid, the one-way function f is applied $2^w - 1 - b_i$ times to each $\sigma_i \in \sigma$. The result is compared to the corresponding $y_i \in Y$. This can also be interpreted as advancing each Winternitz chain in the signature by applying f until the values of the public key are reached.

$$(f^{(2^w-1)-b_0}(\sigma_0), \cdots, f^{(2^w-1)-b_{t-1}}(\sigma_{t-1})) \stackrel{?}{=} (y_0, \cdots, y_{t-1}) \tag{2.20}$$

If each $f^{2^w-1-b_i}(\sigma_i) = y_i$, the signature is valid because $\sigma_i = f^{b_i}(x_i)$ and therefore

$$\begin{gathered} f^{2^w-1-b_i}(\sigma_i) = f^{2^w-1}(x_i) = y_i \\ \text{for } 0 \leq i \leq t-1 \end{gathered} \tag{2.21}$$

As w determines the block size of each block in B, w is allowing a space-time trade-off: When increasing w, the signature size will decrease linearly (the total amount of blocks in B will

decrease) and the effort for key generation, signing and verification will increase exponentially. This is because $w - 1$ hash function calls are necessary for public key generation, and $w - 1$ hash function calls are necessary for signature generation and verification in total.

W-OTS Example Calculation

This section contains an example calculation of W-OTS, including key generation, signature generation, and signature verification. This example is not cryptographically secure, it exists for explanatory reasons only.

1. Choose the parameters $n = 3, w = 2$, message digest $m = 101$, one-way function $f : \{0,1\}^3 \to \{0,1$
2. Calculate t_1, t_2 and t:
   $t_1 = \lceil \frac{n}{w} \rceil = \lceil \frac{3}{2} \rceil = 2, t_2 = \left\lfloor \frac{\lfloor \log_2 t_1 \rfloor + 1 + w}{w} \right\rfloor = \left\lfloor \frac{\lfloor \log_2 2 \rfloor + 1 + 2}{2} \right\rfloor = 2, t = t_1 + t_2 = 2 + 2 = 4$
3. Choose the private key X with $t = 4$ random bit strings of length $n = 3$:
   $X = (x_0, \cdots, x_{t-1}) = (x_0, x_1, x_2, x_3) = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,4)}$
4. Calculate the public key Y from X by applying f to each element in X for $2^w - 1 = 3$ times:
   $Y = (y_0, \cdots, y_{t-1}) = (y_0, y_1, y_2, y_3) = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,4)}$
5. Make m divisible by w, by adding a leading zero. Then, split it into blocks of length w:
   $m = 01 \parallel 01 = m_0 \parallel m_1$. These blocks are used for the checksum calculation:
   $c = (2^w - m_0) + (2^w - m_1) = (4 - 1) + (4 - 1) = 6$. To make c divisible by w as well, one leading zero is added to the binary representation of c. Then, splitting c in blocks of length w yields $c = 01 \parallel 10 = c_0 \parallel c_1$.
6. Generate the block B by concatenating m and c:
   $B = m_0 \parallel m_1 \parallel c_0 \parallel c_1 = b_0 \parallel b_1 \parallel b_2 \parallel b_3 = 01 \parallel 01 \parallel 01 \parallel 10$.
7. The signature σ of m is determined by the parameter B and one-way function f:
   $\sigma = (f^{b_0}(x_0), f^{b_1}(x_1), f^{b_2}(x_2), f^{b_3}(x_3)) = (f^1(5), f^1(7), f^1(3), f^2(6)) = (\sigma_0, \sigma_1, \sigma_2, \sigma_3) =$
   $\begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,4)}$
8. The verifier knows Y, w, n and therefore t_1, t_2 and t. After receiving m and σ from the signer, the block B is calculated as explained in the previous steps. The validity of the signature σ is checked by calculating:
   $(f^{2^w-1-b_0}(\sigma_0), f^{2^w-1-b_1}(\sigma_1), f^{2^w-1-b_2}(\sigma_2), f^{2^w-1-b_3}(\sigma_3)) = (f^2(7), f^2(1), f^2(5), f^1(2)) =$
   $\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,4)}$
   Because these values are the same as in the public key Y, σ is valid.

W-OTS Checksum Example

In this section, the necessity of the W-OTS checksum (see also Equation 2.16) is explained with an example. The attack prevented by the checksum is an *adaptive chosen-message attack*: It is possible for the attacker to generate new messages with matching signatures which depend on previously obtained signatures and messages [12]. After obtaining a message and the corresponding signature, the idea behind the attack is to increase the bits of the received message to generate a new message. The hash function is applied respectively to the corresponding digits in the signature, increasing the Winternitz chain. Then, without the checksum, a new valid signature would be generated for the message. For this example, the same parameters for W-OTS are used as in the the section before.

1. We assume the attacker knows the message digest $m = 101$, the corresponding signature $\sigma = (\sigma_0, \sigma_1, \sigma_2, \sigma_3)$ and the parameters $w = 2, n = 3, Y, f$ of the example in the section before. The attacker can get this information because a digital signature system does not ensure confidentiality of the message or the parameters, the premise is the secrecy of the private key. The goal of the attacker is to forge a signature $\sigma' = (\sigma'_0, \sigma'_1, \sigma'_2, \sigma'_3)$ which is a valid for a message m' chosen by the attacker.

2. The attacker calculates $m = m_0 \parallel m_1 = 01 \parallel 01$, $c = c_0 \parallel c_1 = 01 \parallel 10$ and therefore $B = b_0 \parallel b_1 \parallel b_2 \parallel b_3 = 01 \parallel 01 \parallel 01 \parallel 10$ (see steps 1-6 of section before).

3. The original message digest $m = 101$ is increased by two: $m' = 111$. Make m' divisible by $w = 2$, insert leading zero: $m' = m'_0 \parallel m'_1 = 01 \parallel 11$. Calculate checksum $c' = (2^w - m'_0) + (2^w - m'_1) = (4 - 1) + (4 - 3) = 4$. Insert leading zero to c' to make it divisible by w: $c' = c'_0 \parallel c'_1 = 01 \parallel 00$. Therefore, $B' = b'_0 \parallel b'_1 \parallel b'_2 \parallel b'_3 = 01 \parallel 11 \parallel 01 \parallel 00$.

   Now, the attacker can forge σ'_0, σ'_1 of the signature σ': As $b_0 = b'_0 = 01$, $\sigma'_0 = \sigma_0$. Because the difference between $b_1 = 01$ and $b'_1 = 11$ is 2, applying f two more times to σ_1 leads to σ'_1: $f^2(\sigma_1) = \sigma'_1$.

   Notably, σ_0, σ_1 depend only on the message digest bits m_0, m_1, not on the checksum. Therefore, the attacker could forge a signature for m' without the checksum. But because of the checksum bit $c_3 = b_3 = 10, c'_3 = b'_3 = 00$, $b_3 > b'_3$, it is not possible to calculate σ_3: The attacker would have to calculate $f^{-2}(\sigma_3)$, that is finding two times a preimage to σ_3. As long as the hash function f is preimage resistant, which is assumed (see also Equation 2.2), this is not possible. The attacker can not forge the complete signature σ'.

For a general proof of security of the checksum, see Section 9.3 in McGrew et al. [17]. For LD-OTS, the checksum is not necessary because only one hash function call is used to get to the public key Y. Therefore, it is not possible to generate another valid signature σ' by applying the hash function to the known signature σ again.

2.4. Merkle Signature Scheme (MSS)

This section is also based on the work of Buchmann et al. [7]. The main disadvantage of the one-time signature schemes presented in Section 2.3 is the restriction to use each key pair for only one signature. This is inadequate for most practical situations because the key generation as well as the key distribution take a lot of time and effort. To solve this problem, Merkle [18] proposed the concept of using a binary hash tree, where each leaf represents a different one-time key pair. The root of the tree is the public key Y_{MSS} which combines the one-time key pairs at the leafs. With a tree depth d, 2^d one-time key pairs and corresponding signatures can be generated. This concept is denoted as *Merkle signature scheme (MSS)* and it works with any cryptographic hash function and any one-time signature scheme. The structure of the Winternitz one-time signature scheme fits better into MSS than the Lamport-Diffie one-time signature scheme: If using W-OTS, it is not necessary for the signer to put the public one-time signature key Y_s in the one-time signature σ_s. The verifier will automatically calculate Y_s from the given one-time signature. Therefore, the chosen one-time signature scheme for MSS in this chapter is W-OTS. The cryptographic hash function h (see Equation 2.1) is used.

The combination of W-OTS with the Merkle Tree will be used in the *Leighton-Micali Signature Scheme (LMS)*, see Section 2.5 (and with an advanced Merkle Tree in XMSS, see Section 2.6).

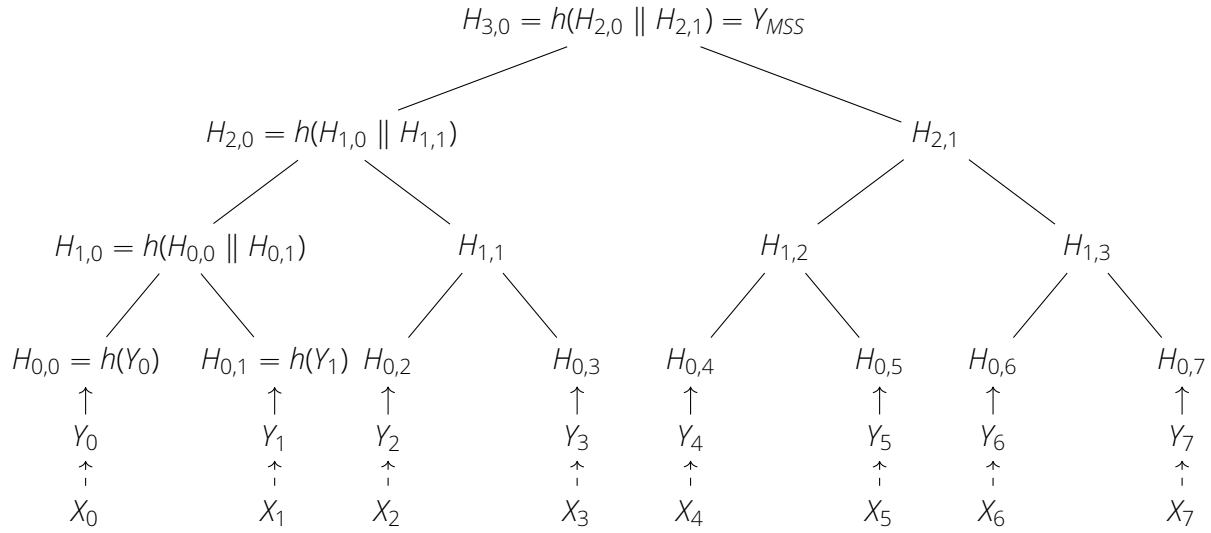


Figure 2.2.: Merkle tree of depth $d = 3$. The composition of the tree is depicted in detail on the left branch, each node consists of the hash digest of its concatenated children (see Equation 2.23). The leafs are the hash digests of their corresponding one-time public key (see Equation 2.22). Because W-OTS is used, the signer generates the public one-time keys by applying the hash function h for $2^w - 1$ times to the corresponding private one-time keys $X_i \in X_{MSS}$ (see Equations 2.13 and Equation 2.51).

2.4.1. MSS Key Generation

The Merkle tree referenced in this section is depicted in Figure 2.2. The signer chooses the tree depth d where $d \geq 2$, and generates 2^d one-time key pairs (X_j, Y_j) where $0 \leq j \leq 2^d - 1$, with X_j being the private key and Y_j being the corresponding public key. The *leaves* of the Merkle tree are the hash digests $H_{i,j}$ of the public key Y_j.

$$H_{i,j} = h(Y_j) \quad \text{for } 0 \leq j \leq 2^d - 1 \tag{2.22}$$

The *inner nodes* of the Merkle tree are computed as follows: Each parent node $H_{i,j}$ is the hash digest of the concatenation of its direct two children:

$$H_{i,j} = h(H_{i-1,2j} \parallel H_{i-1,2j+1}) \quad \text{for } 1 \leq i \leq d \quad \text{for } 0 \leq j < 2^{d-i} \tag{2.23}$$

The *root* of the Merkle tree is the MSS public key Y_{MSS}. The MSS private key X_{MSS} is the collection of one-time signature keys generated before constructing the Merkle tree.

$$\begin{aligned} X_{MSS} &= (X_0, \cdots, X_j, \cdots, X_{2^d-1}) \\ Y_{MSS} &= H_{d,0} \end{aligned} \tag{2.24}$$

The signer publishes the public key Y_{MSS}.

MSS Key Generation: Special Case

For simplification, when explaining MSS, each Winternitz one-time key has *one* value (i.e. contains only one Winternitz chain, $t = 1$). There are two methods to transition from Winternitz one-time public key Y_i (with $t > 1$, includes more than one Winternitz chain) to a leaf $H_{0,j}$ in the Merkle Tree, depending on the specific signature scheme. These methods are explained in detail for the Leighton-Micali Signature Scheme (hashing p Winternitz chains together, see Section 2.5) and XMSS (combining an amount of l Winternitz chains with one L-Tree, see Section 2.6.3 and Figure 2.7).

