Post-quantum signatures for today

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des Grades
Doktor rerum naturalium (Dr. rer. nat.)

von

Dipl.-Math. Erik Dahmen

geboren in Frankfurt am Main.

Referenten:	Prof. Dr. Johannes Buchmann Prof. Dr. Jintai Ding
Tag der Einreichung:	25. November 2008
Tag der mündlichen Prüfung:	5. Februar 2009
Hochschulkennziffer:	D 17

Darmstadt 2009

Wissenschaftlicher Werdegang

April 2006 - heute

Wissenschaftlicher Mitarbeiter und Promotionsstudent am Lehrstuhl von Prof. Johannes Buchmann, Fachbereich Informatik, Fachgebiet Theoretische Informatik, Technische Universität Darmstadt

Oktober 2000 - April 2006

Studium der Mathematik mit Schwerpunkt Informatik an der Technischen Universität Darmstadt

Publikationsliste

[1] Bernstein, D. J., Buchmann, J., Dahmen, E. (Eds.): Post-Quantum Cryptography. Springer, 2009, ISBN: 978-3-540-88701-0.

[2] Buchmann, J., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. Post-Quantum Cryptography - PQCrypto 2008, LNCS 5299, pages 63–77, Springer, 2008.

[3] Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital Signatures out of Second-Preimage Resistant Hash Functions, 2nd International Workshop on Post-Quantum Cryptography - PQCrypto 2008, LNCS 5299, pages 109–123, Springer, 2008.

[4] Rohde, S., Eisenbarth, T., Dahmen, E., Buchmann, J., Paar, C.: Fast Hash-Based Signatures on Constrained Devices, Eighth Smart Card Research and Advanced Application Conference - CARDIS 2008, LNCS 5189, pages 104-117, Springer, 2008.

[5] Rohde, S., Eisenbarth, T., Dahmen, E., Buchmann, J., Paar, C.: Efficient Hash-Based Signatures on Embedded Devices, Technical Report, SECSI - Secure Component and System Identification, 2008.

[6] Vuillaume, C., Okeya, K., Dahmen, E., Buchmann, J.: Public Key Authentication with Memory Tokens. 9th International Workshop on Information Security Applications - WISA 2008. LNCS 5379, Springer, 2008, to appear.

[7] Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle signatures with virtually unlimited signature capacity. Applied Cryptography and Network Security - ACNS 2007, LNCS 4521, pages 31–45, Springer, 2007.

[8] Dahmen, E., Okeya, K., Schepers, D.: Affine Precomputation with Sole Inversion in Elliptic Curve Cryptography. 12th Australasian Conference on Information Security and Privacy - ACISP'07, LNCS 4586, pages 245–258, Springer, 2007.

[9] Dahmen, E., Okeya, K., Takagi, T.: A New Upper Bound for the Minimal Density of Joint Representations in Elliptic Curve Cryptosystems. IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, Special Section on Discrete Mathematics and Its Applications, Volume E90-A, No.5, 2007, pages 952-959.

[10] Buchmann, J., Coronado, C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – an improved Merkle signature scheme. Progress in Cryptology - INDOCRYPT 2006, LNCS 4329, pages 349–363, Springer, 2006.

[11] Buchmann, J., Dahmen, E., May, A., Vollmer, U.: Krypto 2020. KES - The Information Security Journal (in German), Nr 5, 2006.

[12] Dahmen, E., Okeya, K., Takagi, T.: An Advanced Method for Joint Scalar Multiplications on Memory Constraint Devices. 2nd European Workshop on Security and Privacy in Ad hoc and Sensor Networks - ESAS 2005, LNCS 3813, pages 189–204, Springer, 2005.

[13] Dahmen, E., Okeya, K., Takagi, T.: Efficient Left-to-Right Multi-Exponentiations. Technische Universität Darmstadt, Technical Report TI-2/05, 2005.

Acknowlegdement

Most of all I would like to thank Prof. Dr. Johannes Buchmann for introducing me to the intriguing topic of Merkle signatures and for supervising me during the last three years. Further, I would like to thank Prof. Dr. Tsuyoshi Takagi and Dr. Katsuyuki Okeya for showing me that doing research is worthwhile and Dr. Camille Vuillaume for many fruitful discussions about Merkle signatures. I would not have been able to write this thesis without the support of the members of the CDC and MiniCrypt groups: many thanks to all of you, especially to Dr. Marc Fischlin for always having "a couple of minutes". I also would like to thank Kai Endres and Andy Müller for pointing out typos and other syntactical errors. Finally, I would like to thank Linda Schmidt for telling me when to work more and when to work less and Gabriele and Hans-Otto Dahmen for making this possible in the first place.

Darmstadt, *Erik Dahmen*
November 2008

Zusammenfassung

Digitale Signaturen sind von wesentlicher Bedeutung für die Sicherheit von Computernetzwerken wie dem Internet. Digitale Signaturen werden zum Beispiel eingesetzt, um die Authentizität und Integrität von Updates für Betriebssysteme und andere Software-Anwendungen zu gewährleisten. Die Sicherheit der wenigen in der Praxis eingesetzen Signaturverfahren ist durch Quantencomputer bedroht. Alle derzeit verwendeten Signaturverfahren werden unsicher, sobald große Quantencomputer gebaut werden können. Die Erforschung von alternativen Signaturverfahren, welche Angriffen durch Quantencomputer standhalten und konkurrenzfähig zu den heute verwendeten Verfahren sind, ist daher von sehr großer Bedeutung.

Ein viel versprechender Kandidat für ein Signaturverfahren, dass sicher gegen Angriffe durch Quantencomputer ist, ist das im Jahre 1979 von Merkle erfundene Merkle-Signaturverfahren. Allerdings hatte das Merkle-Signaturverfahren, selbst in Kombination mit den Verbesserungen von Szydlo und Coronado Effizienzprobleme, die es davon abgehalten haben wirklich praktisch zu sein. Zunächst einmal sind die Signierzeiten sehr unbalanciert. Signieren dauert im Worst-Case wesentlich länger als im Average-Case. Weiter produziert das Merkle–Szydlo–Coronado-Signaturverfahren sehr große Signaturen. Ebenfalls ist unklar ob Merkles Signaturverfahren in ressourcenbeschränkten Geräten einsetzbar ist.

Diese Arbeit präsentiert das „generalized Merkle signature scheme“ (GMSS), ein Signaturverfahren, welches die oben genannten Probleme löst. GMSS hat eine signifikant bessere Worst-Case-Signierzeit als das Merkle–Szydlo–Coronado-Signaturverfahren. Die Worst-Case-Signierzeit von GMSS entspricht der Average-Case-Signierzeit des Merkle–Szydlo–Coronado-Signaturverfahrens. Weiter ist die Worst-Case-Signierzeit von GMSS sehr nah an seiner Average-Case-Signierzeit. Damit stellt GMSS balancierte Zeiten für die Signaturerzeugung zur Verfügung. GMSS nutzt die verbesserten Signierzeiten, um die Größe der Signaturen spürbar zu verringern und bewahrt dabei Zeiten, die konkurrenzfähig zu den heute verwendeten Signaturverfahren sind. Eine Implementierung auf einem Microcontroller zeigt, dass GMSS auch in ressourcenbeschränkten Geräten einsetzbar ist. Diese Arbeit beschreibt weiterhin eine neue Konstruktionsmethode für Merkle Signaturen. Die neue Konstruktion ist beweisbar sicher unter schwächeren Sicherheitsannahmen und liefert ein Signaturverfahren mit signifikant höherem Sicherheitsniveau im Vergleich zu der ursprünglichen Konstruktion.

Abstract

Digital signatures are essential for the security of computer networks such as the Internet. For example, digital signatures are widely used to ensure the authenticity and integrity of updates for operating systems and other software applications. The security of the few practically used signature schemes is threatened by quantum computers. When large quantum computers are built, all currently used signature schemes will become insecure. It is therefore of extreme importance to develop alternative signature schemes that remain secure in the presence of quantum computers and which are able to compete with currently used signature schemes.

A very promising candidate for a signature scheme that withstands quantum computer attacks is the Merkle signature scheme invented by Merkle in 1979. However, even combined with the improvements by Szydlo and Coronado, the Merkle signature scheme has certain efficiency drawbacks that keep it from being truly practical. First of all, it has highly unbalanced signature generation times. In the worst case, signature generation takes significantly longer than on average. Secondly, the Merkle–Szydlo–Coronado signature scheme produces very large signatures. It is also unclear if Merkle's signature scheme is suitable for application on resource constrained devices.

The generalized Merkle signature scheme (GMSS) presented in this thesis solves the problems mentioned above. It drastically reduces the worst case signature generation time of the Merkle–Szydlo–Coronado signature scheme. The worst case signature generation time of GMSS corresponds to the average case signature generation time of the Merkle–Szydlo–Coronado signature scheme. Further, the worst case signature generation time of GMSS is extremely close to its average case signature generation time and thus, GMSS provides balanced timings for the signature generation. GMSS exploits the improved signature generation times to provide a noticeable reduction of the signature sizes while maintaining timings that are highly competitive to currently used signature schemes. A proof-of-concept implementation shows that GMSS can also be used on resource constrained devices and excellently compares to currently used signature schemes. This thesis also introduces a new construction method for Merkle signatures. This construction is provably secure under weaker security assumptions and yields a signature scheme with a significantly higher security level compared to the original construction.

Contents

1 Introduction

Digital signatures are extremely important for the security of computer networks such as the Internet. For example, digital signatures are widely used to ensure the authenticity and integrity of updates for operating systems and other software applications. Currently used signature schemes like RSA [33] and ECDSA [24] base their security on number theoretic problems. The security of RSA is based on the difficulty of factoring large composite numbers. The security of ECDSA is based on the difficulty of computing discrete logarithms in the group of points on an elliptic curve. Currently, the factoring problem is intractable for composite numbers larger than 2^{1024} and the elliptic curve discrete logarithm problem is intractable for groups of order larger than 2^{160}. In 1994, Shor proposed a quantum algorithm that is able to solve the factoring and discrete logarithm problem in polynomial time [38]. As a result, RSA and ECDSA will be completely broken when large scale quantum computers are built. But quantum computers are not the only threat to the security of RSA and ECDSA. New and unexpected ideas can always lead to more efficient algorithms to solve the underlying problems. For example, in the past 30 years, there has been tremendous progress in solving the factorization problem. As a result, alternative signature schemes that withstand quantum computer attacks are urgently required as replacements for RSA and ECDSA to ensure the security of IT infrastructures in the future. Such signature schemes are called *post-quantum signature schemes*. Post-quantum signature schemes must not only be resistant against quantum computer attacks, but also as efficient as currently used signature schemes and suitable for all applications where digital signatures are required.

A very promising post-quantum signature scheme is the Merkle signature scheme (MSS) invented by Merkle in 1979 [30]. The Merkle signature scheme uses a hash based one-time signature scheme to sign documents and a complete binary hash tree, also called a Merkle tree, to reduce the validity of many one-time verification keys to the validity of a single public key, the root of the Merkle tree. Like any other digital signature scheme, the Merkle signature scheme uses a cryptographic hash function. However, additional number theoretic assumptions are not required by the Merkle signature scheme. Its security solely relies on the cryptographic properties of the hash function. In fact, the Merkle signature scheme is provably secure assuming certain cryptographic properties of the hash function. Coronado showed that the Merkle signature scheme is existentially unforgeable under adaptive chosen message attacks, if the underlying hash function is collision resistant and the used one-time signature scheme is existentially unforgeable under adaptive chosen message attacks [14]. The best known quantum algorithm to break cryptographic properties of hash functions is the Grover algorithm [21]. Contrary to Shor's algo-

rithm, which speeds up factoring and computing discrete logarithms exponentially, applying Grover's algorithm results only in a square-root speed-up of generic attacks used to break cryptographic properties of hash functions. As a result, the security of hash functions and therefore the security of the Merkle signature scheme is only marginally affected when large quantum computers are built. Another benefit of the Merkle signature scheme is, that each new cryptographic hash function yields a new signature scheme. If a hash function is found insecure – which happened in the past [45, 46] and is likely to happen in the future – the Merkle signature scheme is easily repaired by using a new and secure hash function.

At first, the Merkle signature scheme didn't receive much attention because it had severe efficiency drawbacks. This changed in recent years when many improvements for the Merkle signature scheme were proposed. Especially the contributions by Szydlo [23, 42] and Coronado [10, 13] helped to improve the performance of the Merkle signature scheme. However, even with these improvements the Merkle signature scheme still has certain drawbacks that keep it from being truly practical. First of all, the signature generation times are highly unbalanced. In the worst case, signature generation takes significantly longer than on average. Secondly, the signatures produced by the Merkle–Szydlo–Coronado signature scheme are very large. Finally, it is unclear if Merkle's signature scheme is suitable for application on resource constrained devices.

This thesis provides solutions to the problems mentioned above. Two enhancements that drastically reduce the worst case signature generation time of the Merkle–Szydlo–Coronado signature scheme are presented. The signature scheme that combines these enhancements is called *generalized Merkle signature scheme* (GMSS). The worst case signature generation time of GMSS corresponds to the average case signature generation time of the Merkle–Szydlo–Coronado signature scheme. Further, the worst case signature generation time of GMSS is extremely close to its average case signature generation time and thus GMSS provides balanced timings for the signature generation. The first enhancement is a new algorithm for the computation of *authentication paths* [12]. The new algorithm has a significantly better worst case runtime than the best known algorithm for computing authentication paths, that is the algorithm proposed by Szydlo [43]. The difference to Szydlo's algorithm is that the Merkle tree nodes that must be computed are divided into two categories: leaves and inner nodes. Focusing on the computation of leaves, instead of tree nodes in general, leads to reduced worst case cost for the authentication path generation and therefore the signature generation. The second enhancement is the method of *distributed signature generation* [11]. This method drastically reduces the worst case signature generation time of Coronado's tree chaining method [10]. The idea of the distributed signature generation is based on the observation that most parts of the tree chaining signatures change only infrequently. The computation of these parts is not done at once, but distributed evenly across several steps. The improved signature generation time obtained by applying both enhancements is exploited to provide a noticeable reduction of the signature sizes using the Winternitz time-memory trade-off (see Section 2.1.2), while maintaining timings that

are highly competitive to currently used signature schemes. The practicability of both enhancements is substantiated by two implementations. The first is a JCA conform Java implementation included in the FlexiProvider [20]. The second is an implementation on an 8-bit AVR microcontroller [34]. These implementations show that GMSS is highly competitive to RSA and ECDSA and that it is possible to integrate the Merkle signature scheme in resource constrained devices.