2.4.2. MSS Signature Generation

The Merkle tree referenced in this section is depicted in Figure 2.3. To sign a message M, the signer needs to generate the signature σ_s. First, the hash digest $m = h(M)$ of length n (see Equation 2.6) is calculated. Then, by using the chosen one-time signature scheme W-OTS, the one-time signature $\sigma_{s/OTS}$ of m is generated with a one-time key $X_s, s \in \{0, \cdots, 2^d - 1\}, X_s \in X_{MSS}$.

$$\sigma_{s/OTS} \leftarrow_{sign} (X_s, m) \tag{2.25}$$

Additional information about the Merkle tree has to be included in σ_s: The index s and the authentication path for the verification key Y_s. The authentication path A_s consists of a sequence of nodes a_i in the Merkle tree:

$$A_s = (a_0, \cdots, a_i, \cdots, a_{d-1}) \tag{2.26}$$

Each node $a_i \in A_s$ is calculated as follows:

$$\begin{aligned} & a_i = H_{i,j} \\ & j = \begin{cases} \lfloor s/2^i \rfloor - 1 \text{ if } \lfloor s/2^i \rfloor \equiv 1 \mod 2 \\ \lfloor s/2^i \rfloor + 1 \text{ if } \lfloor s/2^i \rfloor \equiv 0 \mod 2 \end{cases} \\ & 0 \leq i \leq d - 1 \end{aligned} \tag{2.27}$$

In summary, one MSS signature contains the following elements:

$$\sigma_s = (s, \sigma_{s/OTS}, A_s) \tag{2.28}$$

2.4.3. MSS Signature Verification

When receiving σ_s (see Equation 2.28), the verifier uses $\sigma_{s/OTS}$ to calculate Y_s. This works specifically because W-OTS is used in combination with the Merkle tree: Applying the hash function h for a specific amount of times to σ_s (this is determined by the underlying W-OTS, see Section 2.3.2) automatically generates Y_s. Because of the index s, the verifier knows the leaf-position of the calculated Y_s in the Merkle tree. In combination with the authentication path A_s, the verifier can construct a path from the leaf Y_s to the root of the Merkle tree:

$$P_s = (p_0, \cdots, p_d) \tag{2.29}$$

The path P_s is constructed by using the index s and the authentication path A_s:

$$\begin{aligned} & p_0 = h(Y_s) \\ & p_i = \begin{cases} h(a_{i-1} \parallel p_{i-1}) \text{ if } \lfloor s/2^{i-1} \rfloor \equiv 1 \mod 2 \\ h(p_{i-1} \parallel a_{i-1}) \text{ if } \lfloor s/2^{i-1} \rfloor \equiv 0 \mod 2 \end{cases} \\ & 0 \leq i \leq d \end{aligned} \tag{2.30}$$

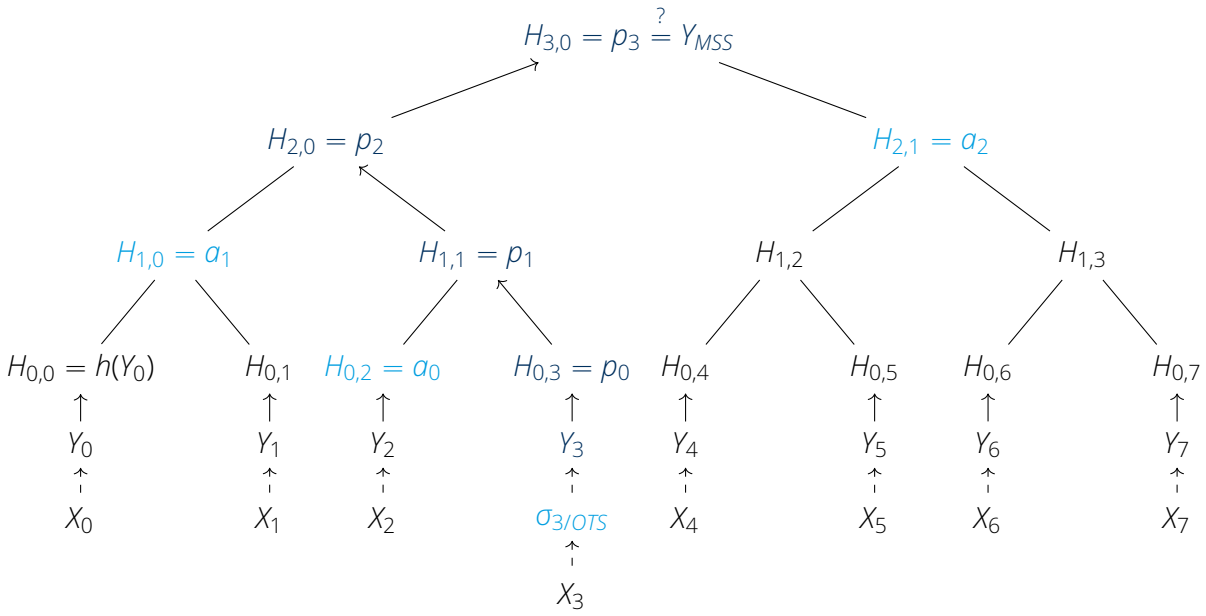


Figure 2.3.: Example for Merkle signature generation and verification, tree depth $d = 3$. The signer generates the Merkle tree and chooses $s = 3$, then calculates the signature $\sigma_3 = (3, \sigma_{s/OTS}, A_3)$. The nodes $H_{0,2}, H_{1,0}, H_{2,1}$ are in the authentication path $A_3 = (a_0, a_1, a_2)$. After receiving σ_3, the verifier uses the one-time signature $\sigma_{3/OTS}$ to calculate Y_3 by applying the hash function h a specific amount of times to it (see also Section 2.3.2). Now, with the knowledge of A_3 and Y_3, the verifier can calculate the path $P_s = (p_0, p_1, p_2, p_3)$, also indicated by the arrows. If the root p_3 calculated by the verifier matches the public key Y_{MSS}, the signature σ_3 is valid.

LMS Parameter	
symbol	meaning
n	security parameter, length of the hash digest
w	Winternitz parameter, $w \in \{1, 2, 4, 8\}$
d	height of Merkle Tree
t	amount of elements/Winternitz chains in a single one-time key
s	next unused W-OTS keypair in Merkle Tree
ℓ	amount of leaves Merkle Tree / amount of one-time keys, $\ell = 2^d$
h	cryptographic secure hash function, see Equation 2.1
X_{LMS}	LMS private key
Y_{LMS}	LMS public key

Table 2.1.: Parameter used for LMS, see Section 2.5 [17].

The verification of signature σ_s is only successful if the root p_d calculated by the verifier matches the public key Y_{MSS}. The one-time key Y_s and therefore the one-time signature σ_s/OTS are implicitly validated, as Y_s is calculated on the way to the root of the Merkle tree by the verifier. This section is also explained in detail with a depiction of the Merkle tree in Figure 2.3.

2.5. Leighton-Micali Signature Scheme (LMS)

The *Leighton-Micali Signature Scheme (LMS)* [17] is basically the Merkle Signature Scheme in combination with W-OTS (see Section 2.4), but the transition from each Winternitz one-time key (which includes several Winternitz chains, see Section 2.3.2) to the leaf of the Merkle Tree is defined in detail. The parameters of LMS are defined in Table 2.1.

2.5.1. LMS Key Generation

The key generation works like for MSS (see Section 2.4): First, the W-OTS parameters w, n are chosen and t is calculated as shown in Section 2.3.2. In difference to MSS, the amount of Winternitz chains for each one-time key can be greater than one ($t \geq 1$, see Section 2.4.1, MSS Key Generation: Special Case). Notably, the LMS RFC [17] states slightly different equations for calculating t, but this is irrelevant for the scope of this work. With the given parameters, $\ell\times$ Winternitz one-time keys $X_i, 0 \leq i \leq \ell - 1$ are randomly generated. These amount to the private key X_{LMS}:

$$X_{LMS} = (X_0, \cdots, X_i, \cdots, X_{\ell-1}) \tag{2.31}$$

To generate the one-time public key $Y_i, 0 \leq i \leq \ell - 1$ out of each Winternitz private key, the hash function f is applied 2^{w-1} times on each Winterintz chain in each $X_i \in X_{LMS}$, see Equation 2.13. Each resulting one-time key $Y_i, 0 \leq i \leq \ell-1$ consists of t Winternitz chains, each of length n (see Equation 2.14). One leaf of the Merkle Tree is generated by concatenating and hashing the Winternitz chains together with function h (see Table 2.1). For a public one-time key Y_i that consists of t Winternitz-chains $y_0, \cdots, y_{t-1}$, the Merkle Tree leaf $h(Y_i)$ is generated as follows:

$$h(Y_i) = h(y_0 \parallel \cdots \parallel y_{t-1}) \tag{2.32}$$

For a depiction of the LMS Merkle Tree leaf generation, see also Figure 2.4.

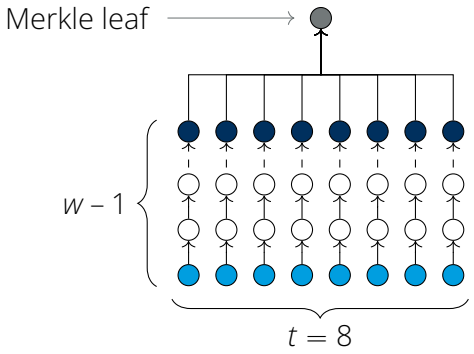


Figure 2.4.: Example depiction for generating the Merkle Tree children out of a single one-time key, with t being the number of Winternitz chains. Each cyan node is the start of one chain, each darkblue node the end. The grey node is a leaf of the LMS Merkle Tree.

The inner nodes and root of the Merkle tree are generated like in MSS (see Equation 2.22, 2.23 and 2.24 respectively). The LMS public key Y_{LMS} is the root of the Merkle Tree.

The amount of hash calls necessary for calculating the Merkle Tree *excluding* the Winternitz chains (i.e. the leaves are already known) is denoted in the following equation:

$$\#\text{ hashcalls Merkle Tree generation} = \ell - 1 \tag{2.33}$$

The amount of hash calls necessary for generating all Merkle Tree leaves is calculated as follows:

$$\#\text{ hash calls leaves generation} = \ell \cdot t \cdot (2^w - 1) \tag{2.34}$$

In summary, the amount of hash calls for generating the public key Y_{LMS} is:

$$\#\text{ hash calls public key generation} = \ell - 1 + \ell \cdot t \cdot (2^w - 1) \tag{2.35}$$

2.5.2. LMS Signature Generation

Signature generation works just like in MSS (see Section 2.4.2), except for the amount of Winternitz chains $t \geq 1$ (see Equation 2.19). The length of the authentication path in LMS can be calculated as follows:

$$\#\text{ elements in authpath} = d = \log_2(\ell) = 1.44 \cdot \log(\ell) \tag{2.36}$$

2.5.3. LMS Signature Verification

Analogously to signature generation, the amount of hash calls for verification is the same as for MSS (see Section 2.4.2), except for the amount of Winternitz chains $t \geq 1$ (see Equation 2.20). The amount of hash calls for signature verification by the verifier are calculated as follows:

$$\#\text{ hash calls verify} = \log_2(\ell) = 1.44 \cdot \log(\ell) \tag{2.37}$$

2.6. Extended Merkle Signature Scheme (XMSS)

The *eXtended Merkle Signature Scheme (XMSS)* [19] is an extension of the Merkle Signature Scheme in combination with W-OTS (see Section 2.4). One of the main advantages of XMSS is that it does not rely on the collision resistance of the used hash functions, but on weaker

properties, namely preimage-resistance, second preimage-resistance. This is achieved by using additional randomly chosen bitmasks for each invocation of the hash function. The Winternitz one-time signature scheme that now includes bitmasks is referred to as *W-OTS+* (see Section 2.6.2). XMSS is a *stateful* signature scheme, so the private key changes after every signature generation. The notation of parameters used for XMSS in this work is shown in Table 2.3. This section is mostly based on the XMSS RFC [19].

2.6.1. Omitting Collision Resistance

Almost all of modern cryptography relies on unproven assumptions. Only a few cryptographic tasks can be achieved with perfect security (e.g. the one-time pad [20]). There clearly is a risk that at least some of these assumptions may be wrong. Therefore, it is important to only make assumptions that are strictly necessary. [21] For all signature schemes shown in this work before, collision resistance of the used hash function is a security requirement (see also Section 2.2). With Grover's algorithm, two ordinary collisions can be found in time $\mathcal{O}(2^{n/3})$, speeding up the classical birthday attack which requires $\mathcal{O}(2^{n/2})$ time (with n being the output length of the specific used hash function) [22, 23]. For maintaining the security of the digital signature system, the requirement for collision resistance is omitted, so the only necessary security assumptions are preimage- and second preimage-resistance of the used hash function. To achieve this, a keyed hash functions (see Equation 2.38) in combination with random bitmasks is used for XMSS.

2.6.2. WOTS+

This section describes the main difference between WOTS+ and WOTS (see Section 2.3.2), an overview is given in Table 2.2. The hash function used for WOTS+ is a *keyed* hash function with *random bitmasks* as additional input for each function call. The rest of WOTS+ works just like WOTS. In general, a keyed hash function h_{keyed} takes a public key K and a message M of arbitrary length and maps it to an output of fixed length n. The public key K is an element of the key space $\mathcal{K} = \{0,1\}^n$, M is an element of the message space $\mathcal{M} = \{0,1\}^*$. [24] In XMSS, the key K corresponds to a public seed *sd*.

$$h_{keyed} : \mathcal{K} \times \mathcal{M} \rightarrow \{0,1\}^n \tag{2.38}$$

The keyed hash function in combination with the bitmasks is denoted as tweakable hash function, a principle introduced by Bernstein et al. [25] and adapted by Campos et al. [26]. The following definitions are based on the work of Campos et al. [26]: Let $\mathcal{K}$ be the keyspace, $\mathcal{T}$ be the tweakspace (containing the bitmasks), $\mathcal{M}$ the message space (containing the possible inputs), $\mathcal{K} = \mathcal{T} = \mathcal{M} = \{0,1\}^n$. Then, a tweakable hash function h_{tweak} maps a key $K \in \mathcal{K}$, a bitmask $T \in \mathcal{B}$, and a message $M \in \mathcal{M}$ to a fixed output of length n. Notably, the bitmask T *changes* after each invocation of h_{tweak}, while the key K (or respectively the public seed *sd*) stays the same.

$$h_{tweak} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \rightarrow \{0,1\}^n \tag{2.39}$$

In detail, the tweakable hash function h_{tweak} works as follows: Let h_1, h_2 be two hash functions.

$$h_1 : \{0,1\}^{2n} \times \{0,1\}^n \rightarrow \{0,1\}^n \tag{2.40}$$

$$h_2 : \{0,1\}^{2n} \rightarrow \{0,1\}^n \tag{2.41}$$

Then, the tweakable hash function h_{tweak} is constructed. For an explanatory depiction of h_{tweak}, see Figure 2.5.

$$h_{tweak}(K,T,M) = h_1(K \parallel T, M^{\oplus}), \text{with } M^{\oplus} = M \oplus h_2(K \parallel T) \tag{2.42}$$

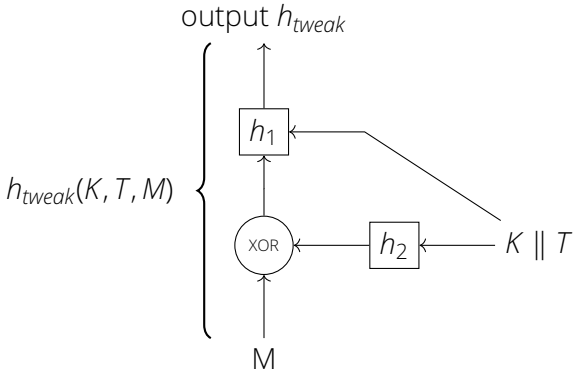


Figure 2.5.: Example depiction of the tweakable hash function h_{tweak} with the input values K, B, M (corresponding to key, bitmask, message). The functions $h1, h2$ are part of h_{tweak} (see Equation 2.42).

WOTS/WOTS+ Parameter		
symbol		meaning
WOTS	WOTS+	
n		security parameter / output length of used hash function
w		Winternitz parameter / blocksize
$b_i \in B$		one block element (of size w)
t	l	amount of elements in private/public key, signature
f	h_{tweak}	used hash function
-	B_{wots+}	bitmasks necessary for h_{tweak}
-	sd	public seed necessary for h_{tweak}

Table 2.2.: Symbols and their meaning used for WOTS (see Section 2.3.2) and WOTS+ (see Section 2.6.2) respectively.

As defined in Equation 2.42, additional distinct random input for each invocation of h_{tweak} is generated by using the output of h_2 as additional input for h_1. For further security details of the tweakable hash function, see Bernstein et al. [25].

WOTS+ Key Generation

Like in W-OTS (see Section 2.3.2), the Winternitz-Parameter w and the security parameter n are selected. In WOTS+, w is an element of the set $\{4, 16\}$, n is the output length of the hash function h_{tweak}. These parameters are used to calculate l_1, l_2, l (corresponding t_1, t_2 and t in WOTS but the calculation is slightly different). The value l_1 determines the amount of blocks the message digest m will be separated into:

$$l_1 = \left\lceil \frac{n}{\log 2w} \right\rceil \tag{2.43}$$

The value l_2 determines the amount of blocks the checksum c will be separated into:

$$l_2 = \left\lfloor \log_2 \frac{l_1(w-1)}{\log_2 w} \right\rfloor + 1 \tag{2.44}$$

The value l determines the amount of elements in the private/public key and the signature:

$$l = l_1 + l_2 \tag{2.45}$$

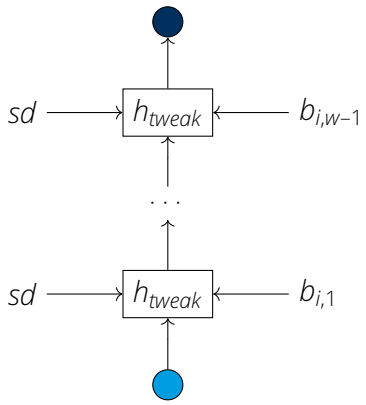


Figure 2.6.: Example WOTS+ key generation for *one* WOTS+ chain (see Figure 2.8 for the WOTS+ chains in scope of the complete XMSS tree). For each hash function call of h_{tweak}, another bitmask $b_{i,j}$ and the same public seed *sd* are used. The value *i* denotes the position of the chain in the one-time key, the value *j* denotes the height in the chain. After $w - 1$ hash function calls applied consecutively on the private key of this chain (cyan node), the public key of this chain (blue node) is generated.

The WOTS+ private key consists of l elements, each of length n, chosen at random.

$$X = (x_0, \cdots, x_{l-1}) \tag{2.46}$$

The WOTS+ public key (see Equation 2.48) is generated by applying the tweakable hash function h_{tweak} (see Equation 2.42) to each element x_i (which can also be seen as the beginning of one WOTS+ chain) in the private key X consecutively for $w - 1$ times. In difference to WOTS, the bitmasks $b_{i,j}$ and the public seed *sd* are additionaly needed for each invocation of the used hash function. Notably, the bitmask $b_{i,j}$ changes for each hash function call, while the seed *sd* stays the same. The index *i* of the bitmask denotes the position of the corresponding key element, the index *j* denotes the height position in the Winternitz chain, B_{wots+} is a set of all bitmasks used for a single WOTS+ key pair.

$$B_{wots+} = (b_{0,0}, \cdots, b_{i,j}, \cdots, b_{l,w-1}) \tag{2.47}$$

$$y_i \in Y = h_{tweak}^{w-1}(sd, b_{i,j}, x_i) \text{ with } j = 0, \cdots, w-1 \tag{2.48}$$

$$Y = (y_0, \cdots, y_{l-1}) \tag{2.49}$$

For an explanatory depiction of the key generation process, see Figure 2.6.

WOTS+ Signature Generation & Verification

The WOTS+ signature generation and verification works just as for WOTS (see Section 2.3.2, WOTS Verification), except that (like for WOTS+ key generation) instead of the function f, the tweakable hash function h_{tweak} is used. The seed *sd* and all bitmasks B_{wots+} (see Equation 2.47) are known to the verifier. Given all necessary parameters, a WOTS+ signature is denoted as follows:

$$\sigma_{wots+} = (\sigma_0, \cdots, \sigma_i, \cdots \sigma_{l-1}) \text{ where } \sigma_i = h_{tweak}^{b_i}(sd, b_{i,j}, x_i) \tag{2.50}$$

XMSS Parameter	
symbol	meaning
l	amount of leaves of the L-Tree / elements in one WOTS+ key
$\lceil log(l) \rceil$	height of one L-Tree in the XMSS tree
d	height of Merkle tree in the XMSS tree
D	height of the complete XMSS tree, $D = \lceil log(l) \rceil + d$
h_{tweak}	tweakable hash function
sd	public seed, key for hash function h_{tweak}
$b_{i,j} \in B_{XMSS}$	bitmask in XMSS tree on position i, j
s_{next}	index of next unused WOTS+ keypair

Table 2.3.: Symbols and parameters used for describing XMSS.

2.6.3. XMSS Key Generation

For a depiction of the XMSS tree referenced in this section, see Figure 2.8.

First, the signer chooses the WOTS+ parameters n, w (see Section 2.6.2/WOTS+ Key Generation). Then, 2^d WOTS+ one-time private keys (see Equation 2.46) are generated, d denotes the height of the Merkle tree inside the XMSS tree. Each leaf of the MSS tree in the XMSS tree is one WOTS+ one-time public key. The leaf index s_{next} denotes the next unused WOTS+ one-time private key (to ensure it is only used once), a seed sd ($K = sd$ when using h_{tweak}, see Equation 2.42).

$$X_{XMSS} = ((X_0, \cdots, X_{2^d-1}), s_{next}, sd) \tag{2.51}$$

The public key Y_{XMSS} consists of the root of the XMSS tree Y_{root} and the public seed sd. The bitmasks B_{xmss} as well as B_{wots+} necessary for building the XMSS tree are already known to the verifier.

$$Y_{XMSS} = (Y_{root}, sd) \tag{2.52}$$

The leaf index s_{next} is initialized to zero when the XMSS private key is created.

As h_{tweak} is used, the bitmasks B_{XMSS} are necessary to generate the whole XMSS tree. One $b_{i,j} \in B_{XMSS}$ corresponds to T when using h_{tweak}, see Equation 2.42), D denotes the height of the complete XMSS tree:

$$\begin{aligned} B_{XMSS} &= (b_{0,0}, \cdots, b_{i,j}, \cdots, b_{D,2^{D-1}}) \\ &0 \leq i \leq D \\ &0 \leq j \leq 2^{D-1} \end{aligned} \tag{2.53}$$

L-Tree

The L-Tree is a concept to combine each WOTS+ one-time key into a leaf of the Merkle tree, see Figure 2.7: It compresses the WOTS+ public key into one value. As h_{tweak} is used, sd and the corresponding bitmasks are also needed for generating the L-Tree.

2.6.4. XMSS Signature Generation

The signature generation works like for MSS in combination with WOTS (see Section 2.4.2), but the used hash function is h_{tweak} with its corresponding inputs sd, B_{XMSS} (see Equation 2.42).

Moreover, the L-Tree structure (see Section 2.6.3/L-Tree) is used to generate the Merkle leaves out of the WOTS+ key pairs. Given the message M, the message digest is computed with

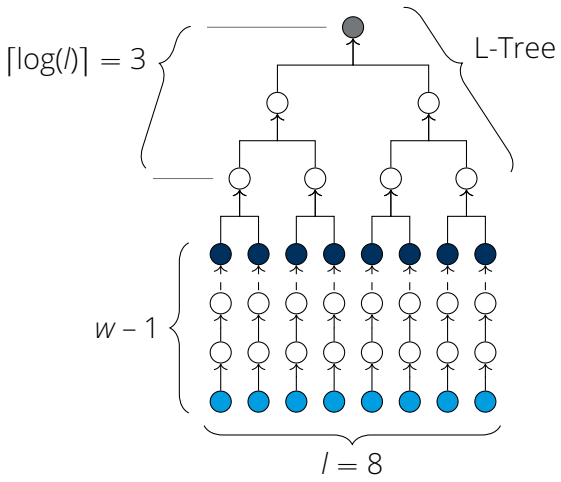


Figure 2.7.: Example depiction of one L-Tree (see Section 2.6.3/L-Tree), the WOTS+ one-time key has $l = 8$ elements. Each cyan node is the beginning of one WOTS+ chain, each blue node denotes the end of one WOTS+ chain. The L-Tree (with height $\lceil \log(l) \rceil = 3$) combines each chain to one Merkle tree child (grey node at the root), see also Figure 2.8.

h_{tweak}. A XMSS signature σ_{xmss} for m contains the WOTS+ signature σ_{wots+}, the authentication path A_s, and the index s (indicates the WOTS+ key pair used for this signature). For a more specific explanation of these parameters, see also Section 2.4.2.

$$\sigma_{XMSS} = (s, \sigma_{wots+}, A_s) \tag{2.54}$$

After signing the message digest m, the index s_{next} of the next unused one-time key pair in the private key X_{XMSS} is updated.

2.6.5. XMSS Signature Verification

The signature verification also works similar to MSS in combination with WOTS (see Section 2.4.3). For an overview of the parameters used for XMSS, see Table 2.3. Like in the steps before, the hash function h_{tweak} is used. Given the public key Y_{XMSS} and signature σ_{XMSS}, the verifier takes $\sigma_{wots+} \in \sigma_{XMSS}$ to calculate the leaves of the L-Tree (or in other words, the public one-time WOTS+ key). Then, the root of the L-Tree or respectively the leaf of the Merkle tree is created (see also Figure 2.7 and Figure 2.8). With $A_s \in \sigma_{XMSS}$, the verifier calculates a path from the Merkle tree leaf to the root of the XMSS tree. Now, the root calculated by the verifier is compared to $Y_{root} \in Y_{XMSS}$. The verification succeeds only if the calculated root matches Y_{root}. For a more specific explanation of the signing process and the used parameters, see also Section 2.4.3.

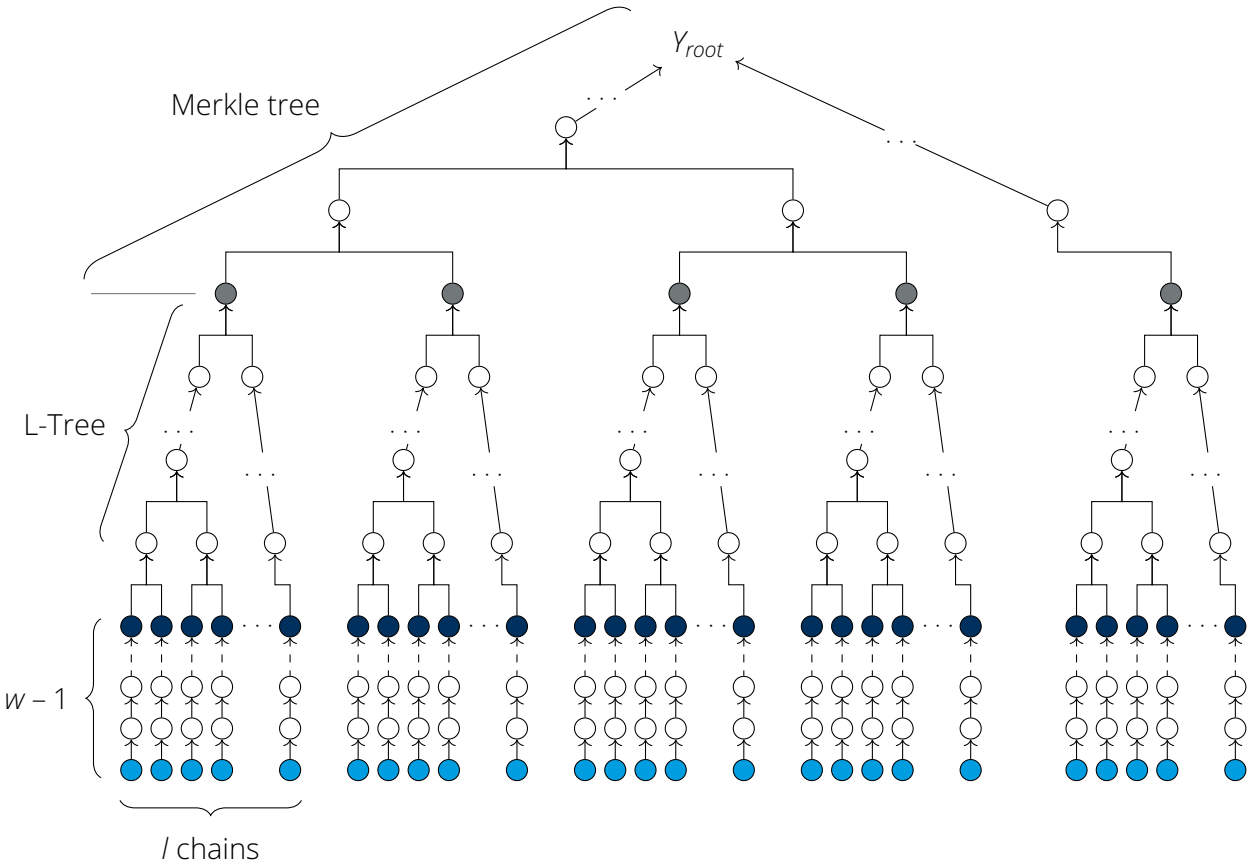


Figure 2.8.: Example depiction of a XMSS tree. The root is the public *XMSS* key Y_{XMSS}, the first d layers are the Merkle tree. The blue nodes are the public keys, the cyan nodes are the private keys of the WOTS+ chains. One leaf of the Merkle tree (grey nodes) or respectively one L-Tree corresponds to a complete one-time WOTS+ public key. The depiction is based on Figure 1 in Campos et al. [26].