As mentioned above, one advantage of the Merkle signature scheme is that it is provably secure. Its existential unforgeability under adaptive chosen message attacks can be reduced to cryptographic properties of the hash function. Based on Coronado's work [13, 14], this thesis states a security reduction to the preimage resistance and collision resistance of the hash function. Then, the security level of the Merkle signature scheme for certain output lengths of the hash function is estimated. The downside is that the security level of the Merkle signature scheme corresponds to the security level of the collision resistance property which is rather low compared to the security level of other cryptographic properties of hash functions, such as preimage resistance and second preimage resistance. Also, when hash functions are cryptanalyzed the collision resistance property is usually targeted first [45, 46]. It is therefore desirable to be able to reduce the existential unforgeability of the Merkle signature scheme under adaptive chosen message attacks to cryptographic properties of hash functions that provide a higher security level. This thesis shows how this can be accomplished. A new construction method for Merkle trees is introduced that yields a signature scheme existentially unforgeable under adaptive chosen message attacks, if the used hash function is preimage resistant and second preimage resistant [17]. Since second preimage resistance has a higher security level than collision resistance, the new scheme also has a higher security level than the original Merkle signature scheme.

The remainder of this thesis is organized as follows: Chapter 2 reviews the Merkle signature scheme. Chapter 3 presents the new authentication path algorithm. Chapter 4 introduces the method of distributed signature generation. Chapter 5 states timings and sizes of two implementations. Chapter 6 deals with the provably security of the Merkle signature scheme and introduces the new construction method for Merkle trees. Chapter 7 states the author's conclusion and future research topics.

2 The Merkle signature scheme

This chapter describes the basics of the Merkle signature scheme (MSS) and defines the notation used throughout this thesis. Section 2.1 introduces hash-based one-time signature schemes (OTS). Section 2.2 reviews Merkle's tree authentication scheme, that converts any one-time signature scheme into a multi-time signature scheme. Section 2.3 describes a space efficient algorithm for the construction of Merkle trees. Sections 2.4 and 2.5 describe efficiency improvements for MSS.

2.1 One-time signature schemes

This section introduces two hash-based one-time signature schemes (OTS). The first is the Lamport–Diffie one-time signature scheme (LD-OTS) proposed in [25]. The second is the Winternitz one-time signature scheme (W-OTS). It is an improvement of the LD-OTS and provides a trade-off between the signature generation time and the signature size [16, 30]. One-time signature schemes are digital signature schemes whose signature key is allowed to be used only once. This is because parts of the signature key are revealed with the signature. If a signature key is used twice, an attacker can combine the parts of the signature key contained in those signatures to generate a valid signature for a third digest. This is illustrated in Examples 2.2 and 2.4.

2.1.1 The Lamport–Diffie one-time signature scheme

Let n be a positive integer, the security parameter of the Lamport–Diffie one-time signature scheme (LD-OTS). LD-OTS uses a one-way function

$$f : \{0,1\}^n \to \{0,1\}^n,$$

and a cryptographic hash function

$$g : \{0,1\}^* \to \{0,1\}^n.$$

LD-OTS key pair generation. The signature key X of LD-OTS consists of $2n$ bit strings of length n chosen uniformly at random,

$$X = \big(x_{n-1}[0], x_{n-1}[1], \ldots, x_1[0], x_1[1], x_0[0], x_0[1]\big) \in_R \{0,1\}^{(n,2n)}. \tag{2.1}$$

The LD-OTS verification key Y is

$$Y = \big(y_{n-1}[0], y_{n-1}[1], \ldots, y_1[0], y_1[1], y_0[0], y_0[1]\big) \in \{0,1\}^{(n,2n)}, \tag{2.2}$$

where

$$y_i[j] = f\big(x_i[j]\big), \quad 0 \le i \le n-1, j = 0, 1. \tag{2.3}$$

So LD-OTS key generation requires $2n$ evaluations of f. The signature and verification keys are $2n$ bit strings of length n.

LD-OTS signature generation. A document $M \in \{0,1\}^*$ is signed using LD-OTS with a signature key X as in Equation (2.1). Let $g(M) = d = (d_{n-1}, \ldots, d_0)$ be the message digest of M. Then the LD-OTS signature is

$$\sigma = \big(x_{n-1}[d_{n-1}], \ldots, x_1[d_1], x_0[d_0]\big) \in \{0,1\}^{(n,n)}. \tag{2.4}$$

This signature is a sequence of n bit strings, each of length n. They are chosen as a function of the message digest d. The ith bit string in this signature is $x_i[0]$ if the ith bit in d is 0 and vice versa. Signing requires no evaluations of f. The length of the signature is n^2 bits.

LD-OTS signature verification. To verify a signature $\sigma = (\sigma_{n-1}, \ldots, \sigma_0)$ of M as in (2.4), the verifier calculates the message digest $d = (d_{n-1}, \ldots, d_0)$. Then he checks whether

$$\big(f(\sigma_{n-1}), \ldots, f(\sigma_0)\big) = \big(y_{n-1}[d_{n-1}], \ldots, y_0[d_0]\big). \tag{2.5}$$

Signature verification requires n evaluations of f.

Example 2.1. *Let $n = 3$, $f : \{0,1\}^3 \to \{0,1\}^3, x \mapsto x+1 \bmod 8$, and let $d = (1,0,1)$ be the hash value of a message M. We choose the signature key*

$$X = \big(x_2[0], x_2[1], x_1[0], x_1[1], x_0[0], x_0[1]\big) = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,6)}$$

and compute the corresponding verification key

$$Y = \big(y_2[0], y_2[1], y_1[0], y_1[1], y_0[0], y_0[1]\big) = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \in \{0,1\}^{(3,6)}.$$

The signature of $d = (1,0,1)$ is

$$\sigma = (\sigma_2, \sigma_1, \sigma_0) = (x_2[1], x_1[0], x_0[1]) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,3)}$$

Example 2.2. *We give an example to illustrate why the signature keys of LD-OTS must be used only once. Let $n = 4$. Suppose the signer signs two messages with*

digests $d_1 = (1,0,1,1)$ *and* $d_2 = (1,1,1,0)$ *using the same signature key. The signatures of these digests are*

$$\sigma_1 = (x_3[1], x_2[0], x_1[1], x_0[1]) \text{ and } \sigma_2 = (x_3[1], x_2[1], x_1[1], x_0[0]),$$

respectively. Then an attacker knows $x_3[1], x_2[0], x_2[1], x_1[1], x_0[0], x_0[1]$ *from the signature key. He can use this information to generate valid signatures for messages with digests* $d_3 = (1,0,1,0)$ *and* $d_4 = (1,1,1,1)$*. This example can be generalized to arbitrary security parameters* n*. Also, the attacker is only able to generate valid signatures for certain digests. As long as the hash function used to compute the message digest is cryptographically secure, he cannot find appropriate messages.*

2.1.2 The Winternitz one-time signature scheme

The Winternitz OTS (W-OTS) explained in the following produces significantly shorter signatures than the LD-OTS. The idea is to use one string in the one-time signature key to simultaneously sign several bits in the message digest. Like LD-OTS, W-OTS uses a one-way function

$$f : \{0,1\}^n \to \{0,1\}^n$$

and a cryptographic hash function

$$g : \{0,1\}^* \to \{0,1\}^n.$$

W-OTS key pair generation. A Winternitz parameter $w \geq 2$ is selected which is the number of bits to be signed simultaneously. Then

$$t_1 = \left\lceil \frac{n}{w} \right\rceil, \quad t_2 = \left\lceil \frac{\lfloor \log_2 t_1 \rfloor + 1 + w}{w} \right\rceil, \quad t = t_1 + t_2. \tag{2.6}$$

are determined. The signature key X is

$$X = \big(x_{t-1}, \ldots, x_1, x_0\big) \in_R \{0,1\}^{(n,t)}. \tag{2.7}$$

where the bit strings x_i are chosen uniformly at random.

The verification key Y is computed by applying f to each bit string in the signature key $2^w - 1$ times. So we have

$$Y = \big(y_{t-1}, \ldots, y_1, y_0\big) \in \{0,1\}^{(n,t)}, \tag{2.8}$$

where

$$y_i = f^{2^w-1}\big(x_i\big), 0 \leq i \leq t-1. \tag{2.9}$$

Key generation requires $t(2^w - 1)$ evaluations of f and the lengths of the signature and verification key are $t \cdot n$ bits, respectively.

W-OTS signature generation. A message M with message digest $g(M) = d = (d_{n-1}, \ldots, d_0)$ is signed. First, a minimum number of zeros is prepended to d such that the length of d is divisible by w. The extended string d is split into t_1 bit strings $b_{t-1}, \ldots, b_{t-t_1}$ of length w. Then

$$d = b_{t-1} \parallel \ldots \parallel b_{t-t_1}, \tag{2.10}$$

where $\parallel$ denotes concatenation. Next, the bit strings b_i are identified with integers in $\{0, 1, \ldots, 2^w - 1\}$ and the checksum

$$c = \sum_{i=t-t_1}^{t-1} (2^w - b_i) \tag{2.11}$$

is calculated. Since $c \leq t_1 2^w$, the length of the binary representation of c is less than

$$\lfloor \log_2 t_1 2^w \rfloor + 1 = \lfloor \log_2 t_1 \rfloor + w + 1. \tag{2.12}$$

A minimum number of zeros is prepended to this binary representation such that the length of the extended string is divisible by w. The extended string is split into t_2 blocks $b_{t_2-1}, \ldots, b_0$ of length w. Then

$$c = b_{t_2-1} || \ldots || b_0.$$

Finally the signature of M is computed as

$$\sigma = \big(f^{b_{t-1}}(x_{t-1}), \ldots, f^{b_1}(x_1), f^{b_0}(x_0)\big). \tag{2.13}$$

In the worst case, signature generation requires $t(2^w - 1)$ evaluations of f. The W-OTS signature size is $t \cdot n$.

W-OTS signature verification. For the verification of an Winternitz signature $\sigma = (\sigma_{t-1}, \ldots, \sigma_0)$ the bit strings $b_{t-1}, \ldots, b_0$ are calculated as explained above. Then we check if

$$\big(f^{2^w-1-b_{t-1}}(\sigma_{n-1}), \ldots, f^{2^w-1-b_0}(\sigma_0)\big) = \big(y_{n-1}, \ldots, y_0\big). \tag{2.14}$$

If the signature is valid, then $\sigma_i = f^{b_i}(x_i)$ and therefore

$$f^{2^w-1-b_i}(\sigma_i) = f^{2^w-1}(x_i) = y_i \tag{2.15}$$

holds for $i = t-1, \ldots, 0$. In the worst case, signature verification requires $t(2^w - 1)$ evaluations of f.

Example 2.3. *Let $n = 3, w = 2$, $f : \{0,1\}^3 \to \{0,1\}^3, x \mapsto x + 1 \bmod 8$ and $d = (1, 0, 0)$. We get $t_1 = 2, t_2 = 2$, and $t = 4$. We choose the signature key as*

$$X = \big(x_3, x_2, x_1, x_0\big) = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,4)}$$

and compute the verification key by applying f three times to the bit strings in X:

$$Y = \big(y_3, y_2, y_1, y_0\big) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \in \{0,1\}^{(3,4)}.$$

Prepending one zero to d and splitting the extended string into blocks of length 2 yields $d = 01||00$. The checksum c is $c = (4-1) + (4-0) = 7$. Prepending one zero to the binary representation of c and splitting the extended string into blocks of length 2 yields $c = 01||11$. The signature is

$$\sigma = (\sigma_3, \sigma_2, \sigma_1, \sigma_0) = \big(f(x_3), x_2, f(x_1), f^3(x_0)\big) = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \in \{0,1\}^{(3,4)}.$$

The signature is verified by computing

$$\big(f^2(\sigma_3), f^3(\sigma_2), f^2(\sigma_1), \sigma_0\big) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \in \{0,1\}^{(3,4)}$$

and comparing it with the verification key Y.

Example 2.4. *We give an example to illustrate why the signature keys of the W-OTS must be used only once. Let $w = 2$. Suppose the signer signs two messages with digests $d_1 = (1,0,0)$ and $d_2 = (1,1,1)$ using the same signature key. The signatures of these digests are*

$$\sigma_1 = \big(f(x_3), x_2, f(x_1), f^3(x_0)\big) \text{ and } \sigma_2 = \big(f(x_3), f^3(x_2), f(x_1), x_0\big),$$

respectively. The attacker can use this information to compute the signature for messages with digest $d_3 = (1,1,0)$ given as $\sigma_3 = \big(f(x_3), f^2(x_2), f(x_1), f(x_0)\big)$. Again this example can be generalized to arbitrary security parameters n. Also, the attacker can only produce valid signatures for certain digests. As long as the hash function used to compute the message digest is cryptographically secure, he cannot find appropriate messages.

2.2 Merkle's tree authentication scheme

The one-time signature schemes introduced in the last section are inadequate for most practical situations since each key pair can only be used for one signature. In 1979 Ralph Merkle proposed a solution to this problem [30]. His idea is to use a complete binary hash tree to reduce the validity of an arbitrary but fixed number of one-time verification keys to the validity of one single public key, the root of the hash tree.

The Merkle signature scheme (MSS) works with any cryptographic hash function and any one-time signature scheme. For the explanation we let $g : \{0,1\}^* \to \{0,1\}^n$ be a cryptographic hash function. We also assume that a one-time signature scheme has been selected.