3. Related Work

This chapter presents related work regarding quantum-secure *hash-based signature systems (HBS)* and their performance improvements are presented. It will focus on HBS and not on classical digital signature systems such as RSA [4], as these do not fulfill the requirement to be quantum secure [5, 10]. The two categories *stateful*- and *stateless* HBS are introduced. Afterwards, current works improving the performance of HBS are proposed.

3.1. Stateful and Stateless HBS

Most common digital signature systems currently in use are stateless: The signer has one secret key that is used multiple times for signing. In stateful signature schemes the secret key is only used once, therefore referred to as one-time key. If the same key is used more than once, the security will be compromised. To assure that every key is only used once, the signer needs to maintain a state of the current key while signing messages. The state and hence the key are updated after every signature. To generate each one-time key, a one-time signature scheme (OTS) is used (e.g. WOTS and WOTS+, explained in detail in Section 2.3.2 and 2.6.2). A stateful scheme is less practical than a stateless scheme, as its one-time key states require careful treatment. In some situations, managing states is acceptable for meeting other demands: In comparison with other quantum-secure stateless HBS, stateful HBS are more efficient, the signature size is smaller and there are more signing possibilities. [27]

Common stateful HBS are the *Leighton-Micali Signature Scheme (LMS)* [17] and the *eXtended Merkle Signature Scheme (XMSS)* [19]. They are standardized and recommended for usage as quantum-secure HBS by the National Institute of Standards and Technology (NIST) [28]. Both these signature schemes are explained in detail as a part of this work (for LMS see Section 2.5, for XMSS see Section 2.6).

An extension of XMSS is *Multi Tree XMSS (XMSS^{MT})*, a hash-based signature scheme that can be used to sign a larger (but still fixed) number of messages. XMSS^{MT} has a broader possible parameter set and reduced effort in comparison to XMSS. XMSS^{MT} uses a hyper-tree, a tree containing several layers of XMSS trees. The root nodes of the lower layers are signed by the trees on top and intermediate layers. To sign a message, the trees on the lowest layer are used. [29, 19]

3.2. Related Work: HBS

Kampanakis & Fluhrer [30] compare general properties of LMS and XMSS: Security assumptions, signature- and public key sizes as well as computation overhead. They conclude that LMS performs significantly better than XMSS and XMSS (with equivalent parameter sets to LMS) has slightly smaller signature sizes than LMS. Therefore, LMS allows more options for selecting parameter sets that fit the specific use cases.

Oliveira et al. [31] improve the performance of LMS and XMSS by optimizing and therefore speeding up the underlying hash functions (SHA-2 or SHA-23) and other building block functions. This leads to higher performance for signature operations in LMS and XMSS. The results show that both HBS schemes can achieve high performance using vector instructions on modern processors.

Hülsing et al. [32] propose *XMSS+*, an HBS based on XMSS. Compared to XMSS, the key generation time is reduced from $\mathcal{O}(n)$ to $\mathcal{O}(\sqrt{n})$, with n being the number of signatures that can be created with one key pair.

Wang et al. [33] propose a software-hardware co-design for XMSS on a RISC-V embedded processor. The implementation with the best performance generates a key pair in 3.44 seconds, achieving an over 54 times speedup compared to the pure software version of XMSS. Signature generation takes $\leq$ 10ms and verification takes < 6ms for such a key pair, resulting in a speed-up of $\geq$ 42 times and $\geq$ 17 times respectively.

Bos et al. [34] propose a method *Rapidly Verifiable XMSS Signatures* which speeds up the XMSS signature verification. It is based on the PZMCM technique [35], which changes the XMSS signing algorithm to find verifiable signatures: In XMSS, the amount of hash calls for generating a signature and afterwards verifying it always sums up to the same value. This *counter value* is added to the input of the message hash, then T different counter values are tried. Afterwards, the counter value leading to the fastest signature is chosen out of the T possibilities. As a result, verifying signatures is about 1.44 times faster than traditionally generated signatures. The most optimized method reduces verification time by ≥ 2 from 13.85 million to 6.56 million cycles.

Campos et al. [26] compare the performance of different implementations of LMS and XMSS on an ARM Cortex-M4. They propose an optimized implementation of XMSS, which outperforms the original by a factor of 3.11 during key generation and signing, and by 4.32 while verifying.

Bernstein et al. [25] propose *SPHINCS+*, a stateless HBS. It is an improvement of SPHINCS [36]. SPHINCS+ uses a few-time signature scheme (FTS) to sign more than one message. SPHINCS+ replaces the leaf generation by an OTS (like for stateful HBS) with a FTS. The FTS used in SPHINCS is HORS/HORST (HORS with trees), while the FTS used in SPHINCS+ is Forest of Random Subsets (FORS). The idea of SPHINCS+ is to authenticate a huge number of few-time keys using a hyper-tree. The root of the hyper-tree is the public key of the signature system. [25, 37]

Hülsing et al. [38] compare the SPHINCS implementation SPHINCS-256 with XMSS^{MT} on an ARM Cortex M3 micro-controller with a small RAM size of 16KB. They conclude that verification time is fast for both schemes. In XMSS^{MT} signature generation is roughly 32 times faster than producing a SPHINCS-256 signature. They state that this difference is not

comparatively big using SPHINCS-256 and it might be a good trade-off for getting the flexibility provided by a stateless scheme anyway.

4. Methods

In Chapter 2 the digital signature systems LMS and XMSS, based on a classical Merkle Tree, were introduced. Dodis et al. [39] propose a method T_5 that can be used to speed up the classical Merkle Tree. In this chapter the T_5 method is explained in detail, equations for speed-up calculations are introduced, a new method called More Aggressive Opening is proposed and the extended Merkle Tree scheme T_5-Tree$^+$ is constructed. The goal is to substitute the standard Merkle Tree with the T_5-Tree and T_5-Tree$^+$ concepts to speed up LMS, XMSS and potentially other signature schemes based on Merkle Trees.

4.1. T_5 Hashing

Dodis et al. [39] propose a method called T_5 for hashing five inputs with three hash compression calls. The $5n$-to-n compression function T_5 (with n being the hash digest length) is constructed out of $2n$-to-n compression functions h_1, h_2, h_3:

$$T_5(m_1, m_2, m_3, m_4, m_5) = h_3(h_1(m_1, m_2) \oplus m_5, h_2(m_3, m_4) \oplus m_5) \oplus m_5 \tag{4.1}$$

It is proven by Dodis et al. [39] that the T_5 construction matches Stam's bound [40], providing $\tilde{\mathcal{O}}(q^2/2^n)$ collision security and $\mathcal{O}(q^3/2^{2n} + nq/2^n)$, $q \leq 2^{n/2}$ preimage security. It provides birthday security $\mathcal{O}(2^{n/2})$ (see also Section 2.6.1) for hashing five inputs using three $2n$-to-n compression calls, instead of only four inputs in prior constructions. For the full proof of collision resistance and preimage resistance of T_5, see Section 4.1 and Section 4.2 in Brassard et al. [39]. Therefore, T_5 is improving the Merkle-Dåmgard construction (with the initialization vector counted as message block) and Merkle trees by processing a fifth message block with the same number of compression function calls and essentially the same level of collision security. For this work, the construction of T_5 in combination with Merkle trees is of interest.

4.1.1. T_5-Block

In Figure 4.1, the construction of one T_5-Block out of a Merkle tree with height $d = 2$ is shown:

- The variables m_1, m_2 (respectively m_3, m_4) denote the left and right halves of the input to compression function h_1 (respectively h_2).
- The variables a and b denote the output of h_1 and h_2 respectively.
- The variables c and d denote the left and right halves of the input to h_3.

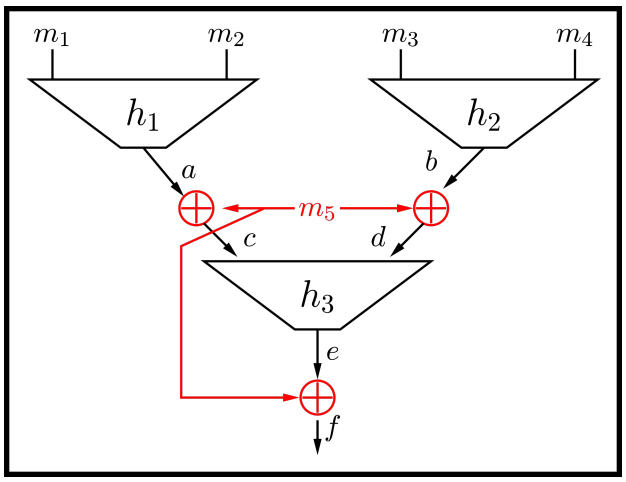


Figure 4.1.: Modified Merkle tree $T_5(m_1, m_2, m_3, m_4, m_5)$ of height d = 2, with an extra input m_5 for the same three hash calls h_1, h_2, h_3. In this work, this is referred to as one T_5-Block. [39]

- The variable e denotes the output of h_3.
- The variable f denotes the total output of $T_5(m_1, m_2, m_3, m_4, m_5)$.

The calculation of $T_5(m_1, m_2, m_3, m_4, m_5)$ consists of the following steps:

1. Calculating of the first layer of one T_5-Block (corresponds to compressing 4 leaves of a binary Merkle tree). Two hash function calls h_1, h_2 are necessary:

$$a = h_1(m_1, m_2) \tag{4.2}$$
$$b = h_2(m_3, m_4) \tag{4.3}$$

2. Calculating of the first T_5-specific intermediate step by adding m_5:

$$c = a \oplus m_5 \tag{4.4}$$
$$d = b \oplus m_5 \tag{4.5}$$

3. Calculating of the second layer of one T_5-Block (i.e. compressing 2 nodes of a binary Merkle tree). One hash function call h_3 is necessary:

$$e = h_3(c, d) \tag{4.6}$$

4. Addition of m_5 (the second T_5 specific intermediate step). This leads to the final result f:

$$T_5(m_1, m_2, m_3, m_4, m_5) = f = e \oplus m_5 \tag{4.7}$$

4.1.2. T_5 Openings

To calculate the authentication path (see also Section 2.4.2) in one T_5-Block, two different approaches Conservative Opening and Aggressive Opening are shown by Dodis et al. [39]. These two versions are depicted in Figure 4.2. The process of calculating an authentication path is referred to as *opening*.

Conservative Opening	
# elements in auth. path	4
# hash calls verify	3

Table 4.1.: Performance of calculating the authentication path for one T_5-Block with Conservative Opening: The necessary amount of hash calls and amount of elements in the authentication path are given.

Aggressive Opening	
# elements in auth. path	3
# hash calls verify	2

Table 4.2.: Performance of calculating the authentication path for one T_5-Block with Aggressive Opening.

Conservative Opening

Given a node $m_i, i \in \{1, 2, 3, 4\}$, the straightforward way to open the T_5-Block is to provide the remaining four nodes $m_{j \neq i}$. This is denoted as Conservative Opening and depicted in Figure 4.2. The current node on the path is m_1 (colored green), the remaining four nodes (colored red) necessary for the authentication path to open m_1 are m_2, m_3, m_4, m_5. The functions h_1, h_2, h_3 (colored red) have to be calculated. The performance of Conservative Opening is less attractive than of the other methods and even worse than using a standard Merkle Tree. This is because for authenticating one T_5-Block, three hash calls and five elements in the authentication path are necessary (see also Table 4.1). Therefore, this method is not further considered in this work.

Aggressive Opening

The second version of opening a T_5-Block with better performance than Conservative Opening is the Aggressive Opening. The provable security bounds decrease for this method but the security remains the same under plausible attack scenarios. It is defined as follows: For a given opening node m_i, where $i \in \{1, 2, 3, 4, 5\}$, the function $open_{aggr}$ returns the authentication path elements for the corresponding T_5-Block:

$$open_{aggr}(m_1) = (m_2, m_5, h_2(m_3, m_4) \oplus m_5) = (m_2, m_5, d) \tag{4.8}$$
$$open_{aggr}(m_2) = (m_1, m_5, h_2(m_3, m_4) \oplus m_5) = (m_1, m_5, d) \tag{4.9}$$
$$open_{aggr}(m_3) = (m_4, m_5, h_1(m_1, m_2) \oplus m_5) = (m_4, m_5, c) \tag{4.10}$$
$$open_{aggr}(m_4) = (m_3, m_5, h_1(m_1, m_2) \oplus m_5) = (m_3, m_5, c) \tag{4.11}$$
$$open_{aggr}(m_5) = (m_1, m_2, h_2(m_3, m_4) \oplus m_5) = (m_1, m_2, d) \tag{4.12}$$

In Figure 4.2, Aggressive Opening is depicted as example for opening node m_1 (colored green, see Equation 4.8). As shown for all opening nodes in Equations 4.8-4.12, the authentication path always contains three elements and the verifier needs two hash calls (h_1 or h_2 and always h_3) to calculate the "root" f of one T_5-Block, see Table 4.2.

4.1.3. T_5-Tree

Given the T_5-Block construction in Section 4.1.1, a complete T_5 Merkle Tree, denoted as *T_5-Tree*, can be built. This is also shown by Dodis et al. [39] (see Figure 4.3). Notably, the

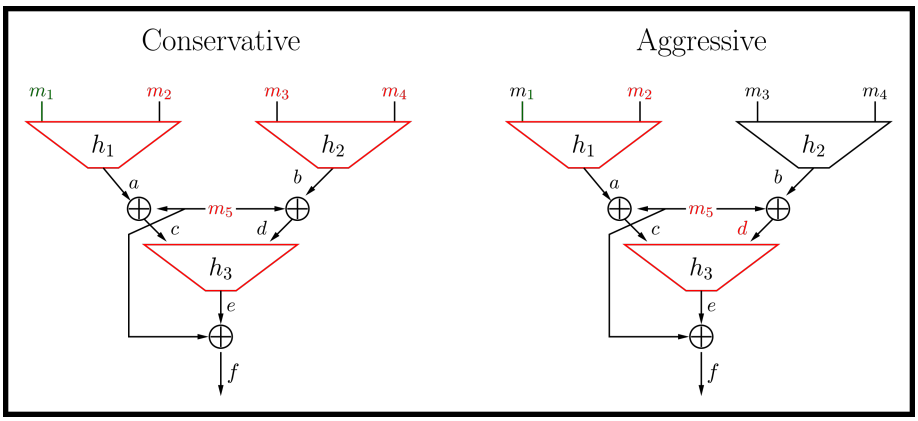


Figure 4.2.: Conservative and Aggressive Opening of one T_5-Block. The green variable denotes the opening node, the red nodes are given in the authentication path for this T_5-Block. The hash functions denoted in red have to be calculated by the verifier to get the path for this T_5-Block. [39]

compression functions h_1, h_2 and h_3 are the same function, the different indices are used for readability. The performance of the T_5-Tree (including Conservative- and Aggressive Opening variants) in comparison to the standard Merkle Tree is shown in Table 4.3. *Build calls* denote the amount of hash calls necessary for building the whole T_5 Merkle Tree, *opening* denotes the length of the authentication path, *verify* denotes the amount of hash calls needed to build the path from authentication path). *Tree depth* is the amount of layers in the binary Merkle tree compared with the amount of layers in T_5-Tree. This corresponds to twice the amount of T_5-Blocks because one T_5-Block corresponds to two Merkle tree layers (see also Figure 4.1).