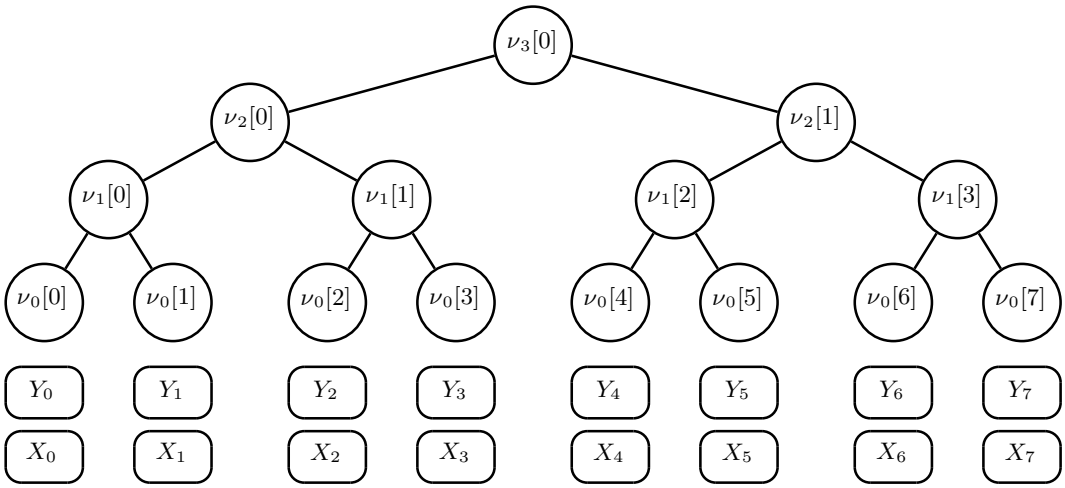

Figure 2.1: A Merkle tree of height $H = 3$

MSS key pair generation. The signer selects $H \in \mathbb{N}$, $H \geq 2$. Then the key pair to be generated will be able to sign/verify 2^H documents[1]. The signer generates 2^H one-time key pairs (X_j, Y_j), $0 \leq j < 2^H$. Here X_j is the signature key and Y_j is the verification key. They are both bit strings. The leaves of the Merkle tree are the digests $g(Y_j)$, $0 \leq j < 2^H$. The inner nodes of the Merkle tree are computed according to the following construction rule: a parent node is the hash value of the concatenation of its left and right children. The MSS public key is the root of the Merkle tree. The MSS private key is the sequence of the 2^H one-time signature keys. To be more precise, denote the nodes in the Merkle tree by $\nu_h[j]$, $0 \leq j < 2^{H-h}$, where $h \in \{0, \ldots, H\}$ is the height of the node. Then

$$\nu_h[j] = g(\nu_{h-1}[2j] \parallel \nu_{h-1}[2j+1]), \quad 1 \leq h \leq H, 0 \leq j < 2^{H-h}. \tag{2.16}$$

Figure 2.1 shows an example for $H = 3$. MSS key pair generation requires the computation of 2^H one-time key pairs and $2^{H+1} - 1$ evaluations of the hash function.

MSS signature generation. MSS uses the one-time signature keys successively for the signature generation. In order to sign a message M, the signer first computes the n-bit digest $d = g(M)$. Then he generates the one-time signature σ_{OTS} of the digest using the sth one-time signature key X_s, $s \in \{0, \ldots, 2^H - 1\}$. The Merkle signature will contain this one-time signature and the corresponding one-time verification key Y_s. To prove the authenticity of Y_s to the verifier, the signer also includes the index s and the authentication path for the verification key Y_s in the signature. The authentication path is a sequence $A_s = (a_0, \ldots, a_{H-1})$ of nodes of the Merkle tree.

[1]This is an important difference to signature schemes such as RSA and ECDSA, where potentially arbitrarily many documents can be signed/verified with one key pair. However, in practice this number is also limited by the devices on which the signature is generated or by policies.

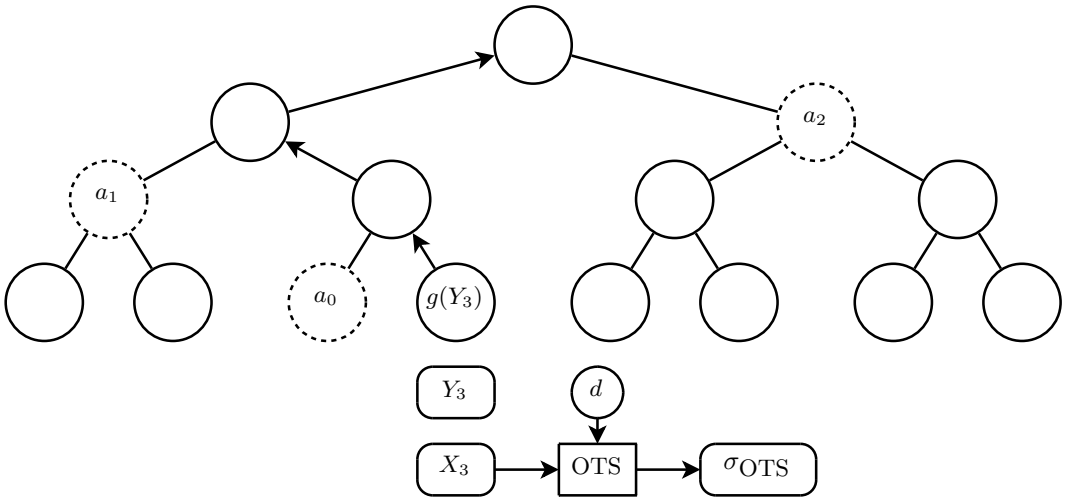

Figure 2.2: Merkle signature generation for $s = 3$. Dashed nodes denote the authentication path for leaf $g(Y_3)$. Arrows indicate the path from leaf $g(Y_3)$ to the root.

Node a_h in the authentication path is the sibling of the height h node on the path from leaf $g(Y_s)$ to the Merkle tree root:

$$a_h = \begin{cases} \nu_h[s/2^h - 1] & \text{, if } \lfloor s/2^h \rfloor \equiv 1 \bmod 2 \\ \nu_h[s/2^h + 1] & \text{, if } \lfloor s/2^h \rfloor \equiv 0 \bmod 2 \end{cases} \tag{2.17}$$

for $h = 0, \ldots H - 1$. The details of authentication path computation are discussed in Chapter 3. In total, the sth Merkle signature is

$$\sigma_s = \big(s, \sigma_{\text{OTS}}, Y_s, A_s\big). \tag{2.18}$$

Figure 2.2 shows an example for $s = 3$.

MSS signature verification. Verification of a Merkle signature as in (2.18) consists of two steps. In the first step, the verifier uses the one-time verification key Y_s to verify the one-time signature σ_{OTS} of the digest d by means of the verification algorithm of the respective one-time signature scheme. In the second step the verifier validates the authenticity of the one-time verification key Y_s by constructing the path $(p_0, \ldots, p_H)$ from the sth leaf $g(Y_s)$ to the root of the Merkle tree. He uses the index s and the authentication path $(a_0, \ldots, a_{H-1})$ and applies the following algorithm.

$$p_h = \begin{cases} g(a_{h-1}||p_{h-1}) & \text{, if } \lfloor s/2^{h-1} \rfloor \equiv 1 \bmod 2 \\ g(p_{h-1}||a_{h-1}) & \text{, if } \lfloor s/2^{h-1} \rfloor \equiv 0 \bmod 2 \end{cases} \tag{2.19}$$

for $h = 1, \ldots H$ and $p_0 = g(Y_s)$. The index s is used for deciding in which order the authentication nodes and the nodes on the path from leaf $g(Y_s)$ to the Merkle tree root are to be concatenated. The authentication of the one-time verification key Y_s is successful if and only if p_H equals the public key.

Performance. Table 2.1 states timings and sizes of a Java implementation of MSS. More details of the implementation can be found in Chapter 5. We use the W-OTS as one-time signature scheme and SHA1 as hash function. The first column denotes the height of the Merkle tree H and the Winternitz parameter w. Recall that 2^H signatures can be generated with one key pair. The remaining columns describe the signature size as well as timings for key pair generation, signature generation (average and worst case), and signature verification.

Table 2.1: Timings for MSS using SHA1

(H, w)	$m_{\text{signature}}$	c_{keygen}	$c_{\text{sign a.c.}}$	$c_{\text{sign w.c.}}$	c_{verify}
$(10, 4)$	1,064 bytes	0.8 sec	3.3 ms	3,9 ms	0.4 ms
$(15, 4)$	1,164 bytes	23.6 sec	4.9 ms	5,4 ms	0.4 ms
$(20, 4)$	1,264 bytes	12.6 min	7.0 ms	7,7 ms	0.4 ms
$(25, 4)$	1,364 bytes	6.7 h	7.7 ms	9,3 ms	0.4 ms

2.3 Efficient root computation

The last section showed that the whole Merkle tree must be computed in order to obtain its root. The number of operations required is exponential in the tree height H. However, it is possible to compute the root of a Merkle tree using space only linear in H. This can be done using the *treehash algorithm* described in Algorithm 2.1. The basic idea of this algorithm is to successively compute leaves and, whenever possible, compute their ancestors. In order to store nodes, the treehash algorithm uses a stack called STACK equipped with the usual push and pop operations. As input, the treehash algorithm takes the height H of the Merkle tree. Output is the root of the Merkle tree. Algorithm 2.1 uses the subroutine LEAFCALC(j) to compute the jth leaf. The LEAFCALC(j) routine first computes the jth one-time key pair (X_j, Y_j) and then the jth leaf as $g(Y_j)$.

Algorithm 2.1 Treehash

Input: Height $H \geq 2$

Output: Root of the Merkle tree

1. **for** $j = 0, \ldots, 2^H - 1$ **do**
   a) Compute the jth leaf: $\text{NODE}_1 \leftarrow \text{LEAFCALC}(j)$
   b) **While** NODE_1 has the same height as the top node on STACK **do**
      i. Pop the top node from the stack: $\text{NODE}_2 \leftarrow \text{STACK.pop}()$
      ii. Compute their parent node: $\text{NODE}_1 \leftarrow g(\text{NODE}_2 \parallel \text{NODE}_1)$
   c) Push the parent node on the stack: $\text{STACK.push}(\text{NODE}_1)$
2. Let R be the single node stored on the stack: $R \leftarrow \text{STACK.pop}()$
3. **Return** R

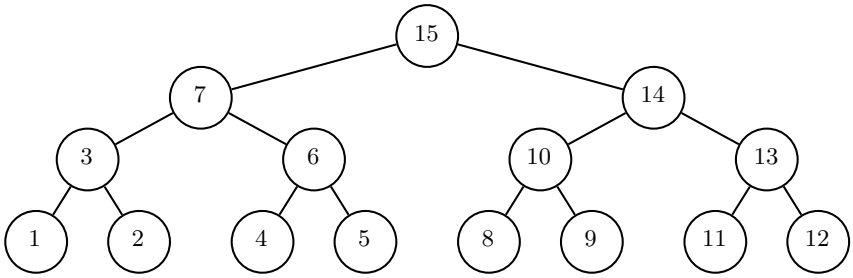

Figure 2.3: The treehash algorithm

Figure 2.3 shows the order in which the nodes of a Merkle tree are computed by the treehash algorithm. In this example, the maximum number of nodes that are stored on the stack is 3. This happens after node 11 is generated and pushed on the stack. In general, the treehash algorithm needs to store at most H so-called *tail nodes* on the stack. Computing the root of a Merkle tree of height H using the treehash algorithm requires 2^H calls of the LEAFCALC subroutine and $2^H - 1$ evaluations of the hash function.

2.4 Generating one-time signature keys

According to the description of MSS in Section 2.2, the MSS private key consists of 2^H one-time signature keys. Storing such a huge amount of data is not feasible for most practical applications. As suggested in [10], space can be saved by using a deterministic pseudo random number generator (PRNG) and storing only the seed of that PRNG. The length of this seed is usually the same as the output length of the hash function. Then each one-time signature key must be generated twice, once for the MSS public key generation and once during the signing phase.

In the following, let PRNG be a cryptographically secure pseudo random number generator that on input of a n-bit seed SEED_{in} outputs a random number RAND and an updated seed SEED_{out}, both of bit length n.

$$\begin{aligned} \text{PRNG} : \{0,1\}^n &\rightarrow \{0,1\}^n \times \{0,1\}^n \\ \text{SEED}_{\text{in}} &\mapsto (\text{RAND}, \text{SEED}_{\text{out}}) \end{aligned} \tag{2.20}$$

In addition to reducing the private key size, using a PRNG for the one-time signature key generation has another benefit. It makes MSS *forward secure* as long as PRNG is forward secure which means that calculating previous seeds from the current seed is infeasible. Forward security of the signature scheme means that all signatures issued before a revocation remain valid. MSS using a PRNG is forward secure, since the current seed can only be used to generate one-time signature keys for upcoming signatures. When using a PRNG for the one-time key pair generation, MSS key pair generation and MSS signature generation must be adjusted.

MSS + PRNG key pair generation. We explain how MSS key-pair generation using a PRNG works. The first step is to choose a n-bit seed SEED_0 randomly with the uniform distribution. For the generation of the one-time signature keys we use a sequence of seeds SEEDOTS_j, $0 \leq j < 2^H$. They are computed iteratively using SEED_j.

$$(\text{SEEDOTS}_j, \text{SEED}_{j+1}) \leftarrow \text{PRNG}(\text{SEED}_j), 0 \leq j < 2^H \tag{2.21}$$

SEEDOTS_j is used to calculate the jth one-time signature key. For example, in the case of W-OTS (see Section 2.1.2) the jth signature key is $X_j = (x_{t-1}, \ldots, x_0)$. The t bit strings of length n in this signature key are generated using SEEDOTS_j.

$$(x_i, \text{SEEDOTS}_j) \leftarrow \text{PRNG}(\text{SEEDOTS}_j), i = t-1, \ldots, 0 \tag{2.22}$$

When computing x_i, the seed SEEDOTS_j is updated. This shows that in order to calculate the signature key X_j only knowledge of SEED_j is necessary. When SEEDOTS_j is computed, the new seed SEED_{j+1} for the generation of the next signature key X_{j+1} is also determined. Figure 2.4 visualizes the one-time signature key generation using a PRNG. If this method is used, the MSS private key is initially SEED_0. Its length is n. It is replaced by the seeds SEED_{j+1} determined during the generation of signature key X_j, $0 \leq j < 2^H - 1$.

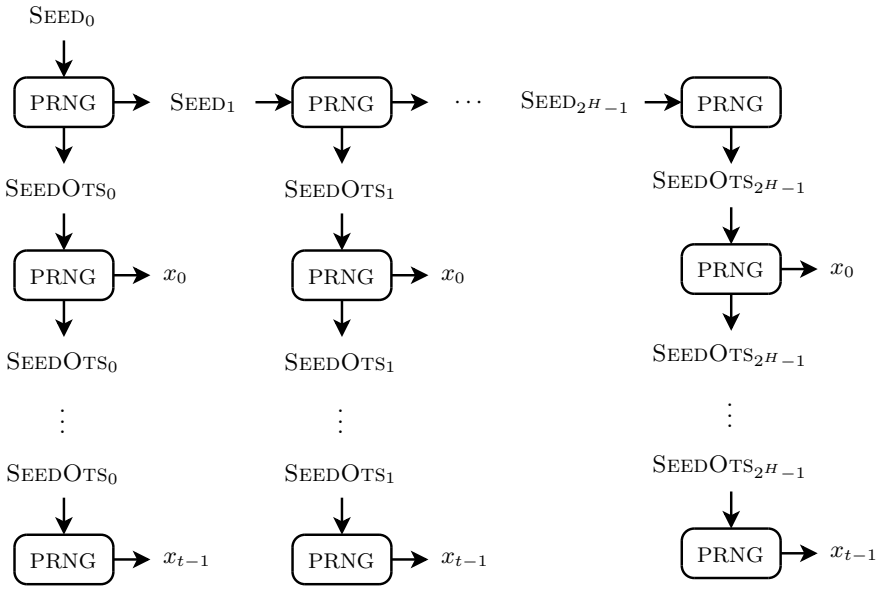

Figure 2.4: One-time signature key generation using a PRNG

MSS + PRNG signature generation. In contrast to the original MSS signature generation, the one-time signature key must be computed before the signature is generated. When the signature key is computed the seed is updated for the next signature.

2.5 Tree chaining

In Section 2.2 we saw that MSS public key generation requires the computation of the full Merkle hash tree. This means that 2^H leaves and $2^H - 1$ inner nodes have to be determined. This is very time consuming when H is large (see Table 2.1). The *tree chaining* method [11] explained in the following solves this problem.

The tree chaining method uses $T \geq 2$ *layers* of Merkle trees. Each Merkle tree on each layer is constructed using the methods described in Sections 2.2, 2.3, and 2.4. The hashes of a sequence of one-time verification keys are the leafs. We call the corresponding one-time signature keys the *signature keys of the Merkle tree*. Those signature keys are calculated using a pseudo random number generator. We call the respective seed the *seed of the Merkle tree*.

The root of the single tree on the top layer 1 is the public key. The signature keys of the Merkle trees on the bottom layer T are used to sign documents. The signature keys of the Merkle trees on the intermediate layers i, $1 \leq i < T$ sign the roots of the Merkle trees on layer $i + 1$.

This is what a tree chaining signature looks like:

$$\begin{aligned} \sigma \quad = \big(s, \quad & \textsc{Sig}_T, Y_T, \textsc{Auth}_T \\ & \textsc{Sig}_{T-1}, Y_{T-1}, \textsc{Auth}_{T-1} \\ & \qquad \vdots \\ & \textsc{Sig}_1, Y_1, \textsc{Auth}_1\big). \end{aligned} \tag{2.23}$$

$\textsc{Sig}_T$ is the one-time signature of the document to be signed. It is generated using a signature key of a Merkle tree on the bottom layer T. The corresponding verification key is Y_T. Also, $\textsc{Auth}_T$ is the authentication path that allows a verifier to construct the path from the verification key Y_T to the root of the corresponding Merkle tree on the bottom layer. That root is not known to the verifier. Therefore, the one-time signature $\textsc{Sig}_{T-1}$ of that root is also included in the signature σ. It is constructed using a signature key of a Merkle tree on level $T - 1$. The corresponding verification key Y_{T-1} and authentication path $\textsc{Auth}_{T-1}$ are also included in the signature σ. The root of the tree on layer $T - 1$ is also not known to the verifier, unless $T = 2$ in which case $T - 1 = 1$ and that root is the public key. So additional one-time signatures of roots $\textsc{Sig}_i$, one-time verification keys Y_i, and authentication paths $\textsc{Auth}_i$ are included in the signature σ for $i = T - 1, \ldots, 1$.

The signature σ is verified as follows. The verifier checks if $\textsc{Sig}_T$ can be verified using Y_T. Next, he uses Y_T and $\textsc{Auth}_T$ to construct the root of a Merkle tree on layer T. He verifies the signature $\textsc{Sig}_{T-1}$ of that root using the verification key Y_{T-1} and constructs the root of the corresponding Merkle tree on layer $T - 1$ from Y_{T-1} and $\textsc{Auth}_{T-1}$. The verifier iterates this procedure until the root of the single tree on layer 1 is constructed. The signature is verified by comparing this root to the public key. If any of those comparisons fail, the signature σ is rejected. Otherwise, it is accepted.

We now discuss the advantage of the tree chaining method. For this purpose, we first compute the number of signatures that can be verified using one public key

when the tree chaining method is applied. All Merkle trees on layer i have the same height H_i, $1 \leq i \leq T$. As mentioned already, there is a single Merkle tree on the top layer 1. Since the Merkle trees on layer i are used to sign the roots of the Merkle trees on layer $i+1$, $1 \leq i < T$, the number of Merkle trees on layer $i+1$ is $2^{H_1+H_2+\ldots+H_i}$. So the total number of documents that can be signed/verified is 2^H where $H = H_1 + H_2 + \ldots + H_T$.

The advantage of the tree chaining construction is the following. The generation of a public MSS key that can verify 2^H documents requires the construction of a tree of height H, which in turn requires the computation of 2^H one-time key pairs and $2^{H+1} - 1$ evaluations of the hash function. When tree chaining is used, the construction of a public key that can verify 2^H documents only requires the construction of the single Merkle tree on the top layer which is of height H_1. Also, in the tree chaining method, signature generation requires knowledge of the one-time signature of the root of one Merkle tree on each layer. Those roots and one-time signatures can be successively computed as they are used, whereas the root of the first tree on each layer is generated during the key generation. Hence, the key pair generation requires the computation of $2^{H_1} + \ldots + 2^{H_T}$ one-time key pairs and $2^{H_1+1} + \ldots + 2^{H_T+1} - T$ evaluations of the hash function. This is a drastic improvement compared to the original MSS key pair generation as illustrated in the following example.

Example 2.5. *Assume that the heights of all Merkle trees are equal, so* $H_1 = \ldots = H_T = H$. *The number of signatures that can be generated with this key pair is* 2^{TH}. *When using the tree chaining method, key pair generation requires* $T2^H$ *one-time key pairs and* $T2^{H+1} - T$ *evaluations of the hash function. The original MSS key pair generation requires* 2^{TH} *one-time key pairs and* $2^{TH+1} - 1$ *evaluations of the hash function.*

The combination of the original MSS and the tree chaining method is called CMSS. We now describe the key pair generation, signature generation, and signature verification of CMSS and show some timings.

CMSS key pair generation. For the CMSS key pair generation, the number of layers T and the respective heights H_i, $1 \leq i \leq T$ of the trees on layer i are selected. With $H = H_1+H_2+\ldots+H_T$ the number of signatures that can be generated/verified using the key pair to be constructed is 2^H. For each layer, one initial Merkle tree TREE_i is constructed as described in Sections 2.2, 2.3, and 2.4. The CMSS public key is the root of TREE_1. The CMSS secret key is the sequence of the random seeds used to construct the T trees. The signer also stores the one-time signatures of the roots of all those trees generated with the first signature key of the tree on the next layer.

CMSS key pair generation requires the computation of $2^{H_1} + \ldots + 2^{H_T}$ one-time key pairs and $2^{H_1+1} + \ldots + 2^{H_T+1} - T$ evaluations of the hash function.

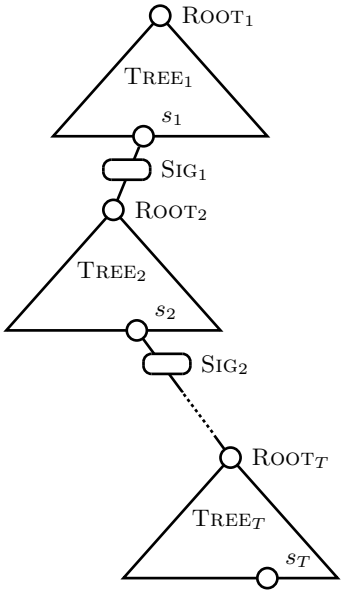

Figure 2.5: The tree chaining method. TREE_i denotes the active tree on layer i, ROOT_i its root, and SIG_{i-1} this root's one-time signature generated with the s_{i-1}th signature key of the tree on layer $i-1$.

CMSS signature generation. We use the notation of the previous sections. When a signature is issued, the signer knows one active Merkle tree TREE_i for each layer and the seed SEED_i from which its signature keys can be generated, $i = 1, 2, \ldots, T$. The signer also knows the signature SIG_i of the root of TREE_{i+1}, and the verification key Y_i for that signature, $1 \leq i \leq T-1$. Further, the signer knows the index s_i, $1 \leq i \leq T-1$, of the signature key used to generate the signature SIG_i of the root of the tree TREE_{i+1} and the index s_T of the signature key used to issue the next document signature. The signer constructs the corresponding signature key from the seed SEED_T, he generates the one-time signature SIG_T of the document to be signed and he generates the signature as in Equation (2.23). The index s in this signature can be recursively computed. Set $t_1 = s_1$ and

$$t_{i+1} = t_i 2^{H_{i+1}} + s_{i+1}, 1 \leq i < T,$$

then $s = t_T$. Figure 2.5 shows an example.

After signing, the signer prepares for the next signature by partially constructing the next tree on certain layers using the treehash algorithm of Section 2.3. He first computes the s_Tth leaf of the next tree on layer T and executes the treehash algorithm with this leaf as input. Then he increments s_T. If $s_T = 2^{H_T}$, then the construction of the next Merkle tree on layer T is completed and its root is available. The signer computes the one-time signature of this root using a signature key of the tree on layer $T-1$ and sets the index s_T to zero. In the same way, the

signer constructs the next tree on layer $T-1$ and increments the index s_{T-1}. More generally, the signer partially constructs the next tree on layer i, increments s_i, and generates the one-time signature SIG_i whenever the construction of the next tree on layer $i+1$ is complete, $1 < i < T$. On layer 1, no new tree is required and the signer only increments the index s_1 if the construction of a tree on layer 2 is completed. When $s_1 = 2^{H_1}$, CMSS cannot sign new documents anymore.

Since a CMSS signature consists of T MSS signatures, the signature size increases by a factor T compared to MSS. Also, the computation of the roots of the following trees and their signatures increases the signature generation time.

CMSS signature verification. The basics of the CMSS signature verification are straight forward and have already been explained above.

We now explain how the verifier uses s to determine a positive integer s_i for each layer i, such that Y_i is the s_ith verification key of the active tree on that layer. The verifier uses s_i to construct the path from Y_i to the root of the corresponding tree on layer i (see Section 2.2). The following formulae show how this can be accomplished.

$$\begin{aligned} j_T &= \lfloor s/2^{H_T} \rfloor, & j_i &= \lfloor j_{i+1}/2^{H_i} \rfloor, i = T-1, \ldots, 1 \\ s_T &= s \bmod 2^{H_T}, & s_i &= j_{i+1} \bmod 2^{H_i}, i = T-1, \ldots, 1 \end{aligned} \tag{2.24}$$

Performance. Table 2.2 states timings and sizes of a Java implementation of CMSS. More details of the implementation can be found in Chapter 5. We use two layers of trees ($T = 2$), the W-OTS as one-time signature scheme, and SHA1 as hash function. The first column denotes the height of the Merkle tree and the Winternitz parameter used on layer one (H_1, w_1) and layer two (H_2, w_2). In this case $2^{H_1+H_2}$ signatures can be generated with one key pair. The remaining columns describe the signature size as well as timings for key pair generation, signature generation (average and worst case), and signature verification.

Table 2.2: Timings for CMSS using SHA1

$((H_1, H_2), (w_1, w_2))$	$m_{\text{signature}}$	c_{keygen}	$c_{\text{sign a.c.}}$	$c_{\text{sign w.c.}}$	c_{verify}
$((10, 10), (4, 4))$	2,128 bytes	1.5 sec	4.7 ms	8.5 ms	0.7 ms
$((15, 15), (4, 4))$	2,328 bytes	46.7 sec	6.5 ms	11.6 ms	0.8 ms
$((20, 20), (4, 4))$	2,528 bytes	24.9 min	8.2 ms	16.2 ms	0.8 ms

This table clarifies the impact of the tree chaining method. For the same number of possible signatures, the key generation time is reduced drastically compared to MSS (see Table 2.1). As a result, it is now possible to efficiently generate key pairs that are able to sign more than 2^{20} documents.

When using CMSS with two layers, the signature size roughly doubles compared to MSS. This is because there are two one-time signatures included in the CMSS signature. Using CMSS results in unbalanced signature generation times; there is a large gap between the average case and worst case signature generation times.

3 Authentication path computation

In this chapter we describe an algorithm for the successive computation of authentication paths [12]. Authentication path computation is a major component of the MSS signature generation and accounts for a large part of the signature generation time. There are two different approaches to compute authentication paths. In [30] Merkle proposes to compute each authentication node separately. This idea is adopted by Szydlo in [42] where he implements a better scheduling of the node calculations and achieves the optimal trade-off, that is $O(H)$ time and $O(H)$ space. In [43], Szydlo further improves the constants of his algorithm. For each authentication path, the algorithm from [43] computes H nodes of the Merkle tree and requires storage for $3H - 2$ nodes. The second approach is called fractal Merkle tree traversal [23]. This approach splits the Merkle tree into smaller subtrees and stores a stacked series of subtrees that contain authentication paths for several succeeding leaves. Varying the height h of the subtrees allows a trade-off between time and space needed for the tree traversal. Using the low space solution ($h = \log H$) requires $O(H/\log H)$ time and $O(H^2/\log H)$ space. In [8], the authors improve the constants of this algorithm and prove the optimality of the fractal time-memory trade-off. Current algorithms for computing authentication paths have fairly unbalanced running times. The best case runtime of those algorithms is significantly shorter than the worst case runtime. So the computation of some authentication paths is very slow while other authentication paths can be computed very quickly.