In order to compare the performance of the T_5-Tree to other tree variants (like binary Merkle Tree and T_5-Tree$^+$, see Chapter 5), performance metrics are constructed. The amount of hash calls for constructing the whole T_5-Tree and the T_5-Tree$^+$ are identical (for a detailed derivation of the formula see Section 4.2.2 and Equation 4.28):

$$\#\text{ hash calls tree gen. } T_5\text{-Tree} = \frac{3}{4}(\ell - 1) \tag{4.13}$$

Let ℓ be the amount of leaves of the T_5-Tree, the opening method is Aggressive Opening. Then, the length of the authentication path is calculated as follows:

$$\#\text{ el. in auth.path } T_5\text{-Tree} = 3\log_5(\ell) = \frac{3\log(\ell)}{\log(5)} = 1.86\log(\ell) \tag{4.14}$$

The amount of hash calls to calculate the path through the whole T_5-Tree (i.e. signature verification), given the authentication path (see Equations 4.8-4.12):

$$\#\text{ hash calls path generation } T_5\text{-Tree} = 2\log_5(\ell) = 2 \cdot \frac{\log(\ell)}{\log(5)} = 1.24\log(\ell) \tag{4.15}$$

4.2. Extended T_5-Tree: T_5-Tree$^+$

In this section, the idea of the T_5-Tree is extended by the opening method *More Aggressive Opening*. When Aggressive Opening is in the whole T_5 Merkle tree, the tree construction is denoted as *T_5-Tree$^+$*. As a next step, the performance of the T_5-Tree$^+$ is compared to T_5-Tree

T_5-Tree Performance			
	Merkle Tree (standard)	Conserv. Opening	Aggr. Opening
build calls/t	1	0.75	0.75
tree depth/ $\log_2(t)$	1	0.86	0.86
verify (hash calls)/ $\log_2(t)$	1	1.29	0.86
opening (length)/ $\log_2(t)$	1	1.72	1.29

Table 4.3.: Performance of the T_5-Tree with the opening variants Conservative- and Aggressive Opening in comparison to the standard Merkle Tree given by Dodis et al. [39]. The variable t denotes the amount of children.

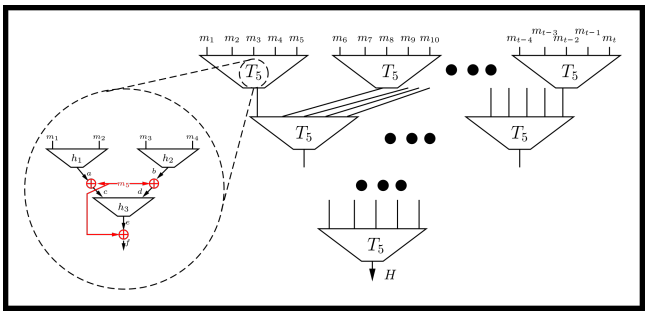


Figure 4.3.: Construction of a complete T_5-Tree consisting of several T_5-Blocks [39]. The value H denotes the root of the tree.

with Aggressive Opening of Dodis et al. [39]. For a Python implementation of the T_5-Tree$^+$ as digital signature scheme that contains the steps from key generation to signature verification, see Appendix A. The parameters used for T_5-Tree$^+$ construction are denoted in Table 4.4.

4.2.1. More Aggressive Opening

In this work, the new opening method *More Aggressive Opening* is proposed (see Figure 4.4). It has better overall performance than Conservative and Aggressive Opening (see Section 4.1.2 and 4.1.2). For the signing and verifying process, see Table 4.5. Notably, the length of the authentication path is **not** constant.

Signature Generation: T_5-Block

When calculating the authentication path (generating the signature) for one T_5-Block, two cases can be distinguished:

- **Case 1**: The possible opening nodes are $m_i, i \in \{1, 2, 3, 4\}$. For a given opening node m_i,

T_5-Tree$^+$ Parameters	
symbol	meaning
T	height of the tree in T_5-Blocks, $T = \log_5(\ell)$
ℓ	amount of leaves, $\ell = 5^T$
d	height of the tree in actual Merkle nodes, $d = 3T$

Table 4.4.: The parameters of T_5-Tree$^+$. One T_5-Block contains three Merkle layers, therefore $d = 3T$ (see also Figure 4.1).

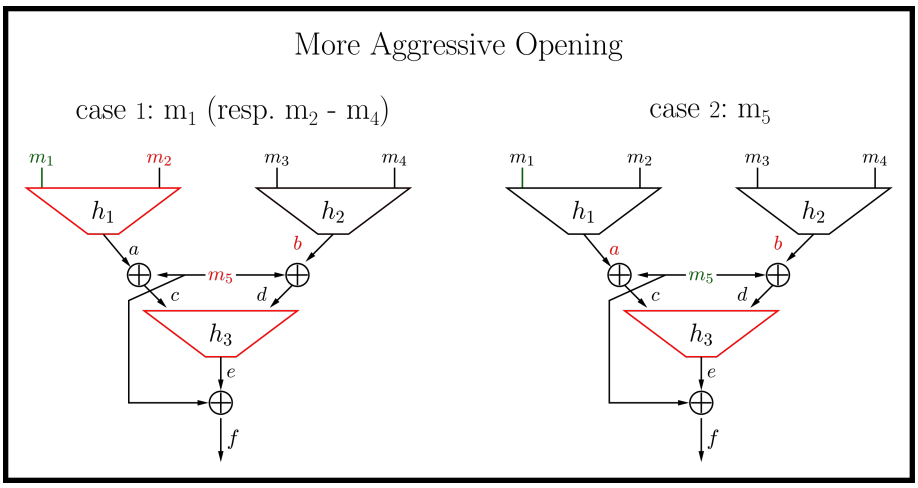


Figure 4.4.: More Aggressive Opening for one T_5-Block, with case differentiation. The green variable denotes the opening node, the red nodes are given in the authentication path the respective T_5-Block. The hash functions denoted in red have to be calculated by the verifier to get the path for this T_5-Block.

the function $open_{aggr+}$ returns the authentication path for the corresponding T_5-Block:

$$open_{aggr+}(m_1) = (m_2, m_5, h_2(m_3, m_4)) = (m_2, m_5, b) = A_{m_1+} \tag{4.16}$$
$$open_{aggr+}(m_2) = (m_1, m_5, h_2(m_3, m_4)) = (m_1, m_5, b) = A_{m_2+} \tag{4.17}$$
$$open_{aggr+}(m_3) = (m_4, m_5, h_1(m_1, m_2)) = (m_4, m_5, a) = A_{m_3+} \tag{4.18}$$
$$open_{aggr+}(m_4) = (m_3, m_5, h_1(m_1, m_2)) = (m_3, m_5, a) = A_{m_4+} \tag{4.19}$$

An example of m_1 as opening node is depicted on the left side of Figure 4.4: The signer is putting the elements m_2, m_5 and b in the authentication path (see Equation 4.16). The verifier needs two hash calls (h_1, h_3) to calculate the path to the root f. Therefore, the authentication path has always three elements and the verifier always needs two hash calls to calculate the path to the root of one T_5-Block in case 1 (see also Table 4.5).

- **Case 2**: The opening node is m_5. This case is depicted on the right side of Figure 4.4. For m_5 as opening node, the function $open_{aggr+}$ returns the authentication path for the corresponding T_5-Block:

$$open_{aggr+}(m_5) = (h_1(m_1, m_2), h_2(m_3, m_4)) = (a, b) = A_{m_5+} \tag{4.20}$$

An example of m_5 as opening node is depicted on the right side of Figure 4.4: The signer is putting the elements a and b in the authentication path (see Equation 4.20). The verifier needs one hash call h_3 to calculate the path to the root f. Therefore, the authentication path always has two elements and the verifier always needs one hash call to calculate the path to the root of one T_5-Block in case 2, see also Table 4.5.

Signature Verification: T_5-Block

When calculating the path (i.e. verifying the signature) of one T_5-Block, the same two cases as for signature generation are distinguished (see Equation 4.16-4.19 and Equation 4.20). For an explanatory depiction of the two cases, see Figure 4.4.

- **Case 1**: The possible leaves of the T_5-Block, for which a path to the root f is calculated by the verifier, are $m_i, i \in \{1, 2, 3, 4\}$. For leaf m_i and the corresponding authentication path

More Aggressive Opening		
	case 1: $m_i, i \in [1,4]$	case 2: m_5
# el. in auth. path	3	2
# hash calls verify	2	1

Table 4.5.: Performance of calculating the authentication path and amount of elements in the authentication path for More Aggressive Opening. Case 1 with $m_i, i \in \{1,2,3,4\}$ as possible opening/path nodes and case 2 with m_5 as opening/path node are shown.

A_{m_i+} (given by the signer, see Equation 4.16-4.19), the function $path(m_i, A_{m_i+})$ returns the root f for the corresponding T_5-Block:

$$path(m_1, A_{m_1+}) = path(m_1, (m_2, m_5, b)) = h_3(h_1(m_1, m_2) \oplus m_5, b \oplus m_5) \oplus m_5 = f \quad (4.21)$$
$$path(m_2, A_{m_2+}) = path(m_2, (m_1, m_5, b)) = h_3(h_1(m_1, m_2) \oplus m_5, b \oplus m_5) \oplus m_5 = f \quad (4.22)$$
$$path(m_3, A_{m_3+}) = path(m_3, (m_4, m_5, a)) = h_3(a \oplus m_5, h_2(m_3, m_4) \oplus m_5) \oplus m_5 = f \quad (4.23)$$
$$path(m_4, A_{m_4+}) = path(m_4, (m_3, m_5, a)) = h_3(a \oplus m_5, h_2(m_3, m_4) \oplus m_5) \oplus m_5 = f \quad (4.24)$$

- **Case 2**: Path calculation for m_5 as "leaf" of the T_5-Block. Given m_5 and the corresponding authentication path A_{m_5+} (by the signer, see Equation 4.20), f is calculated by $path(m_5, A_{m_5+})$:

$$path(m_5, A_{m_5+}) = path(m_5, (a, b)) = h_3(a \oplus m_5, b \oplus m_5) \oplus m_5 = f \quad (4.25)$$

The process of tree generation, signing and verifying using Aggressive Opening in the T_5-Tree$^+$ is shown in the following section.

4.2.2. T_5-Tree$^+$ Generation

The T_5-Tree$^+$ generation works like tree generation for the T_5-Tree (see Section 4.1.3, Section 2.4.1 and Figure 4.3). The implementation of the T_5-Tree$^+$ construction in Python is shown in Appendix A. The performance equations for building an T_5-Tree$^+$ are derived as follows (see also Table 4.4). The total number of T_5-Blocks in one T_5-Tree$^+$ is:

$$\#T_5\text{-Blocks} = \sum_{i=0}^{T-1} 5^i \quad (4.26)$$

For one T_5-Block, three hash calls are necessary (see Section 4.1.1). Therefore, the total amount of hash calls to build a T_5-Tree$^+$ based on T is:

$$\#\text{ hash calls tree gen. (depending on } T) = 3 \cdot \sum_{i=0}^{T-1} 5^i \quad (4.27)$$

To get the hash calls for tree generation based on the leaves ℓ, T is substituted by $\log_5(\ell)$.

$$\#\text{ hash calls tree gen. (depending on } \ell) = 3 \cdot \sum_{i=0}^{\log_5(\ell)-1} 5^i = 3 \cdot \sum_{i=0}^{\frac{\log(\ell)}{\log(5)}-1} 5^i = \frac{3}{4}(\ell - 1) \quad (4.28)$$

4.2.3. T_5-Tree$^+$ Signature Generation

The signature generation for the T_5-Tree$^+$ works as follows: After calculating the complete T_5-Tree$^+$, the signer first calculates the T_5-Path (see Appendix A, line 17, `def calc_t5_path` $(\cdots)$). Now, the signer calculates the authentication path from the T_5-Path (see Appendix A, line 167, `def calc_auth_path` $(\cdots)$ and Section 2.4.2).

For this version of the T_5-Tree$^+$, it is assumed that the signer saves the whole tree after key generation. Therefore, no additional hash calls are necessary for generating the authentication path. As T_5-Tree$^+$ uses More Aggressive Opening, the length of the authentication path differs depending on the case (see Section 4.2.1). For **case 1**, there are always three elements in the authentication path of one T_5-Block (see Equations 4.16-4.19 and Table 4.5). For **case 2**, there are always two elements in the authentication path (see Equation 4.20). The average number of signatures in one T_5-Block (in one T_5-Tree$^+$) is $\frac{4}{5}$ for case 1 and $\frac{1}{5}$ for case 2. Therefore, the average number of elements in the authentication path for one T_5-Block is calculated as follows:

$$\#\text{ elements auth.path } T_5\text{-Block (avg.)} = 3 \cdot \frac{4}{5} + 2 \cdot \frac{1}{5} = 2.8 \tag{4.29}$$

The height of the T_5-Tree$^+$ (in T_5-Blocks) is $\log_5(\ell)$ (see Table 4.4). Therefore, the average authentication path length for the whole T_5-Tree$^+$ is calculated as follows:

$$\#\text{ elements in auth.path} = 2.8 \cdot \log_5(\ell) = 2.8 \cdot \frac{\log(\ell)}{\log(5)} = 1.74 \cdot \log(\ell) \tag{4.30}$$

4.2.4. T_5-Tree$^+$ Signature Verification

After receiving the authentication path by the signer, the verifier calculates the path through the T_5-Tree$^+$ to its root (see Appendix A, line 28, `def calc_path_verifier` $(\cdots)$). The verifier compares the resulting root with the public key of the signer. If they match, the signature is valid (see Section 2.4.3). For More Aggressive Opening, the calculation of the path for one T_5-Block of the T_5-Tree$^+$ needs two has calls in **case 1** (see Equations 4.21-4.24) and one hash call in **case 2** (see Equations 4.25).