The algorithm described in this chapter has a significantly better worst case runtime than the best algorithm known so far, which is Szydlo's algorithm from [43] providing the optimal time-memory trade-off. In fact, the worst case runtime of our algorithm is very close to its average case runtime which, in turn, equals the average case runtime of Szydlo's algorithm. The idea of our algorithm is to balance the number of leaves computed in each authentication path computation, since leaves are by far the most expensive nodes in the Merkle tree. All known approaches balance the number of nodes. This does not balance the running time since computing an inner node only requires one hash function evaluation, while computing a leaf takes several hundred hash function evaluations. This is because leaves are essentially one-time verification keys and thus the cost for computing a leaf is determined by the key pair generation cost of the respective one-time signature scheme. This problem is pointed out in [8, 31] but no solution has been provided so far. By balancing the number of leaves that are computed in each round and computing inner nodes as required, our algorithm achieves a worst case runtime which is extremely close to its average case runtime and thus provides balanced timings for the authentication path computation and MSS signature generation.

3.1 The algorithm

We now describe our authentication path algorithm in detail. As before, we denote the nodes in the Merkle tree by $\nu_h[j]$, where $h = H, \ldots, 0$ denotes the height of the node in the tree and $j = 0, \ldots, 2^{H-h} - 1$ its position on that height. Leaves have height 0 and the root has height H. We also use a cryptographic hash function $g : \{0,1\}^* \to \{0,1\}^n$.

Like Szydlo's algorithm from [43] we deploy two different strategies to compute authentication nodes, depending on whether the node is a left child (left authentication node, left node) or a right one. The difference to Szydlo's algorithm is, that we use a different approach for scheduling the computation of right nodes. Also, the computation of right nodes is split into two cases: right nodes close to the root are stored during the initialization and the remaining nodes are computed. We use the parameter K to determine up to which height right nodes are computed.

First, we describe the value τ that plays an essential role in our algorithm. Then we describe the used data structures and introduce the strategies to compute left and right authentication nodes. Finally we state a pseudo-code description of the algorithm.

The value τ. The algorithm determines $\tau = \max\{h : 2^h | (s+1)\}$ which is the height of the first ancestor of the sth leaf which is a left child. If leaf s is a left child itself, then $\tau = 0$. Figure 3.1 shows an example.

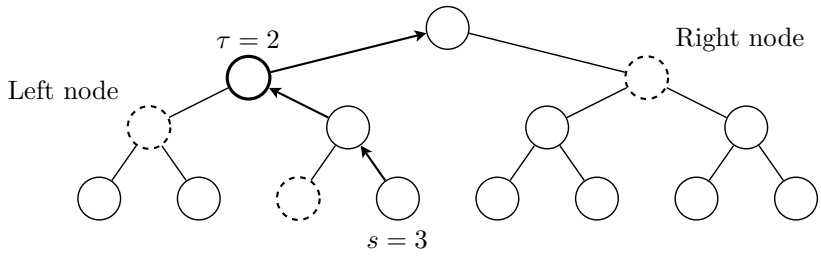

Figure 3.1: The height of the first ancestor of leaf s that is a left child is $\tau = 2$. The dashed nodes denote the authentication path for leaf s. The arrows indicate the path from leaf s to the root.

The value τ is used to determine at which heights the authentication path for leaf $s + 1$ requires new nodes. The authentication path for leaf $s + 1$ requires new right authentication nodes at heights $h = 0, \ldots, \tau - 1$ and one new left authentication node at height τ.

Data structures. Our algorithm uses the following data structures:

- $\text{AUTH}_h, h = 0, \ldots, H-1$. An array of nodes that stores the current authentication path.
- STACK. A stack of nodes with the usual push and pop operations.
- $\text{RETAIN}_h, h = H-K, \ldots, H-2$. Stack that stores all right nodes on height h.
- $\text{TREEHASH}_h, h = 0, \ldots, H-K-1$. These are instances of the treehash algorithm (see Algorithm 2.1 in Section 2.3). They are used to compute right authentication nodes on height h. All these treehash instances share the stack STACK. Further, each instance has the following entries and methods.
  - TREEHASH_h.node. This entry can store one tail node. Once the treehash algorithm is done, this entry contains the height h authentication node just computed.
  - TREEHASH_h.initialize(φ). This method initializes the treehash algorithm with the index φ of the leaf to begin with.
  - TREEHASH_h.update(). This method executes Algorithm 2.1 once, meaning that it computes the next leaf and performs the necessary hash function evaluations to compute this leaf's ancestors, if tail nodes are stored on the stack.
  - TREEHASH_h.height. This entry stores the height of the lowest tail node stored by this treehash instance, either on the stack STACK or in the entry TREEHASH_h.node. If TREEHASH_h does not store any tail nodes, then $\text{TREEHASH}_h.\text{height} = h$ holds. If TREEHASH_h is finished or not initialized, then $\text{TREEHASH}_h.\text{height} = \infty$ holds.
- $\text{KEEP}_h, h = 0, \ldots, H-2$. An array of nodes that stores certain nodes for the efficient computation of left authentication nodes.

Computing left nodes. We review the efficient computation of left authentication nodes due to [43]. As explained above, we require a left node on height τ for the next authentication path. Call this node AUTH_τ. If $\tau = 0$, then we set AUTH_0 to $\text{LEAFCALC}(s)$. Let $\tau > 0$. Then leaf s is a right child. The left child of AUTH_τ, is contained in the current authentication path as $\text{AUTH}_{\tau-1}$. We assume that the right child of AUTH_τ is stored in $\text{KEEP}_{\tau-1}$. Then the new node AUTH_τ is computed as

$$\text{AUTH}_\tau = g\big(\text{AUTH}_{\tau-1} \parallel \text{KEEP}_{\tau-1}\big). \tag{3.1}$$

This requires only one hash evaluation. We also explain how KEEP is updated. If $\lfloor s/2^{\tau+1} \rfloor = 0 \pmod 2$, i.e. if the ancestor on height $\tau+1$ is a left child, then AUTH_τ is a right node and we store it in KEEP_τ.

Computing right nodes. Unlike left authentication nodes, right authentication nodes must be computed from scratch, i.e. starting from the leaves. This is because none of their child nodes were used in previous authentication paths.

We use two different methods for computing right nodes. To distinguish those cases we select a positive integer $K \geq 2$ such that $H - K$ is even. Suppose that

we wish to compute a right node on height h. If $H - K \leq h \leq H - 2$, then the right node on height h is calculated by RETAIN_h.pop() which pops the top element from the stack RETAIN_h. That stack has been filled with the right nodes $\nu_h[3], \ldots, \nu_h[2^{H-h} - 1]$ during the initialization of our algorithm. This is very useful since the nodes close to the root are the most expensive ones to compute.

For the computation of a right node on height h with $h < H - K$ we use an instance TREEHASH_h of the treehash algorithm. All treehash instances share one stack. During initialization, the second right node $\nu_h[3]$ on height h is stored in TREEHASH_h.node. When a new right authentication node on height h is required, it is determined by TREEHASH_h.pop() for $h = 0, \ldots, \min\{H - K - 1, \tau - 1\}$, which yields the node stored in TREEHASH_h.node. Then all treehash instances for heights $h = 0, \ldots, \min\{H - K - 1, \tau - 1\}$ are initialized for the computation of the next right node. The index of the leaf they have to begin with is $s + 1 + 3 \cdot 2^h$ and the initialization is done using the method TREEHASH_h.initialize($s+1+3\cdot 2^h$). Then the algorithm updates the treehash instances using the TREEHASH_h.update() method. One update corresponds to one round of Algorithm 2.1, i.e. computing one leaf and computing this leaf's ancestors using tail nodes stored on the stack.

We allow a budget of $(H-K)/2$ updates in each round. We use the strategy from [42] to decide which of the $H-K$ treehash instances receives an update. For this, we need the value TREEHASH_h.height. The treehash instance that receives an update is the instance where TREEHASH_h.height contains the smallest value. If there is more than one such instance, we choose the one with the lowest index.

Pseudo-code. Algorithm 3.1 shows the pseudo-code of the initialization of our algorithm which is performed during MSS key pair generation. As input, it takes the parameters $H \geq 2$ and $K \geq 2$, where $H - K$ must be even. It outputs the algorithm's initial state. Algorithm 3.2 contains the precise description of the update and output phase of our algorithm. As input, it takes the index s of the current leaf, the parameters H, K, and the algorithm state AUTH, KEEP, RETAIN, TREEHASH prepared in previous rounds or the initialization. It outputs the authentication path for the next leaf $s + 1$ and the updated algorithm state.

Algorithm 3.1 Initialization

Input: $H \geq 2$, $K \geq 2$ such that $H - K$ is even.

Output: The initial state of the algorithm.

1. Store the authentication path for the first leaf ($s = 0$):
   $\text{AUTH}_h \leftarrow \nu_h[1], h = 0, \ldots, H - 1$
2. Store the next right authentication node in the treehash instances:
   TREEHASH_h.push($\nu_h[3]$), $h = 0, \ldots, H - K - 1$
3. Store the right authentication nodes close to the root:
   RETAIN_h.push($\nu_h[2j + 3]$), $h = H - K, \ldots, H - 2, j = 2^{H-h-1} - 2, \ldots, 0$.
4. Return the initial state:
   return AUTH, RETAIN, TREEHASH.

Algorithm 3.2 Update and output

Input: $s \in \{0, \ldots, 2^H - 2\}$, $H \geq 2$, K such that $H - K$ is even, and the algorithm state $\textsc{Auth}, \textsc{Keep}, \textsc{Retain}, \textsc{Treehash}$.

Output: Authentication path for leaf $s + 1$ and the updated algorithm state.

1. Let $\tau = 0$ if leaf s is a left node or let τ be the height of the first ancestor of leaf s which is a left node:
   $\tau \leftarrow \max\{h : 2^h | (s + 1)\}$
2. If the parent of leaf s on height $\tau + 1$ is a left node, store the current authentication node on height τ in $\textsc{Keep}_\tau$:
   if $\lfloor s/2^{\tau+1} \rfloor$ is even **and** $\tau < H - 1$ **then** $\textsc{Keep}_\tau \leftarrow \textsc{Auth}_\tau$
3. If leaf s is a left node, it is required for the authentication path of leaf $s + 1$:
   if $\tau = 0$ **then** $\textsc{Auth}_0 \leftarrow \textsc{Leafcalc}(s)$
4. Otherwise, if leaf s is a right node, the authentication path for leaf $s + 1$ changes on heights $0, \ldots, \tau$:
   if $\tau > 0$ **then**
   a) The authentication path for leaf $s + 1$ requires a new left node on height τ. It is computed using the current authentication node on height $\tau - 1$ and the node on height $\tau - 1$ previously stored in $\textsc{Keep}_{\tau-1}$. The node stored in $\textsc{Keep}_{\tau-1}$ can then be removed:
   $\textsc{Auth}_\tau \leftarrow g(\textsc{Auth}_{\tau-1} || \textsc{Keep}_{\tau-1})$, remove $\textsc{Keep}_{\tau-1}$
   b) The authentication path for leaf $s+1$ requires new right nodes on heights $h = 0, \ldots, \tau - 1$. For $h < H - K$ these nodes are stored in $\textsc{Treehash}_h$ and for $h \geq H - K$ in $\textsc{Retain}_h$:
   for $h = 0$ **to** $\tau - 1$ **do**
   if $h < H - K$ **then** $\textsc{Auth}_h \leftarrow \textsc{Treehash}_h.\text{pop}()$
   if $h \geq H - K$ **then** $\textsc{Auth}_h \leftarrow \textsc{Retain}_h.\text{pop}()$
   c) For heights $0, \ldots, \min\{\tau - 1, H - K - 1\}$ the treehash instances must be initialized anew. The treehash instance on height h is initialized with the index $\varphi = s + 1 + 3 \cdot 2^h$ of the leaf to begin with if $\varphi < 2^H$:
   for $h = 0$ **to** $\min\{\tau - 1, H - K - 1\}$ **do**
   if $s + 1 + 3 \cdot 2^h < 2^H$ **then** $\textsc{Treehash}_h.\text{initialize}(s + 1 + 3 \cdot 2^h)$
5. Next we spend the budget of $(H - K)/2$ updates on the treehash instances to prepare upcoming authentication nodes:
   repeat $(H - K)/2$ **times**
   a) We consider only stacks which are initialized and not finished. Let k be the index of the treehash instance whose lowest tail node has the lowest height. In case there is more than one such instance we choose the instance with the lowest index:
   $k \leftarrow \min \left\{ h : \textsc{Treehash}_h.\text{height}() = \min_{j=0,\ldots,H-K-1} \{\textsc{Treehash}_j.\text{height}()\} \right\}$
   b) The treehash instance with index k receives one update:
   $\textsc{Treehash}_k.\text{update}()$
6. Output the authentication path for leaf $s+1$ and the updated algorithm state:
   return $\textsc{Auth}_0, \ldots, \textsc{Auth}_{H-1}, \textsc{Keep}, \textsc{Retain}, \textsc{Treehash}$.

3.2 Correctness and analysis

In this section we show the correctness of Algorithm 3.2 and estimate its time and space requirements. First we show that the budget of $(H-K)/2$ updates per round is sufficient for the treehash instances to compute the nodes on time. Then we show that it is possible for all treehash instances to share a single stack. Next, we consider the time and space requirements of Algorithm 3.2. In detail we show that

i) On average, our algorithm computes $(H-K+1)/2$ leaves and $(H-K-1)/2+2^{K-H}$ hashes per round.
ii) The number of tail nodes stored on the stack is bounded by $H-K-2$.
iii) The number of hash function evaluations per round is bounded by $3(H-K-1)/2$.
iv) The number of nodes stored in KEEP is bounded by $\lfloor H/2 \rfloor + 1$.

To estimate the space complexity, we have to add the H nodes stored in AUTH, the $H-K$ nodes stored in TREEHASH and the $2^K - K - 1$ nodes stored in RETAIN. To estimate the time complexity, we have to add the $(H-K)/2$ leaf computations required to determine right nodes and one leaf and one hash to compute left nodes (Lines 3, 4a in Algorithm 3.2). Summing up the total time and space requirements results in the following theorem.

Theorem 3.1. *Let $H \geq 2$ and $K \geq 2$ such that $H-K$ is even. Algorithm 3.2 stores at most $3H + \lfloor H/2 \rfloor - 3K - 2 + 2^K$ nodes and each node requires n bits of memory. Algorithm 3.2 requires at most $(H-K)/2+1$ leaf computations and $3(H-K-1)/2+1$ hash function evaluations per round to successively compute authentication paths. On average, Algorithm 3.2 computes $(H-K+1)/2$ leaves and $(H-K-1)/2+2^{K-H}$ hashes.*

Nodes are computed on time. If TREEHASH_h is initialized in round s, the authentication node on height h computed by this instance is required in round $s+2^{h+1}$. In these 2^{h+1} rounds there are $(H-K)2^h$ updates available. TREEHASH_h requires 2^h updates. During the 2^{h+1} rounds, $2^{h+1}/2^{i+1}$ treehash instances are initialized on heights $i = 0, \ldots, h-1$, each requiring 2^i updates. In addition, active treehash instances on heights $i = h+1, \ldots, H-K-1$ might receive updates until their lowest tail node has height h, thus requiring at most 2^h updates.

Summing up the number of updates required by all treehash instances yields

$$\sum_{i=0}^{h-1} \frac{2^{h+1}}{2^{i+1}} \cdot 2^i + 2^h + \sum_{i=h+1}^{H-K-1} 2^h = (H-K)2^h \tag{3.2}$$

as an upper bound for the number of updates required to finish TREEHASH_h on time. For $h = H-K-1$ this bound is tight.

Sharing a single stack works. To show that it is possible for all treehash instances to share a single stack, we have to show that if TREEHASH_h receives an update and has tail nodes stored on the stack, all these tail nodes are on top of the stack.

When TREEHASH_h receives its first update, the height of the lowest tail node of TREEHASH_i, $i \in \{h+1, \ldots, H-K-1\}$ is at least h. This means that TREEHASH_h is completed before TREEHASH_i receives another update and thus tail nodes of higher treehash instances do not interfere with tail nodes of TREEHASH_h.

While TREEHASH_h is active and stores tail nodes on the stack, it is possible that treehash instances on lower heights $i \in \{0, \ldots, h-1\}$ receive updates and store nodes on the stack. If TREEHASH_i receives an update, the height of the lowest tail node of TREEHASH_h has height $\geq i$. This implies that TREEHASH_i is completed before TREEHASH_h receives another update and therefore doesn't store any tail nodes on the stack.

Average costs. We now estimate the average cost of our algorithm in terms of leaves (L) and hash function evaluations to compute inner nodes (I). We begin with the right nodes. On height $h = 0$ there are 2^{H-1} right leaves to compute. On heights $h = 1, \ldots, H-K-1$, there are 2^{H-h-1} right nodes to compute. The computation of each of these nodes requires 2^h leaves and $2^h - 1$ hash function evaluations. For the left nodes, we must compute one leaf and one inner node every second step, alternating. This makes a total of 2^{H-1} leaves and inner nodes. Hence, the total number of leaves and hash function evaluations that must be computed is

$$\left(\sum_{h=0}^{H-K-1} 2^{H-h-1} \cdot 2^h + 2^{H-1}\right) L + \left(\sum_{h=1}^{H-K-1} 2^{H-h-1} \cdot (2^h - 1) + 2^{H-1}\right) I \quad (3.3)$$

$$= \left(\frac{H-K+1}{2} \cdot 2^H\right) L + \left(\frac{H-K-1}{2} \cdot 2^H + 2^K\right) I. \quad (3.4)$$

To obtain the average cost per round we divide by 2^H.

Space required by the stack. We will show that the stack stores at most one tail node on each height $h = 0, \ldots, H-K-3$ at a time. TREEHASH_h, $h \in \{0, \ldots, H-K-1\}$ stores up to h tail nodes on different heights to compute the authentication node on height h. The tail node on height $h-1$ is stored by the treehash instance and the remaining tail nodes on heights $0, \ldots, h-2$ are stored on the stack. When TREEHASH_h receives its first update, the following two conditions hold: (1) all treehash instances on heights $< h$ are either empty or completed and store no tail nodes on the stack. (2) All treehash instances on heights $> h$ are either empty or completed or have tail nodes of height at least h. If a treehash instance on height $i \in \{h+1, \ldots, H-K-1\}$ stores a tail node on the stack, then all treehash instances on heights $i+1, \ldots, H-K-1$ have tail nodes of height at least i, otherwise the treehash instance on height i wouldn't have received any updates in the first place. This shows that there is at most one tail node on each height $h = 0, \ldots, H-K-3$ which bounds the number of nodes stored on the stack by $H-K-2$. This bound is tight for round $s = 2^{H-K+1} - 2$, before the update that completes the treehash instance on height $H-K-1$.

Number of hashes required per round. Assume that the maximum number of hash function evaluations is required in the following case: TREEHASH$_{H-K-1}$ receives all $u = (H-K)/2$ updates and is completed in this round. On input of an index s, the number of hashes required by the treehash algorithm is equal to the height of the first ancestor of leaf s which is a left node. On height h, a left node occurs every 2^h leaves, which means that every 2^h updates at least h hashes are required by treehash. During the u available updates, there are $\lceil u/2^h \rceil$ updates that require at least h hashes for $h = 1, \ldots, \lceil \log_2 u \rceil$. The last update requires $H-K-1 = 2u-1$ hashes to complete the treehash instance on height $H-K-1$. So far only $\lceil \log_2 u \rceil$ of these hashes were counted, so we have to add another $2u - 1 - \lceil \log_2 u \rceil$ hashes. In total, we get the following upper bound for the number of hashes required per round.

$$B = \sum_{h=1}^{\lceil \log_2 u \rceil} \left\lceil \frac{u}{2^h} \right\rceil + 2u - 1 - \lceil \log_2 u \rceil \tag{3.5}$$

In round $s = 2^{H-K+1} - 2$ this bound is tight. This is the last round before the treehash instance on height $H-K-1$ must be completed and as explained above, all available updates are required in this case. The desired upper bound is estimated as follows:

$$\begin{aligned} B &\leq \sum_{h=1}^{\lceil \log_2 u \rceil} \left(\frac{u}{2^h} + 1 \right) + 2u - 1 - \lceil \log_2 u \rceil \\ &= u \sum_{h=1}^{\lceil \log_2 u \rceil} \frac{1}{2^h} + 2u - 1 = u \left(1 - \frac{1}{2^{\lceil \log_2 u \rceil}} \right) + 2u - 1 \\ &\leq u \left(1 - \frac{1}{2u} \right) + 2u - 1 = 3u - \frac{3}{2} = \frac{3}{2}(H-K-1) \end{aligned}$$

The next step is to show that the above mentioned case is indeed the worst case. If a treehash instance on height $< H-K-1$ receives all updates and is completed in this round, less than B hashes are required. The same holds if the treehash instance receives all updates but is not completed in this round. The last case to consider is the one where the u available updates are spent on treehash instances on different heights. If the active treehash instance has a tail node on height j, it will receive updates until it has a tail node on height $j+1$, which requires 2^j updates and 2^j hashes. Additional $t \in \{1, \ldots, H-K-j-2\}$ hashes are required to compute the parent of this node on height $j+t+1$, if the active treehash instance stores tail nodes on heights $j+1, \ldots, j+t$ on the stack and in the treehash instance itself. The next treehash instance that receives updates has a tail node of height $\geq j$. Since the stack stores at most one tail node for each height, this instance can receive additional hashes only if there are enough updates to compute a tail node on height $\geq j+t$, the height of the next tail node possibly stored on the stack. But this is the same scenario that appears in the above mentioned worst case, i.e. if a node on

height $j+1$ is computed, the tail nodes on the stack are used to compute its parent on height $j+t+1$ and the same instance receives the next update.

Space required to compute left nodes. First we show that whenever an authentication node is stored in KEEP_h, $h = 1, \ldots, H-2$, the node stored in KEEP_{h-1} is removed in the same round. This immediately follows from Steps 2 and 4a in Algorithm 3.2. Second we show that if a node gets stored in KEEP_h, $h = 0, \ldots, H-3$, then KEEP_{h+1} is empty. To see this we have to consider in which rounds a node is stored in KEEP_{h+1}. This is true for rounds $s \in A_a = \{2^{h+1} - 1 + a \cdot 2^{h+3}, \ldots, 2^{h+2} - 1 + a \cdot 2^{h+3}\}, a \in \mathbb{N}_0$. In rounds $s' = 2^h - 1 + b \cdot 2^{h+2}, b \in \mathbb{N}_0$, a node gets stored in KEEP_h. It is straight forward to compute that $s' \in A_a$ implies that $2a + 1/4 \leq b \leq 2a + 3/4$ which is a contradiction to $b \in \mathbb{N}_0$.

As a result, at most $\lfloor H/2 \rfloor$ nodes are stored in KEEP at a time and two consecutive nodes can share one entry. One additional entry is required to temporarily store the authentication node on height h (Step 2) until node on height $h-1$ is removed (Step 4a).

3.3 Performance

We now compare the performance of Algorithm 3.2 and Szydlo's algorithm [43]. Table 3.1 compares the number of leaves and inner nodes computed per step in the worst case and on average. This table also shows the number of tree nodes that must be stored. We set the value K for our algorithm as small as possible, i.e. $K = 2$ is H is even and $K = 3$ otherwise. We state the comparison only for Merkle trees up to height $H = 20$, since for larger heights the MSS key pair generation becomes too inefficient and such trees cannot be used in practice (see Section 2.5).

Table 3.1: Leaves and inner nodes computed by Algorithm 3.2 and Szydlo's algorithm on average and in the worst case and the number of tree nodes that must be stored.

	average case		worst case		
H	leaves	inner nodes	leaves	inner nodes	space
Algorithm 3.2					
10	4.0	3.0	5	8	31
15	6.3	5.3	7	14	49
20	9.0	8.0	10	24	66
Szydlo's algorithm					
10	4.5	3.5	7	3	28
15	7.0	6.0	10	5	43
20	9.5	8.5	12	8	58

Table 3.1 shows that in the worst case, our algorithm computes less leaves and more inner nodes than Szydlo's algorithm. The average performance of our algorithm is better, which is a result of the slightly increased memory requirements.

The practical advantage of our algorithm, that is the impact of computing less leaves, is clarified by Table 3.2. This table states the actual number of hash function evaluations and calls to PRNG required per step, again for the the worst case and average case. We use SHA1 as hash function and the Winternitz one-time signature scheme with $w = 4$. Then, according to Equation (2.6) and Section 2.4, each leaf calculation requires 646 hash function evaluations and 44 calls to PRNG. The computation of an inner node requires one hash function evaluation. Table 3.2 also shows timings for both algorithms. To estimate the timings, we measured the time required for one hash function evaluation and one call to PRNG on the test platform[1] and multiplied these times with the total number of hash function evaluations and calls to PRNG required. On the test platform, one SHA1 evaluation takes 1.1 microseconds and one call to PRNG 1.27 microseconds.

Table 3.2: Number of hash function evaluations and calls to PRNG required by Algorithm 3.2 and Szydlo's algorithm on average and in the worst case when using SHA1 and Winternitz parameter $w = 4$.

	average case			worst case		
H	hashes	PRNG	time	hashes	PRNG	time
Algorithm 3.2						
10	2588.3	176.1	3.1 ms	3238	220	3.8 ms
15	4042.8	275.0	4.8 ms	4536	308	5.4 ms
20	5822.0	396.0	6.9 ms	6484	440	7.7 ms
Szydlo's algorithm						
10	2911.1	198.0	3.5 ms	4525	308	5.4 ms
15	4528.0	308.0	5.4 ms	6465	440	7.7 ms
20	6145.5	418.0	7.3 ms	7760	528	9.2 ms

In the worst case, our algorithm is significantly faster than Szydlo's algorithm. When using SHA1 as hash function and the Winternitz parameter $w = 4$, our algorithm is 28.5%, 29.8%, 16.5% faster than Szydlo's algorithm for $H = 10, 15, 20$, respectively. On average, our algorithm is 3.1%, 4.8%, 6.9% faster than Szydlo's algorithm for $H = 10, 15, 20$, respectively. As mentioned above, this is a result of the slightly increased memory requirement. More importantly, comparing the average case and worst case runtime of our algorithm shows, that the worst case runtime of our algorithm is extremely close to its average case runtime. This certifies that our algorithm provides balanced timings for the authentication path generation and thus the MSS signature generation.

[1] AMD Athlon 64 X2 5200+ EE, 2.6 GHz, 2GB memory, MS Windows XP, Java 1.6.0_02.

We would like to point out that in order to achieve the space bounds for Szydlo's algorithm stated in Table 3.1, additional implementing effort and possibly overhead must be taken into account on platforms without dynamic memory allocation. This is because Szydlo's algorithm uses separate stacks for each of the H treehash instances. Roughly speaking, each stack can store up to $O(H)$ nodes but all stacks together never store more than $O(H)$ nodes at a time. Simply reserving the memory required by each stack yields memory usage quadratic in H.

Furthermore, comparing the timings of our algorithm with the signature generation times of MSS stated in Table 2.1 makes it clear that the majority of the signature generation time is spend on authentication path computation. Improving the authentication path computation is thus essential for improving the MSS signature generation times.