For **case 1**, the average number of hash calls is $\frac{4}{5}$ and for **case 2** the average number of hash calls is $\frac{1}{5}$ for one T_5-Block. Therefore, the average amount of hash calls necessary for calculating the root f of one T_5-Block is calculated as follows:

$$\#\text{ hash calls path calculation} = \frac{4}{5} \cdot 2 + \frac{1}{5} = 1.8 \tag{4.31}$$

The height of the T_5-Tree$^+$ (in T_5-Blocks) is $\log_5(\ell)$ (see Table 4.4). Therefore, the average amount of hash calls to calculate the root of T_5-Tree$^+$ with the given authentication path, is calculated as follows:

$$\#\text{ hash calls verify } T_5\text{-Tree}^+ = 1.8 \log_5(\ell) = 1.8 \cdot \frac{\log(\ell)}{\log(5)} = 1.12 \cdot \log(\ell) \tag{4.32}$$

5. Evaluation

In Chapter 2, the two most common quantum-secure hash-based signature systems based using a binary Merkle Trees are explained: LMS (see Section 2.5) and XMSS (see Section 2.6). In Chapter 4, the concepts T_5-Tree (see Section 4.1.3) and T_5-Tree$^+$ (see Section 4.2) are proposed. In this chapter, the different tree concepts are compared in general and for specific use cases: The performance of T_5-Tree and T_5-Tree$^+$ is compared with the standard Merkle Tree (see Section 5.1). The speedup of using T_5-Tree and T_5-Tree$^+$ in LMS for key generation, signing and verification follows in Section 5.2.

5.1. Performance Comparison

The general performance of the different tree concepts depending on the leaves ℓ is calculated with the equations in Table 5.1. The derivation of each formula is referenced in the column *Source*.

Notably, these formulas do not take into consideration that, depending on the tree concept, the leaves have a power of two or five. Still, it is possible to derive the differences in performance for each tree concept, as it is shown in Table 5.2. For key generation (i.e. tree generation), both T_5-Tree and T_5-Tree$^+$ outperform the Merkle Tree by using 25% fewer hash calls. For verification, T_5-Tree$^+$ outperforms the other concepts by using 22% less hash calls than the Merkle Tree. For signature generation, the length of the authentication path increases for both T_5-Tree and T_5-Tree$^+$: T_5-Tree has the longest authentication path (29% longer in comparison to Merkle Tree), T_5-Tree$^+$ has a 20% longer authentication path than the Merkle Tree.

Summary: Equations Performance Calculation				
	Merkle Tree	T_5-Tree Aggr.	T_5-Tree$^+$ More Aggr.	Source
# hash calls keygen	$\ell - 1$	$\frac{3}{4}(\ell - 1)$		Eq. 2.33, 4.28
# hash calls verify	$1.44 \log(\ell)$	$1.24 \log(\ell)$	$1.12 \log(\ell)$	Eq. 2.37, 4.15, 4.32
# el. in auth. path	$1.44 \log(\ell)$	$1.86 \log(\ell)$	$1.74 \log(\ell)$	Eq. 2.36, 4.14, 4.30

Table 5.1.: Performance of the standard Merkle Tree (used in LMS) and T_5-Tree$^+$ with the opening variants Aggressive Opening (Aggr.) and More Aggressive Opening (More Aggr.) (see Sections 4.2.1 and 4.1.2 respectively). The variable ℓ denotes the amount of leaves.

General Performance Comparison		
	T_5-Tree	T_5-Tree$^+$
hash calls: key gen.	–25%	
hash calls: verify	–14%	–22%
length auth. path	+29%	+20%

Table 5.2.: Performance of T_5-Tree and T_5-Tree$^+$ in comparison to the Merkle Tree: The amount of hash calls for tree generation and verification decreases for T_5-Tree and T_5-Tree$^+$ respectively, while the length of the authentication path increases.

NIST Parameter Set, LMS				
Parameter Set Name	Numeric Identifier	n	d	ℓ
LMS_SHA256_M32_H5	0x00000005	32	5	32
LMS_SHA256_M32_H10	0x00000006	32	10	1024
LMS_SHA256_M32_H15	0x00000007	32	15	32768
LMS_SHA256_M32_H20	0x00000008	32	20	1048576
LMS_SHA256_M32_H35	0x00000009	32	25	33554432

Table 5.3.: NIST SHA-256 parameter sets for LMS. [28]. The variable n denotes the number of bytes associated with each node in the (standard binary) Merkle tree, the parameter d denotes the height and the parameter ℓ the leaves of the Merkle Tree.

5.2. LMS Parameter Set

As LMS is based on a Merkle Tree (see Section 2.5), it is easily possible to replace the Merkle Tree in LMS with either the T_5-Tree or the T_5-Tree$^+$. Notably, this speedup is independent of the Winternitz parameters, as it only changes the tree structure of LMS, not the leaf generation. There exist standardized sets of values that are used for the proposed signature systems. One common example is the LMS SHA-256 parameter set by the National Institute of Standards and Technology (NIST) [28]: It is denoted in Table 5.3.

5.2.1. NIST Parameter Adaption

When used as a digital signature scheme, the leaves of the Merkle Tree, T_5-Tree and T_5-Tree$^+$ correspond to the amount of one-time keys. For this evaluation, we assume each leaf contains a one-time public key (i.e. there are no empty nodes). When comparing the performance of the standard Merkle Tree with T_5-Tree or T_5-Tree$^+$ based on the amount of leaves ℓ, it is not possible to get the same amount of leaves for each concept, because the leaves of a perfect Merkle Tree are always a power of two, whereas the leaves of a perfect T_5-Tree / T_5-Tree$^+$ are always a power of five.

In order to still get a similar amount of leaves, all $2^d, d \in \{5, 10, 15, 20, 25\}$ are paired with their lower and upper closest power of 5: These *upper bounds* and *lower bounds* for a given 2^d are calculated as $5^{\lfloor \log_5(2^d) \rfloor}$ and $5^{\lceil \log_5(2^d) \rceil}$ respectively. The results are shown in Table 5.4.

5.2.2. LMS Parameter Results

In this section, the NIST LMS parameters are inserted into the equations for evaluation (see Table 5.1) to get tangible results. This performance calculation is implemented in the Python

Evaluation Results: Lower / Upper Bound T_5-Tree / T_5-Tree$^+$				
	lower / upper bound: ℓ	Tree Generation (# hash calls)	Auth.path Length aggr. / more aggr. (# el. auth. path)	Verify aggr. / more aggr. (# hash calls)
2^5	$\rightarrow 5^{\lfloor \log_5(2^5) \rfloor} = 5^2$	18	6 / 6	4 / 4
	$\rightarrow 5^{\lceil \log_5(2^5) \rceil} = 5^3$	93	9 / 8	6 / 5
2^{10}	$\rightarrow 5^{\lfloor \log_5(2^{10}) \rfloor} = 5^4$	468	12 / 11	8 / 7
	$\rightarrow 5^{\lceil \log_5(2^{10}) \rceil} = 5^5$	2343	15 / 14	10 / 9
2^{15}	$\rightarrow 5^{\lfloor \log_5(2^{15}) \rfloor} = 5^6$	11718	18 / 17	12 / 11
	$\rightarrow 5^{\lceil \log_5(2^{15}) \rceil} = 5^7$	58593	21 / 20	14 / 13
2^{20}	$\rightarrow 5^{\lfloor \log_5(2^{20}) \rfloor} = 5^8$	292968	24 / 22	16 / 14
	$\rightarrow 5^{\lceil \log_5(2^{20}) \rceil} = 5^9$	1464843	27 / 25	18 / 16
2^{25}	$\rightarrow 5^{\lfloor \log_5(2^{25}) \rfloor} = 5^{10}$	7324218	30 / 28	20 / 18
	$\rightarrow 5^{\lceil \log_5(2^{25}) \rceil} = 5^{11}$	36621093	33 / 31	22 / 20

Table 5.4.: In this table, the evaluation results of lower / upper T_5-Tree, T_5-Tree$^+$ for tree generation (key generation), length of the authentication path (signing) and path generation (verify) are shown. The first column shows the Merkle leaves predefined by the NIST SHA-256 parameter set (see Table 5.3), the second column contains leaves of the upper / lower bound T_5-Tree, T_5-Tree$^+$ corresponding to the predefined Merkle Tree.

script `performance_evaluation.py`, see Appendix B. The evaluation results for the Merkle Tree are shown in Table 5.5, for the lower/upper bound T_5-Tree and T_5-Tree$^+$ in Table 5.4.

The performance for tree generation is additionally depicted in Figure 5.1: Notably, T_5-Tree and T_5-Tree$^+$ have the same performance for tree calculation, therefore they are not mentioned separately in Figure 5.1. The height d of the Merkle Tree is directly based on the LMS parameter set (see Table 5.3). The height d of the lower- and upper bound T_5-Tree is derived from the LMS parameter set (see Table 5.4). The equations used for calculating the tree generation performance are shown in Table 5.1. Please note that the scales on each axis are logarithmic for better readability.

Evaluation Results NIST: Merkle Tree			
Leaves ℓ	Tree Generation (# hash calls)	Auth.path Length (# el. auth. path)	Verification (# hash calls)
2^5	31	5	5
2^{10}	1023	10	10
2^{15}	32767	15	15
2^{20}	1048575	20	20
2^{25}	33554431	25	25

Table 5.5.: Evaluation results for the standard Merkle Tree, given the NIST SHA-256 parameter sets as number of leaves ℓ (see Table 5.3). The results contain the number of hash calls for tree generation and verification, as well as the length of the authentication path given the number of leaves ℓ.

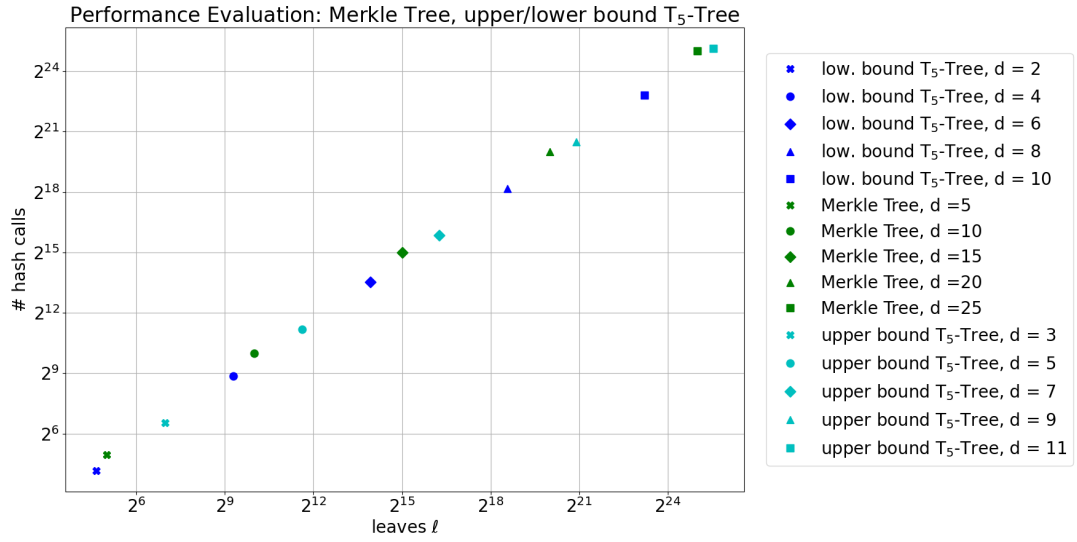


Figure 5.1.: Amount of hash calls for tree generation of Merkle Tree, upper/lower T_5-Tree (inlcudes T_5-Tree$^+$, not mentioned separately). The height d of the Merkle Tree is based on the LMS parameter set (see Table 5.3). The height d of the upper- and lower bound T_5-Tree is derived from the LMS parameter set, see Table 5.4.

6. Conclusion

In this work, the goal was to analyze quantum-secure hash-based signature systems (HBS) in detail and to find opportunities for performance improvements and implementing them. To solve this task, the T_5-Method of Dodis et al. [39] is used to introduce the tree concepts T_5-Tree (see Section 4.1.3) and T_5-Tree$^+$ (see Section 4.2). It is shown that the Merkle Tree in LMS (and potentially XMSS and other signature schemes) can be substituted with these alternate tree concepts.

6.1. Discussion

Both T_5 tree concepts outperform the classical Merkle Tree in key generation and verification time when used in LMS. T_5-Tree needs 25% fewer hash calls for key generation and 14% fewer hash calls for verification. T_5-Tree$^+$ also reduces the amount of hash calls for key generation by 25% and achieves the best result for verification time with 22% fewer hash calls. The length of the authentication path increases for T_5-Tree and T_5-Tree$^+$ by 29% and 20% respectively. The length of the authentication path for T_5-Tree$^+$ is not constant. If a constant length is needed, T_5-Tree is the better choice though T_5-Tree$^+$ has better performance. Notably, key generation time increases exponentially, whereas verification time and length of the authentication path increases linear (dependent on the tree height). Therefore, the worse performance for authentication path length does not have as much impact.

6.2. Future Work

XMSS was inspected in detail in this work, but inserting the T_5 trees into XMSS is still an open task. As XMSS uses bitmasks in its Merkle Tree, the T_5 tree concepts would need to be adapted. One idea how to achieve this, is using the bitmasks as node m_5 for one T_5-Tree, as the bitmasks in XMSS are also inserted to the tree via XOR operations. Another idea worth exploring is adapting T_5-Tree and T_5-Tree$^+$ for not using each leaf as a one-time key (i.e. not making them perfect trees). This would lead to a broader variety of possible one-time keys: With the current concept, the possible amount of one-time keys has to be a power of five. This could be achieved by not calculating each sub-tree of a T_5-Tree, but only parts of it. To distribute the time and space effort between signer and verifier, the signing and authentication operations for one T_5-Block could be adapted: For example if the signer has more computing resources, the signer could directly calculate c, d (of one T_5-Block) instead of a, b (see Figure 4.4). As a result, the signer computes the two XOR computations

during signing instead of the verifier during verification. Finally, speeding up a stateless HBS with the T_5 methods is also achievable: For SPHINCS+ exists an instantiation denoted as SPHINCS+-simple [41], which contains a standard Merkle Tree. This makes inserting T_5-Tree and T_5-Tree$^+$ easily possible.