4 Distributed signature generation

This chapter describes the idea of *distributed signature generation* [11]. This method counteracts the problems that arise when using the tree chaining method described in Section 2.5, namely the unbalanced signature generation times and the increased signature size. It is based on the observation that the one-time signatures of the roots and the authentication paths in upper layers change only infrequently. The idea is to distribute the operations required for the generation of these one-time signatures and authentication paths evenly across each step. This significantly improves the worst case signature generation time. The signature scheme that uses distributed signature generation is called *generalized Merkle signature scheme* (GMSS). The worst case signature generation time of GMSS corresponds to the average case signature generation time of the tree chaining method (CMSS). Also, the worst case signature generation time of GMSS is extremely close to its average case signature generation time and thus GMSS provides balanced signature generation times. The improved signature generation time enables us to reduce the signature size using the Winternitz time-memory trade-off explained in Section 2.1 without sacrificing efficiency.

4.1 The idea

Fix a layer $2 \leq i \leq T$. Denote the active tree on layer i by $\textsc{Tree}_i$. It is currently used to sign roots or documents. The preceding tree on that layer is denoted by $\textsc{TreePrev}_i$. The next tree on layer i is $\textsc{TreeNext}_i$. The idea of the distributed signature generation is the following. When $\textsc{Tree}_i$ is used, the root of $\textsc{TreeNext}_i$ is known. The root of $\textsc{TreeNext}_i$ is signed while the signature keys of $\textsc{Tree}_i$ are used. The root of $\textsc{TreeNext}_i$ was calculated while $\textsc{TreePrev}_i$ was used to sign documents or roots. In the following, c_{hash} denotes the cost for one hash function evaluation and c_{prng} denotes the cost of one call to PRNG.

Distributed root signing. We use the notation from above. We explain how the root of $\textsc{TreeNext}_i$ is signed while using $\textsc{Tree}_i$. By construction, the necessary signature key from layer $i-1$ is known.

We distribute the computation of the signature of the root of $\textsc{TreeNext}_i$ across the leaves of $\textsc{Tree}_i$. When the first leaf of $\textsc{Tree}_i$ is used we initialize the Winternitz one-time signature generation by calculating the parameters and executing the padding. Then we calculate the number of hash function evaluations and calls to PRNG required to compute the one-time signature key and the one-time signature.

We divide those numbers by 2^{H_i} where H_i is the height of TREE_i to estimate the number of operations required per step. When a leaf of TREE_i is used, the appropriate amount of computation for the signature of the root of TREENEXT_i is performed. The distributed generation of the one-time signatures is visualized in Figure 4.1.

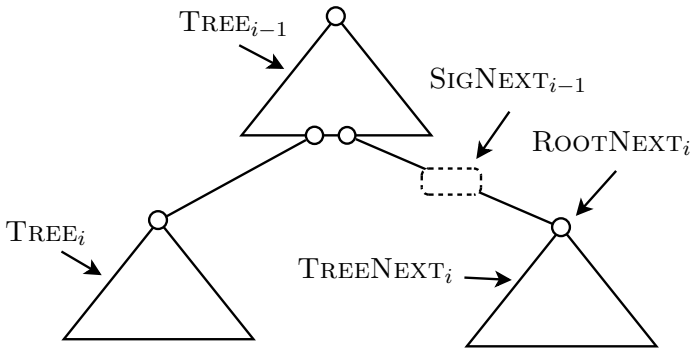

Figure 4.1: Distributed generation of SIGNEXT_{i-1}, the one-time signature of the root of TREENEXT_i.

We estimate the running time of the distributed root signing. The one-time signature of a root of a tree on layer i is generated using the Winternitz parameter w_{i-1} of layer $i-1$. According to Section 2.1 the generation of this signature requires $(2^{w_{i-1}} - 1)t_{w_{i-1}}$ hash function evaluations in the worst case. As shown in Section 2.4 the generation of the one-time signature key requires $t_{w_{i-1}} + 1$ calls to PRNG. Since each tree on layer i has 2^{H_i} leaves, the computation of its root signature is distributed across 2^{H_i} steps. Therefore, the total number of extra operations for each leaf of TREE_i to compute the root signature of TREENEXT_i is at most

$$c_{\text{sig}}(i) = \left\lceil \frac{(2^{w_{i-1}} - 1)t_{w_{i-1}}}{2^{H_i}} \right\rceil c_{\text{hash}} + \left\lceil \frac{t_{w_{i-1}} + 1}{2^{H_i}} \right\rceil c_{\text{prng}}. \tag{4.1}$$

Distributed root computation. We explain, how the root of TREENEXT_i is computed while TREEPREV_i is active. Both TREEPREV_i and TREENEXT_i have the same number of leaves. When a leaf of TREEPREV_i is used, the leaf with the same index in TREENEXT_i is calculated and passed to the treehash algorithm from Section 2.3.

If $i < T$, i.e. TREENEXT_i is not on the lowest level, the computation of each leaf of TREENEXT_i can also be distributed. This is explained next. Suppose that we want to construct the jth leaf of TREENEXT_i while we are using the jth leaf of TREEPREV_i. This computation is distributed across the leaves of the tree TREELOWER on layer $i+1$ whose root is signed using the jth leaf of TREEPREV_i. When the first leaf of TREELOWER is used, we determine the number of hash function evaluations and calls to PRNG required to compute the jth leaf of TREENEXT_i. Recall that the calculation of this leaf requires the computation of a Winternitz one-time key pair. We divide those numbers by $2^{H_{i+1}}$ to obtain the number of operations

we have to execute for each leaf of TreeLower. Whenever a leaf of TreeLower is used, the computation of the jth leaf of TreeNext is advanced by executing those operations.

Once the jth leaf of TreeNext_i is generated, it is passed to the treehash algorithm. This contributes to the construction of the root of TreeNext_i. This construction is complete, once we switch from TreePrev_i to Tree_i. So in fact, when Tree_i is used, the root of TreeNext_i is known. The distributed computation of the roots is visualized in Figure 4.2. While constructing TreeNext_i, we also perform the initialization steps of the authentication path algorithm of Chapter 3. That is, we store the authentication path of leaf 0 and the algorithm's initial state.

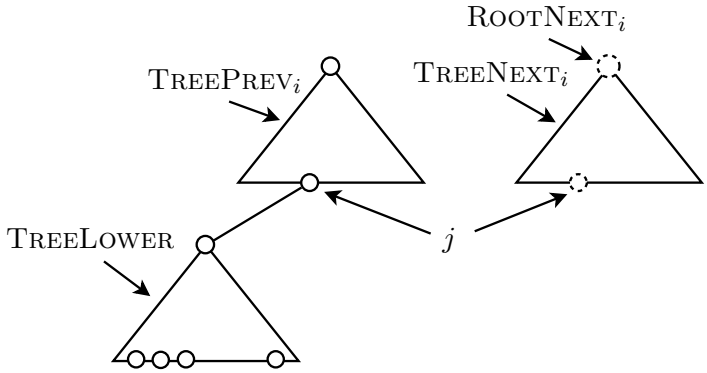

Figure 4.2: Distributed computation of RootNext_i. Leaf j of tree TreeNext_i is precomputed while using tree TreeLower. It is then used to partially compute RootNext_i.

We estimate the extra time required by the distributed root computation. Recall that for the generation of a leaf of TreeNext_i we first determine the corresponding Winternitz one-time key pair. This key pair is constructed using the Winternitz parameter w_i of layer i. The generation of the one-time signature key requires $t_{w_i}+1$ calls to PRNG. The generation of the one-time verification key requires $(2^{w_i}-1)t_{w_i}$ hash function evaluations and the computation of a leaf of TreeNext_i requires one additional evaluation of the hash function. This has been shown in Sections 2.1 and 2.4. Since TreeLower has $2^{H_{i+1}}$ leaves, the computation of a leaf of TreeNext_i can be distributed over $2^{H_{i+1}}$ steps. Therefore, the total number of extra operations for each leaf of TreeLower to compute a leaf of TreeNext_i is

$$c^{(1)}_{\text{leaf}}(i) = \left\lceil \frac{(2^{w_i}-1)t_{w_i}+1}{2^{H_{i+1}}} \right\rceil c_{\text{hash}} + \left\lceil \frac{t_{w_i}+1}{2^{H_{i+1}}} \right\rceil c_{\text{prng}}. \tag{4.2}$$

Once a leaf of TreeNext_i is found, it is passed to the treehash algorithm. By the results of Section 2.3 this costs at most

$$c^{(2)}_{\text{leaf}}(i) = H_i \cdot c_{\text{hash}} \tag{4.3}$$

additional evaluations of the hash function.

Distributed authentication path computation. Next, we describe the computation of the authentication path of the next leaf of tree TREE_i. We use the algorithm described in Chapter 3. This algorithm requires the computation of at most $(H_i - K_i)/2 + 1$ leaves per round to generate upcoming authentication paths on layer $i = 1, \ldots, T$. As described above, the computation of these leaves is distributed over the $2^{H_{i+1}}$ leaves (or steps) of tree TREELOWER, the current tree on the next lower layer $i + 1$. Again, this is possible only for leaves on layers $i = 1, \ldots, T - 1$. The computation of the leaves on layer T cannot be distributed.

When we use TREELOWER for the first time we calculate the number of hash function evaluations and calls to PRNG required to compute the $(H_i - K_i)/2 + 1$ leaves. Recall that we have to compute a Winternitz one-time key pair to obtain this leaf. Then we divide these costs by $2^{H_{i+1}}$ to estimate the number of operations we have to spend for each leaf of tree TREELOWER. At the beginning we don't know which leaves must be computed, we only know how may. Therefore, we have to interact with Algorithm 3.2. We perform the necessary steps to decide which leaf must be computed first. After computing this leaf we pass it to the authentication path algorithm which updates the treehash instance and determines which leaf must be computed next. This procedure is iterated until all required leaves are computed. The distributed authentication path computation is visualized in Figure 4.3.

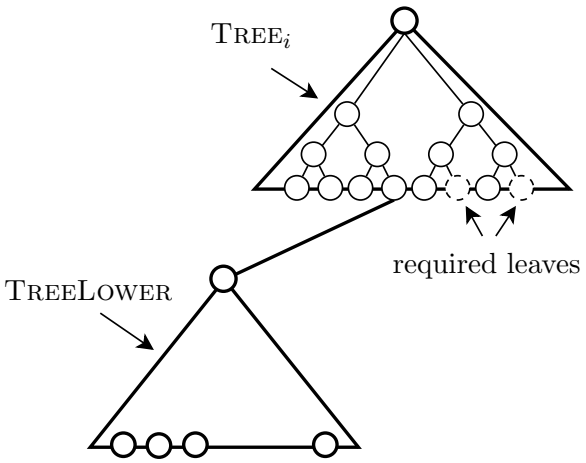

Figure 4.3: Distributed computation of the next authentication path. The $(H_i - K_i)/2 + 1$ required leaves are computed while using tree TREELOWER.

We estimate the cost of the distributed authentication path computation. The algorithm of Chapter 3 requires the computation of $(H_i - K_i)/2 + 1$ leaves for each authentication path. The leaves are computed using the Winternitz parameter w_i of layer i. The generation of one leaf requires $t_{w_i} + 1$ calls to PRNG and $(2^{w_i} - 1)t_{w_i} + 1$ hash function evaluations, see Sections 2.1 and 2.4. The computation of the those $(H_i - K_i)/2 + 1$ leaves is distributed over the $2^{H_{i+1}}$ steps in the tree on layer $i + 1$. Therefore, the total number of operations for each leaf of TREELOWER to compute

the $(H_i - K_i)/2 + 1$ leaves is

$$c^{(1)}_{\text{auth}}(i) = \frac{H_i - K_i + 2}{2} \cdot c^{(1)}_{\text{leaf}}(i). \tag{4.4}$$

The completed leaves are passed to the treehash algorithm that computes their ancestors. The algorithm of Chapter 3 requires at most $3(H_i - K_i - 1)/2 + 1$ evaluations of the hash function for the computation of ancestors. Another $H_i - K_i$ calls to PRNG are required to prepare upcoming seeds (see Section 5.1). These operations are not distributed but performed at once. Hence, the total number of operations for each leaf of TREE_i is at most

$$c^{(2)}_{\text{auth}}(i) = \frac{3(H_i - K_i) - 1}{2} \cdot c_{\text{hash}} + (H_i - K_i) \cdot c_{\text{prng}}. \tag{4.5}$$

Example 4.1. *This example illustrates how the distributed signature generation improves the signature generation time. Let $H_1 = \ldots = H_T = H$. Further, all layers use the same Winternitz parameter w and the same value for K. Let c_{sig} denote the worst case cost for generating a one-time signature with Winternitz parameter w, let c_{auth} denote the worst case cost for generating an authentication path in a tree of height H using K, and let c_{tree} denote the cost for partially computing the next tree. The worst case signature generation cost of the distributed signature generation method then is*

$$c_{sig} + c_{auth} + c_{tree} + \frac{(T-1)c_{sig} + (T-1)c_{auth} + (T-2)c_{tree}}{2^H}.$$

When the signature generation is not distributed, as in case of CMSS, the worst case signature generation cost is

$$Tc_{sig} + Tc_{auth} + (T-1)c_{tree}.$$

4.2 GMSS

The signature scheme that uses the distributed signature generation is called the *generalized Merkle signature scheme* (GMSS). In the following, we describe the GMSS key pair generation, signature generation and signature verification and analyze the costs.

GMSS key pair generation. We explain GMSS key pair generation, establish the size of the keys, and the cost for computing them. The following parameters are selected. The number T of layers, the heights $H_1, \ldots, H_T$ of the Merkle trees on each layer, the Winternitz parameters $w_1, \ldots, w_T$ for each layer, and the parameters $K_1, \ldots, K_T$ for the authentication path algorithm of Chapter 3.