A. Python Implementation: Extended T5 Tree

The python implementation of T_5-*Tree*$^+$ (see also Section 4.2) is shown in the following python script `T5Tree.py`:

```
from __future__ import annotations
import math
from abc import ABC, abstractmethod
from typing import List, Optional


def hash_t5(left_input: int, right_input: int):
    return hash((left_input, right_input))


def xor(left_input: int, right_input: int):
    return left_input ^ right_input


# necessary/pre-step for auth.path calculation
# != auth.path, tree_height = T5Block Tree height, != path calculated by verifier
def calc_t5_path(ot_key_pos: int, tree_height: int):
    path_list = []
    i = ot_key_pos
    for _ in range(tree_height):
        j = i % 5
        path_list.insert(0, j)  # insert in 1st position of list
        i = (i - j) // 5  # relative position in T5 Block
    return path_list  # list: from root to leaf


# path calculated by verifier
def calc_path_verifier(auth_path: List[int], key_pos: int, child_count: int, one_time_signature: int):
    # count hashcalls for authentication
    hash_count = 0
```

```
31
32      # calc depth of tree
33      tree_depth = int(math.log(child_count, 5))
34      path_t5 = calc_t5_path(key_pos, tree_depth)
35      last_result = one_time_signature  # later: last_result == output of leaf
        ↪ before
36
37      for index in reversed(path_t5):  # reverse because path calc is bottom
        ↪ to root; index == child pos. in T5 Block
38          if index == 0:  # case m1
39              m2 = auth_path[-3]
40              h2 = auth_path[-2]
41              m5 = auth_path[-1]
42              h1 = hash_t5(last_result, m2)
43              h1_xor = xor(h1, m5)
44              h2_xor = xor(h2, m5)
45              h3 = hash_t5(h1_xor, h2_xor)
46              last_result = xor(h3, m5)  # end result, h3 XOR m5
47              auth_path = auth_path[:-3]  # remove already used elements of
                ↪ authpath
48              hash_count += 2  # 2 hash calls were used
49
50          elif index == 1:  # case m2
51              m1 = auth_path[-3]
52              h2 = auth_path[-2]
53              m5 = auth_path[-1]
54              h1 = hash_t5(m1, last_result)  # last_result == m2
55              h1_xor = xor(h1, m5)
56              h2_xor = xor(h2, m5)
57              h3 = hash_t5(h1_xor, h2_xor)
58              last_result = xor(h3, m5)  # end result, h3 XOR m5
59              auth_path = auth_path[:-3]  # remove already used elements of
                ↪ authpath
60              hash_count += 2  # 2 hash calls were used
61
62          elif index == 2:  # case m3
63              m4 = auth_path[-3]
64              h1 = auth_path[-2]
65              m5 = auth_path[-1]
66              h2 = hash_t5(last_result, m4)  # last_result == m3
67              h1_xor = xor(h1, m5)
68              h2_xor = xor(h2, m5)
69              h3 = hash_t5(h1_xor, h2_xor)
70              last_result = xor(h3, m5)  # end result, h3 XOR m5
71              auth_path = auth_path[:-3]  # remove already used elements of
                ↪ authpath
72              hash_count += 2  # 2 hash calls were used
73
74          elif index == 3:  # case m4
75              m3 = auth_path[-3]
76              h1 = auth_path[-2]
```

```
77              m5 = auth_path[-1]
78              h2 = hash_t5(m3, last_result)  # last_result == m4
79              h1_xor = xor(h1, m5)
80              h2_xor = xor(h2, m5)
81              h3 = hash_t5(h1_xor, h2_xor)
82              last_result = xor(h3, m5)  # end result, h3 XOR m5
83              auth_path = auth_path[:-3]  # remove already used elements of
                ↪ authpath
84              hash_count += 2  # 2 hash calls were used
85
86          elif index == 4:  # case: m5
87              # here: last_result == m5
88              # other possibility: auth_path[-3] has padding with 0 (to get
                ↪ same authpath length for every case)
89              h1 = auth_path[-2]
90              h2 = auth_path[-1]
91              h1_xor = xor(h1, last_result)
92              h2_xor = xor(h2, last_result)
93              h3 = hash_t5(h1_xor, h2_xor)
94              last_result = xor(h3, last_result)
95              auth_path = auth_path[:-2]  # remove already used elements of
                ↪ authpath
96              hash_count += 1  # 1 hashcall is used for special case 5
97
98      return last_result, hash_count  # return root, amount of used hash calls
99
100
101 class T5Node(ABC):  # ABC == abstract class
102     def __init__(self):
103         pass
104
105     @abstractmethod
106     def calc_end_hash(self) -> int:
107         pass
108
109     @abstractmethod
110     def get_hash_count(self) -> int:
111         pass
112
113     @abstractmethod
114     def calc_auth_path(self, path: List[int]) -> List[int]:
115         pass
116
117     @abstractmethod
118     def get_child_count(self) -> int:
119         pass
120
121
122 class T5Block(T5Node):
123     def __init__(self, m1: T5Node, m2: T5Node, m3: T5Node, m4: T5Node, m5:
        ↪ T5Node):
```

```
124             super().__init__()  # init T5Node class
125             self.m1: T5Node = m1
126             self.m2: T5Node = m2
127             self.m3: T5Node = m3
128             self.m4: T5Node = m4
129             self.m5: T5Node = m5
130             self.h1: Optional[int] = None
131             self.h2: Optional[int] = None
132
133         def get_child_count(self) -> int:
134             return self.m1.get_child_count() + \
135                    self.m2.get_child_count() + \
136                    self.m3.get_child_count() + \
137                    self.m4.get_child_count() + \
138                    self.m5.get_child_count()
139
140         def calc_end_hash(self):
141             h1 = self.calc_h1()
142             h2 = self.calc_h2()
143             m5_endhash = self.m5.calc_end_hash()
144             h11 = xor(h1, m5_endhash)
145             h21 = xor(h2, m5_endhash)
146
147             h3 = hash_t5(h11, h21)
148             return xor(h3, m5_endhash)
149
150         def calc_h1(self):
151             if self.h1 is None:
152                 self.h1 = hash_t5(self.m1.calc_end_hash(),
                    ↪  self.m2.calc_end_hash())
153             return self.h1
154
155         def calc_h2(self):
156             if self.h2 is None:
157                 self.h2 = hash_t5(self.m3.calc_end_hash(),
                    ↪  self.m4.calc_end_hash())
158             return self.h2
159
160         def get_hash_count(self):  # notably: XOR count == hash count
161             return self.m1.get_hash_count() + \
162                    self.m2.get_hash_count() + \
163                    self.m3.get_hash_count() + \
164                    self.m4.get_hash_count() + \
165                    self.m5.get_hash_count() + 3
166
167         def calc_auth_path(self, path: List[int]) -> List[int]:
168             child_pos = path[0]
169             remaining_path = path[1:]
170
171             auth_path = []
172             if child_pos == 0:  # if "key" = m0
```

```
173             auth_path.append(self.m2.calc_end_hash())  # value of m2
                ↪ "before" current node
174             auth_path.append(self.calc_h2())
175             auth_path.append(self.m5.calc_end_hash())
176             auth_path.extend(self.m1.calc_auth_path(remaining_path))  #
                ↪ path[1:] -> give rest of path to subtrees of m0
177         elif child_pos == 1:
178             auth_path.append(self.m1.calc_end_hash())  # value of m1
                ↪ "before" current node
179             auth_path.append(self.calc_h2())
180             auth_path.append(self.m5.calc_end_hash())
181             auth_path.extend(self.m2.calc_auth_path(remaining_path))  #
                ↪ path[1:] -> give rest of path to subtrees of m0
182
183         elif child_pos == 2:
184             auth_path.append(self.m4.calc_end_hash())
185             auth_path.append(self.calc_h1())
186             auth_path.append(self.m5.calc_end_hash())
187             auth_path.extend(self.m3.calc_auth_path(remaining_path))  #
                ↪ path[1:] -> give rest of path to subtrees of m0
188
189         elif child_pos == 3:
190             auth_path.append(self.m3.calc_end_hash())
191             auth_path.append(self.calc_h1())
192             auth_path.append(self.m5.calc_end_hash())
193             auth_path.extend(self.m5.calc_auth_path(remaining_path))  #
                ↪ path[1:] -> give rest of path to subtrees of m0
194
195         elif child_pos == 4:  # special case m5
196             # other possibility: add padding with 0 here to get same authpath
                ↪ len for every case
197             auth_path.append(self.calc_h1())
198             auth_path.append(self.calc_h2())
199             auth_path.extend(self.m5.calc_auth_path(remaining_path))  #
                ↪ path[1:] -> give rest of path to subtrees of m0
200
201         return auth_path
202
203
204 class T5Leaf(T5Node):
205     def __init__(self, leaf: int):
206         super().__init__()
207         self.leaf: int = leaf
208
209     def calc_end_hash(self):
210         return self.leaf
211
212     def get_hash_count(self) -> int:  # leaf does not have previous hash
        ↪ calls
213         return 0
214
```

```
215     def calc_auth_path(self, path: List[int]) -> List[int]:
216         return []
217 
218     def get_child_count(self) -> int:
219         return 1
220 
221 
222 if __name__ == '__main__':
223     s = 22  # leaf position of one-time key used by the signer
224     one_time_key = 22  # value of one-time key (here: has same value as
        ↪ position it's on, for debugging purposes)
225 
226     path = calc_t5_path(s, 2)
227     print('t5_path "T5 layers":', path)
228 
229     # Amount T5Leafs -> has to be power of 5, values of leaves == value of
        ↪ one-time public key
230     t5tree = T5Block(
231         T5Block(T5Leaf(0), T5Leaf(1), T5Leaf(2), T5Leaf(3), T5Leaf(4)),
232         T5Block(T5Leaf(5), T5Leaf(6), T5Leaf(7), T5Leaf(8), T5Leaf(9)),
233        T5Block(T5Leaf(10), T5Leaf(11), T5Leaf(12), T5Leaf(13), T5Leaf(14)),
234        T5Block(T5Leaf(15), T5Leaf(16), T5Leaf(17), T5Leaf(18), T5Leaf(19)),
235         T5Block(T5Leaf(20), T5Leaf(21), T5Leaf(22), T5Leaf(23), T5Leaf(24))
236     )
237 
238     child_count = t5tree.get_child_count()
239     print('Amount of leaves', child_count)
240 
241     print('Hash of root == public key:', t5tree.calc_end_hash())  # public
        ↪ key Y from signer
242 
243     print('# hash calls keygen / T5 tree generation:',
        ↪ t5tree.get_hash_count())
244 
245     # signer uses already constructed path -> no new hashcalls necessary
246     auth_path = t5tree.calc_auth_path(path)
247     print('Auth. path (calculated by signer):', auth_path)
248 
249     auth_path_by_verifier, hash_count_authentication =
        ↪ calc_path_verifier(auth_path, s, child_count, one_time_key)
250     print('Root calculated by verifier using Authentication path:',
        ↪ auth_path_by_verifier)
251     print('# hash calls verification (Path calculation by verifier):',
        ↪ hash_count_authentication)
```

B. Python Implementation: Performance Evaluation

The performance of the tree concepts Merkle Tree (see Section 2.4), T_5-Tree (see Section 4.1.3) and T_5-Tree$^+$ (see Section 4.2) for Chapter 5 is calculated with the following python script `performance_evaluation.py`:

```
# performance calculation of Binary Merkle Tree, T5 Tree, T5 Tree+
from typing import List
import math
import matplotlib.pyplot as plt

plt.rcParams.update({'font.size': 20})


# tree generation hash calls: standard Merkle Tree
def merkle_tree_gen_hash_calls(merkle_leaves_list: List[int]):
    merkle_tree_gen_hash_call_list = []
    for leaves in merkle_leaves_list:
        merkle_tree_gen_hash_call_list.append(leaves - 1)
    return merkle_tree_gen_hash_call_list


# auth.path length + hash calls verify: standard Merkle Tree
# is same calculation for: auth.path length, hash calls verify
def merkle_tree_len_auth_path_and_verify(merkle_leaves_list: List[int]):
    merkle_tree_len_auth_path_and_verify_list = []
    for leaves in merkle_leaves_list:
        merkle_tree_len_auth_path_and_verify_list.append(math.log2(leaves))
        ↪  # == 1.443 * log(leaves)
    return merkle_tree_len_auth_path_and_verify_list  # ! returns float


# tree generation hash calls: T5 Merkle Tree
def t5_tree_gen_hash_calls(t5_leaves_list: List[int]):
    t5_tree_gen_hash_calls_list = []
    for leaves in t5_leaves_list:
        t5_tree_gen_hash_calls_list.append(3 / 4 * (leaves - 1))
```

```
    return t5_tree_gen_hash_calls_list  # ! returns float


# auth.path length: t5 tree with aggressive opening
def t5_tree_aggr_len_auth_path(t5_leaves_list: List[int]):
    t5_tree_aggr_len_auth_path_list = []
    for leaves in t5_leaves_list:
        t5_tree_aggr_len_auth_path_list.append(round(3 * math.log(leaves,
        ↪  5)))  # == 1.86 * log(leaves)
    return t5_tree_aggr_len_auth_path_list  # ! returns float


# auth.path length: t5 tree with more aggressive opening
def t5_tree_more_aggr_len_auth_path(t5_leaves_list: List[int]):
    t5_tree_more_aggr_len_auth_path_list = []
    for leaves in t5_leaves_list:
        t5_tree_more_aggr_len_auth_path_list.append(round(2.8 *
        ↪  math.log(leaves, 5)))  # == 1.74 * log(leaves)
    return t5_tree_more_aggr_len_auth_path_list


# verify t5 aggressive: hash calls for path calculation
def t5_tree_aggr_verify(t5_leaves_list: List[int]):
    t5_aggr_verify_hash_calls_list = []
    for leaves in t5_leaves_list:
        t5_aggr_verify_hash_calls_list.append(round(2 * math.log(leaves,
        ↪  5)))  # == 1.24 * log(leaves)
    return t5_aggr_verify_hash_calls_list


# verify t5 more aggressive: hash calls for path calculation
def t5_tree_more_aggr_verify(t5_leaves_list: List[int]):
    t5_more_aggr_verify_hash_calls_list = []
    for leaves in t5_leaves_list:
        t5_more_aggr_verify_hash_calls_list.append(round(1.8 *
        ↪  math.log(leaves, 5)))  # == 1.12 * log(leaves)
    return t5_more_aggr_verify_hash_calls_list


# Merkle tree: convert list of tree-heights in list of leaves
def power_of_two(exponents: List[int]):
    powered_list = []
    for element in exponents:
        powered_list.append(pow(2, element))
    return powered_list


# t5 tree: convert list of tree-heights in list of leaves
def power_of_five(exponents: List[int]):
    powered_list = []
    for element in exponents:
```

```
78          powered_list.append(pow(5, element))
79      return powered_list
80
81
82  # NIST parameter: tree generation evaluation results plotted
83  def plot_hash_calls_tree_gen(merkle_leaves: List[int], low_bound_leaves:
    ↪  List[int],
84                               up_bound_leaves: List[int],
85                               merkle_hash_calls: List[int],
                                  ↪  low_bound_hash_calls: List[int],
86                               up_bound_hash_calls: List[int]):
87      markers_list = ["X", "o", "D", "^", "s"]
88
89      # lower bound plot
90      x2 = low_bound_leaves
91      y2 = low_bound_hash_calls
92      for (x, y, marker) in zip(x2, y2, markers_list):
93          plt.plot(x, y, marker=marker,  markersize=9, linestyle='None',
             ↪  color='b',
94                   label=r"low. bound T$_5$-Tree$^+$, d = " +
                      ↪  str(round(math.log(y, 5))))
95
96      # merkle tree plot
97      x1 = merkle_leaves
98      y1 = merkle_hash_calls
99      for (x, y, marker) in zip(x1, y1, markers_list):
100         plt.plot(x, y, marker=marker, markersize=9, linestyle='None',
             ↪  color='g',
101                  label='Merkle Tree, d =' + str(round(math.log2(y))))
102
103     # upper bound plot
104     x3 = up_bound_leaves
105     y3 = up_bound_hash_calls
106     for (x, y, marker) in zip(x3, y3, markers_list):
107         plt.plot(x, y, marker=marker,  markersize=9, linestyle='None',
             ↪  color='c',
108                  label=r"upper bound T$_5$-Tree, d = " +
                      ↪  str(round(math.log(y, 5))))
109
110     # make axes logarithmic
111     plt.xscale('log', base=2)
112     plt.yscale('log', base=2)
113
114     # labeling the axes
115     plt.xlabel(r'leaves $\ell$')
116     plt.ylabel('# hash calls')
117
118     # add legend
119     # plt.legend(loc='upper left')
120     plt.subplots_adjust(right=0.7)
121     plt.legend(bbox_to_anchor=(1.02, 0.5), loc='center left')
```

```
122
123     plt.title(label="Performance Evaluation: Merkle Tree, upper/lower bound
        ↪  T$_5$-Tree")
124     plt.grid()
125     plt.show()
126
127
128 if __name__ == '__main__':
129     param_set_d = [5, 10, 15, 20, 25]  # height in LMS parameter set for
        ↪  standard Merkle tree
130     lower_bound_d = [2, 4, 6, 8, 10]  # height for lower bound t5 trees
131     upper_bound_d = [3, 5, 7, 9, 11]  # height for upper bound t5 trees
132
133     # lists of possible leaves for each tree construct,
134     # based on LMS param. set
135     leaves_list_merkle_standard = power_of_two(param_set_d)
136     leaves_list_low_bound = power_of_five(lower_bound_d)
137     leaves_list_up_bound = power_of_five(upper_bound_d)
138
139     # ---- hash calls tree generation ----
140     # hash calls tree generation: merkle tree
141     hash_calls_tree_gen_merkle_tree =
        ↪  merkle_tree_gen_hash_calls(leaves_list_merkle_standard)
142     # hash calls tree generation: t5 upper/lower bound tree
143     hash_calls_tree_gen_low_bound =
        ↪  t5_tree_gen_hash_calls(leaves_list_low_bound)
144     hash_calls_tree_gen_up_bound =
        ↪  t5_tree_gen_hash_calls(leaves_list_up_bound)
145
146     # get int values for hash calls in t5 -> only when leaves are power of 5!
147     hash_calls_tree_gen_low_bound = [int(leaves) for leaves in
        ↪  hash_calls_tree_gen_low_bound]
148     hash_calls_tree_gen_up_bound = [int(leaves) for leaves in
        ↪  hash_calls_tree_gen_up_bound]
149
150     print('merkle tree: hash calls tree gen.',
        ↪  hash_calls_tree_gen_merkle_tree)
151     print('lower bound t5: hash calls tree gen.',
        ↪  hash_calls_tree_gen_low_bound)
152     print('upper bound t5: hash calls tree gen.',
        ↪  hash_calls_tree_gen_up_bound)
153
154     # ---- auth.path length ----
155     # amount elements in auth.path: merkle tree
156     len_auth_path_list_merkle_tree =
        ↪  merkle_tree_len_auth_path_and_verify(leaves_list_merkle_standard)
157
158     # amount elements in auth.path: lower bound / (more) aggressive
159     len_auth_path_list_low_aggr =
        ↪  t5_tree_aggr_len_auth_path(leaves_list_low_bound)
```

```
160     len_auth_path_list_low_more_aggr =
        ↪ t5_tree_more_aggr_len_auth_path(leaves_list_low_bound)
161
162     # amount elements in auth.path: upper bound / (more) aggressive
163     len_auth_path_list_up_more_aggr =
        ↪ t5_tree_aggr_len_auth_path(leaves_list_up_bound)
164     len_auth_path_list_up_aggr =
        ↪ t5_tree_more_aggr_len_auth_path(leaves_list_up_bound)
165
166     print('merkle tree: length auth.path:', len_auth_path_list_merkle_tree)
167
168     print('lower bound t5: aggr. length auth.path:',
        ↪ len_auth_path_list_low_aggr)
169     print('lower bound t5: more aggr. length auth.path:',
        ↪ len_auth_path_list_low_more_aggr)
170
171     print('upper bound t5: aggr. length auth.path:',
        ↪ len_auth_path_list_up_more_aggr)
172     print('upper bound t5: more aggr. length auth.path:',
        ↪ len_auth_path_list_up_aggr)
173
174     # ---- hash calls verify ----
175     # Merkle tree: hash calls for path generation / verify
176     # for Merkle tree: is same calculation as for length of auth.path
177     hash_calls_verify_merkle_tree = len_auth_path_list_merkle_tree
178
179     hash_calls_verify_aggr_low_bound =
        ↪ t5_tree_aggr_verify(leaves_list_low_bound)
180     hash_calls_verify_more_aggr_low_bound =
        ↪ t5_tree_more_aggr_verify(leaves_list_low_bound)
181
182     hash_calls_verify_aggr_up_bound =
        ↪ t5_tree_aggr_verify(leaves_list_up_bound)
183     hash_calls_verify_more_aggr_up_bound =
        ↪ t5_tree_more_aggr_verify(leaves_list_up_bound)
184
185     print('Merkle tree: hash calls verify', hash_calls_verify_merkle_tree)
186
187     print('lower bound t5: aggr verify', hash_calls_verify_aggr_low_bound)
188     print('lower bound t5: more aggr verify',
        ↪ hash_calls_verify_more_aggr_low_bound)
189
190     print('upper bound t5: aggr verify', hash_calls_verify_aggr_up_bound)
191     print('upper bound t5: more aggr verify',
        ↪ hash_calls_verify_more_aggr_up_bound)
192
193     plot_hash_calls_tree_gen(leaves_list_merkle_standard,
        ↪ leaves_list_low_bound,
194                              leaves_list_up_bound,
                                 ↪ hash_calls_tree_gen_merkle_tree,
```

```
195                                 hash_calls_tree_gen_low_bound,
                                    ↪  hash_calls_tree_gen_up_bound)
```

Bibliography

[1] V. Mavroeidis, K. Vishi, M. D., and A. Jøsang, "The impact of quantum computing on present cryptography," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 3, 2018.

[2] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[3] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack," *Phys. Rev. Lett.*, vol. 79, pp. 325–328, Jul 1997.

[4] R. L. Rivest, A. Shamir, and L. M. Adleman, United States Patent Patent US4 405 829A, 1983. [Online]. Available: https://patents.google.com/patent/US4405829

[5] K. K. Soni and A. Rasool, "Cryptographic attack possibilities over rsa algorithm through classical and quantum computation," in *2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 2018, pp. 11–15.

[6] Y. Wang, H. Zhang, and H. Wang, "Quantum polynomial-time fixed-point attack for RSA," *China Communications*, vol. 15, no. 2, pp. 25–32, 2018.

[7] M. S. Johannes Buchmann, Erik Dahmen, "Hash-based digital signature schemes," in *Post-Quantum Cryptography*, E. D. Daniel J. Bernstein, Johannes Buchmann, Ed. Springer-Verlag Berlin Heidelberg, 2009, pp. 35 – 93.

[8] C. Peng, J. Chen, S. Zeadally, and D. He, "Isogeny-based cryptography: A promising post-quantum technique," *IT Professional*, vol. 21, no. 6, pp. 27–32, 2019.

[9] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.

[10] M. D. Noel, V. O. Waziri, S. M. Abdulhamid, and J. A. Ojeniyi, "Review and analysis of classical algorithms and hash-based post-quantum algorithm," *Journal of Reliable Intelligent Environments*, pp. 1–18, 2021.

[11] C. NIST, "The digital signature standard," *Commun. ACM*, vol. 35, no. 7, p. 36–40, Jul. 1992. [Online]. Available: https://doi.org/10.1145/129902.129904

[12] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1997.

[13] K. Matusiewicz, *Analysis of Modern Dedicated Cryptographic Hash Functions*. Macquarie University, 2007.

[14] D. R. Stinson, "Some observations on the theory of cryptographic hash functions," *Designs, Codes and Cryptography*, vol. 38, no. 2, pp. 259–277, Feb 2006.

[15] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *Fast Software Encryption*, B. Roy and W. Meier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388.

[16] L. Lamport, "Constructing digital signatures from a one way function," Tech. Rep. CSL-98, October 1979, this paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010.

[17] D. McGrew, M. Curcio, and S. Fluhrer, "Leighton-micali hash-based signatures," Internet Requests for Comments, RFC Editor, RFC 8554, April 2019.

[18] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology — CRYPTO' 89 Proceedings*, G. Brassard, Ed. New York, NY: Springer New York, 1990, pp. 218–238.

[19] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: extended merkle signature scheme," Internet Requests for Comments, RFC Editor, RFC 8391, May 2018.

[20] C. Matt and U. Maurer, "The one-time pad revisited," in *2013 IEEE International Symposium on Information Theory*. IEEE, 2013, pp. 2706–2710.

[21] N. Fleischhacker, "Minimal assumptions in cryptography," Ph.D. dissertation, 2016.

[22] Q. Liu and M. Zhandry, "On finding quantum multi-collisions," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 189–218.

[23] G. Brassard, P. Høyer, and A. Tapp, "Quantum cryptanalysis of hash and claw-free functions," *Lecture Notes in Computer Science*, p. 163–169, 1998.

[24] P. Rogaway, "Formalizing human ignorance: Collision-resistant hashing without the keys," in *Progress in Cryptology - VIETCRYPT 2006*, P. Q. Nguyen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 211–228.

[25] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The SPHINCS+ signature framework," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2129–2146.

[26] F. Campos, T. Kohlstadt, S. Reith, and M. Stöttinger, "LMS vs XMSS: Comparison of stateful hash-based signature schemes on ARM Cortex-M4," in *Progress in Cryptology - AFRICACRYPT 2020*, A. Nitaj and A. Youssef, Eds. Cham: Springer International Publishing, 2020, pp. 258–277.

[27] Q. Yuan, M. Tibouchi, and M. Abe, "Security notions for stateful signature schemes," *IET Information Security*, vol. 16, no. 1, pp. 1–17, 2022.

[28] D. A. Cooper, D. C. Apon, Q. H. Dang, M. S. Davidson, M. J. Dworkin, C. A. Miller *et al.*, "Recommendation for stateful hash-based signature schemes," *NIST Special Publication*, vol. 800, p. 208, 2020.

[29] A. Hülsing, L. Rausch, and J. A. Buchmann, "Optimal parameters for XMSS mt," in *CD-ARES Workshops*, 2013.

[30] S. F. Panos Kampanakis, "LMS vs XMSS: Comparion of two hash-based signature standards," Cryptology ePrint Archive, Report 2017/349, 2017, https://ia.cr/2017/349.

[31] A. K. de Oliveira, J. Lopez, and R. Cabral, "High performance of hash-based signature schemes," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 3, pp. 421–432, 2017.

[32] A. Hülsing, C. Busold, and J. Buchmann, "Forward secure signatures on smart cards," Cryptology ePrint Archive, Report 2018/924, 2018.

[33] W. Wang, B. Jungk, J. Wälde, S. Deng, N. Gupta, J. Szefer, and R. Niederhagen, "XMSS and embedded systems," in *Selected Areas in Cryptography – SAC 2019*, K. G. Paterson and D. Stebila, Eds. Cham: Springer International Publishing, 2020, pp. 523–550.

[34] J. W. Bos, A. Hülsing, J. Renes, and C. van Vredendaal, "Rapidly verifiable xmss signatures," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, pp. 137–168, 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8730

[35] L. P. Perin, G. Zambonin, D. M. B. Martins, R. F. Custódio, and J. E. Martina, "Tuning the winternitz hash-based digital signature scheme," *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 00 537–00 542, 2018.

[36] 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 *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 368–397.

[37] J.-P. Aumasson, D. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange, M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, and P. Schwabe, "SPHINCS+ - submission to the 2nd round of the nist post-quantum project," march 2019.

[38] A. Hülsing, J. Rijneveld, and P. Schwabe, "ARMed SPHINCS – computing a 41kb signature in 16kb of ram," Cryptology ePrint Archive, Report 2015/1042, 2015.

[39] Y. Dodis, D. Khovratovich, N. Mouha, and M. Nandi, "T5: Hashing five inputs with three compression calls," Cryptology ePrint Archive, Report 2021/373, 2021.

[40] M. Stam, "Beyond uniformity: Better security/efficiency tradeoffs for compression functions," in *Advances in Cryptology – CRYPTO 2008*, D. Wagner, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 397–412.

[41] J.-P. Aumasson, D. Bernstein, W. Beullens, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange, M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, and B. Westerban, "SPHINCS+ - submission to the 3rd round of the nist post-quantum project," NIST PQC submission, October 2020.