We use the approach introduced in Section 2.4 and use a PRNG for the one-time signature generation. Therefore we must choose initial seeds SEED_i, for each layer

$i = 1, \ldots, T$. The GMSS public key is the root ROOT_1 of the single tree on layer $i = 1$. The GMSS private key consists of the following entries:

$$\begin{array}{rlcrl} \text{SEED}_i, & i = 1, \ldots, T & , & \text{SEEDNEXT}_i, & i = 2, \ldots, T \\ \text{SIG}_i, & i = 1, \ldots, T-1 & , & \text{ROOTNEXT}_i, & i = 2, \ldots, T \\ \text{AUTH}_i, & i = 1, \ldots, T & , & \text{AUTHNEXT}_i, & i = 2, \ldots, T \\ \text{STATE}_i, & i = 1, \ldots, T & , & \text{STATENEXT}_i, & i = 2, \ldots, T \end{array} \tag{4.6}$$

The seeds SEED_i are required for the generation of the one-time signature keys used to sign the data and the roots. The seeds SEEDNEXT_i are required for the distributed generation of subsequent roots. These seeds are available after the generation of the roots ROOTNEXT_i. The one-time signatures SIG_i of the roots are required for the GMSS signatures. The signatures SIG_i do not have to be computed explicitly. They are intermediate values during the computation of leaf 0 of tree TREE_{i-1}. The roots ROOTNEXT_i of the next tree on each layer are required for the distributed generation of the one-time signatures SIGNEXT_{i-1}. Also, the authentication path for the first leaf of the first and second tree on each layer is stored. STATE_i and STATENEXT_i denote the state of the authentication path algorithm of Chapter 3 required to compute authentication paths in trees TREE_i and TREENEXT_i, respectively. This state contains the seeds and the treehash instance and is initialized during the generation of the root.

The construction of a tree on layer i requires the computation of 2^{H_i} leaves and $2^{H_i} - 1$ evaluations of the hash function to compute inner nodes. Each leaf computation requires $(2^{w_i} - 1) \cdot t_{w_i} + 1$ hash function evaluations and $t_{w_i} + 1$ calls to PRNG. The total cost for one tree on layer i is given as

$$c_{\text{tree}}(i) = \left(2^{H_i}\left(t_{w_i}(2^{w_i} - 1) + 2\right) - 1\right) c_{\text{hash}} + 2^{H_i}\left(t_{w_i} + 1\right) c_{\text{prng}}. \tag{4.7}$$

Since we construct two trees on layers $i = 2, \ldots, T$ and one on layer $i = 1$, the total cost for the key pair generation is

$$c_{\text{keygen}} = \sum_{i=1}^{T} c_{\text{tree}}(i) + \sum_{i=2}^{T} c_{\text{tree}}(i). \tag{4.8}$$

The memory requirements of the keys depend on the output size n of the used hash function. A root is a single hash value and requires n bits. A seed also requires n bits. A one-time signature SIG_i requires $t_{w_{i-1}} \cdot n$ bits. According to Section 5.1, an authentication path together with the algorithm state requires

$$m_{\text{auth}}(i) = \left(5H_i + \left\lfloor \frac{H_i}{2} \right\rfloor - 5K_i - 2 + 2^{K_i}\right) \cdot n \text{ bits}. \tag{4.9}$$

For each layer $i = 2, \ldots, T$, we store two seeds, two authentication paths and algorithm states, one root and the one-time signature of one root. For layer $i = 1$, we

store one seed and one authentication path and algorithm state. The total sizes of the public and the private key are

$$m_{\text{pubkey}} = n \text{ bits}, \tag{4.10}$$

$$m_{\text{privkey}} = \left(\sum_{i=1}^{T} (m_{\text{auth}}(i) + 1) + \sum_{i=2}^{T} (m_{\text{auth}}(i) + t_{w_{i-1}} + 2) \right) n \text{ bits}. \tag{4.11}$$

GMSS signature generation. The GMSS signature generation is split into two parts, an online part and an offline part. In the online part, the signer constructs the current signature key from the seed SEED_T and generates the one-time signature SIG_T of the document to be signed. Then he prepares the signature as in Equation (4.12). The offline part takes care of the distributed computation of upcoming roots, one-time signatures of roots and authentication paths as described above.

$$\begin{aligned} \sigma_s = \big(s, \quad & \text{SIG}_T, Y_T, \text{AUTH}_T, \\ & \text{SIG}_{T-1}, Y_{T-1}, \text{AUTH}_{T-1} \\ & \quad \vdots \\ & \text{SIG}_1, Y_1, \text{AUTH}_1\big). \end{aligned} \tag{4.12}$$

The online part requires the generation of a single one-time signature. This signature is generated using the Winternitz parameter of the lowest layer T. According to Section 2.1, this requires

$$c_{\text{online}} = (2^{w_T} - 1)t_{w_T} \cdot c_{\text{hash}} + (t_{w_T} + 1)c_{\text{prng}} \tag{4.13}$$

operations in the worst case. We now compute the size of a GMSS signature. It consists of T authentication paths ($H_i \cdot n$ bits) and T one-time signatures ($t_{w_i} \cdot n$ bits), one for each layer $i = 1, \ldots, T$. Adding up results in

$$m_{\text{signature}} = \sum_{i=1}^{T} (H_i + t_{w_i}) \cdot n \text{ bits}. \tag{4.14}$$

To estimate the computational effort required for the offline part we assume the worst case where we have to advance one leaf on all layers $i = 1, \ldots, T$. The computation of the one-time signature SIGNEXT_i can be distributed for all layers $i = 1, \ldots, T-1$. The computation of the leaves required to construct the root ROOTNEXT_i can be distributed for all layers $i = 2, \ldots, T-1$. For layer $i = T$, the respective leaf of tree TREENEXT_T must be computed at once. Together with the hash function evaluations for the treehash algorithm, this requires at most

$$c_{\text{leaf}}^{(3)} = ((2^{w_T} - 1)t_{w_T} + H_T + 1)c_{\text{hash}} + (t_{w_T} + 1)c_{\text{prng}} \tag{4.15}$$

operations. The leaves required for the computation of upcoming authentication paths can be distributed for all layers $i = 1, \ldots, T-1$. For layer $i = T$, the

$(H_T - K_T)/2 + 1$ leaves must be computed at once. Together with the hash function evaluations for the treehash algorithm, this requires at most

$$\begin{aligned} c_{\text{auth}}^{(3)} =& \frac{H_T - K_T + 2}{2} \cdot c_{\text{leaf}}^{(3)} + \frac{3(H_T - K_T) - 1}{2} \cdot c_{\text{hash}} \\ &+ (H_T - K_T) \cdot c_{\text{prng}} \end{aligned} \tag{4.16}$$

operations. In summary, the number of operations required by the offline part in the worst case are

$$\begin{aligned} c_{\text{offline}} =& \sum_{i=2}^{T} c_{\text{sig}}(i) + \sum_{i=2}^{T-1} \left(c_{\text{leaf}}^{(1)}(i) + c_{\text{leaf}}^{(2)}(i) \right) + c_{\text{leaf}}^{(3)} \\ &+ \sum_{i=1}^{T-1} \left(c_{\text{auth}}^{(1)}(i) + c_{\text{auth}}^{(2)}(i) \right) + c_{\text{auth}}^{(3)}. \end{aligned} \tag{4.17}$$

The last step is to estimate the space required by the offline part. We have to store the partially constructed one-time signature SIGNEXT_i for layers $i = 1, \ldots, T-1$ which requires at most $t_{w_{i-1}} \cdot n$ bits. We also have to store the treehash stack for the generation of the root ROOTNEXT_i for layers $i = 2, \ldots, T$ which requires $H_i \cdot n$ bits. We further require memory to store partially constructed leaves. One leaf requires at most $t_{w_i} \cdot n$ bits. For the generation of ROOTNEXT_i we have to store at most one leaf for each layer $i = 2, \ldots, T-1$. For the authentication path, we have to store at most one leaf for each layer $i = 1, \ldots, T-1$. Note that since we compute the leaves required for the authentication path successively, we have to store only one partially constructed leaf at a time. Finally, we need to store the partial state STATENEXT_i of the authentication path algorithm for layers $i = 2, \ldots, T$ which requires at most $m_{\text{auth}}(i)$ bits (see Equation (4.9)). In summary, the memory required by the offline part in the worst case is

$$m_{\text{offline}} = \left(\sum_{i=2}^{T} \left(t_{w_{i-1}} + H_i + m_{\text{auth}}(i) \right) + \sum_{i=2}^{T-1} t_{w_i} + \sum_{i=1}^{T-1} t_{w_i} \right) \cdot n \text{ bits}. \tag{4.18}$$

GMSS signature verification. Since the main idea of GMSS is to distribute the signature generation, the signature verification doesn't change compared to CMSS. The verifier successively verifies a one-time signature and uses the corresponding authentication path and Equation (2.24) to compute the root. This is done until the root of the tree on the top layer is computed. If this root matches the signers public key, the signature is valid.

The verifier must verify T one-time signatures which in the worst case requires $(2^{w_i} - 1)t_{w_i}$ evaluations of the hash function, for $i = 1, \ldots, T$. Another H_i evaluations of the hash function are required to reconstruct the path to the root using the authentication path. In total, the number of hash function evaluations required in the worst case is

$$c_{\text{verify}} = \sum_{i=1}^{T} \left((2^{w_i} - 1)t_{w_i} + H_i \right) c_{\text{hash}}. \tag{4.19}$$

Performance. Table 4.1 shows timings and sizes of a Java implementation of GMSS. More details of the implementation can be found in Chapter 5. We use two layers of trees ($T = 2$), the W-OTS as one-time signature scheme, and SHA1 as hash function. The first column denotes the height of the Merkle tree and the Winternitz parameter used on layer one (H_1, w_1) and layer two (H_2, w_2). In this case $2^{H_1+H_2}$ signatures can be generated with one key pair. The remaining columns describe the signature size as well as the key pair generation, signature generation (average and worst case), and signature verification times. This table also shows the timings and sizes of CMSS according to Table 2.2.

Table 4.1: Timings for GMSS and CMSS using SHA1

$((H_1, H_2), (w_1, w_2))$	$m_{\text{signature}}$	c_{keygen}	$c_{\text{sign a.c.}}$	$c_{\text{sign w.c.}}$	c_{verify}
CMSS					
$((10, 10), (4, 4))$	2,128 bytes	1.5 sec	4.7 ms	8.5 ms	0.7 ms
$((15, 15), (4, 4))$	2,328 bytes	46.7 sec	6.5 ms	11.6 ms	0.8 ms
$((20, 20), (4, 4))$	2,528 bytes	24.9 min	8.2 ms	16.2 ms	0.8 ms
GMSS					
$((10, 10), (4, 4))$	2,128 bytes	2.4 sec	4.0 ms	4.7 ms	0.7 ms
$((15, 15), (4, 4))$	2,328 bytes	1.2 min	5.6 ms	6.2 ms	0.8 ms
$((20, 20), (4, 4))$	2,528 bytes	37.4 min	7.7 ms	8.6 ms	0.8 ms
GMSS					
$((10, 10), (8, 4))$	1,708 bytes	7.2 sec	4.0 ms	4.7 ms	3.4 ms
$((15, 15), (7, 4))$	1,968 bytes	2.5 min	5.6 ms	6.2 ms	1.7 ms
$((20, 20), (6, 4))$	2,248 bytes	57.5 min	7.7 ms	8.6 ms	1.6 ms
GMSS					
$((10, 10), (9, 5))$	1,508 bytes	13.1 sec	6.3 ms	7.6 ms	6.3 ms
$((15, 15), (8, 5))$	1,748 bytes	4.3 min	8.9 ms	10.0 ms	3.7 ms
$((20, 20), (7, 5))$	2,008 bytes	95.4 min	12.2 ms	13.8 ms	2.8 ms

This table shows that GMSS works as promised. The first half of Table 4.1 shows, that the worst case signature generation times are reduced drastically compared to CMSS. Also, the GMSS worst case signature generation times are extremely close to its average case signature generation times. Thus, GMSS provides balanced signature generation times. The reduced signature generation times enable us to choose larger Winternitz parameters without sacrificing efficiency as shown by the second half of Table 4.1. This results in smaller signatures. Using such large Winternitz parameters for CMSS is not advisable. For example, when using $(H_1, H_2) = (20, 20)$ and $(w_1, w_2) = (7, 5)$, CMSS signature generation takes 47.6 ms in the worst case.

5 Implementation and performance

The techniques discussed in the last two chapters aim at improving the practical performance of the Merkle signature scheme. This chapter presents timings of two implementations of these techniques to substantiate their practicability. The first implementation is a JCA conform Java implementation of GMSS. The second implementation is designed for 8-bit AVR microcontrollers. It incorporates the improvements from Section 2.4 and Chapter 3. We will present timings and sizes using different efficiency parameters to clarify the flexibility of the improvements of the last Chapters. We will also compare our implementations to the widely used signature schemes RSA and ECDSA. We begin by explaining some implementation specific refinements of GMSS.

5.1 Implementation specific refinements of GMSS

The first refinement regards the pseudo random number generator used for the generation of the one-time signature keys as explained in Section 2.4.

$$\begin{aligned} \text{PRNG} : \{0,1\}^n &\rightarrow \{0,1\}^n \times \{0,1\}^n \\ \text{SEED}_{\text{in}} &\mapsto (\text{RAND}, \text{SEED}_{\text{out}}) \end{aligned} \tag{5.1}$$

In our implementations, we use the hash based PRNG described in [19]. Let $g : \{0,1\}^* \rightarrow \{0,1\}^n$ be a cryptographic hash function with output length n bits. On input of a seed SEED_{in} this PRNG computes the pseudo random number RAND and the updated seed SEED_{out} as follows.

$$\begin{aligned} \text{RAND} &\leftarrow g(\text{SEED}_{\text{in}}), \\ \text{SEED}_{\text{out}} &\leftarrow (1 + \text{SEED}_{\text{in}} + \text{RAND}) \bmod 2^n \end{aligned} \tag{5.2}$$

Note that different implementations of GMSS must use the same PRNG in order to guarantee interoperability of the private keys.

The second refinement reduces the size of the GMSS signatures when using the Winternitz one-time signature scheme. The one-time verification keys are no longer included as part of the signature. This is possible because the verifier recomputes the one-time verification key during the Winternitz one-time signature verification (see Section 2.1.2). Instead of verifying the one-time signature using the allegedly authentic one-time verification key included in the signature, the verifier directly validates the authenticity of the recomputed one-time verification key using the authentication path. If the so computed root does not match the signers public key,

then either the one-time signature or the authentication path is invalid. The saving of this refinement is the size of a Winternitz one-time verification key.

The third refinement improves the runtime of the GMSS signature generation. Using the authentication path algorithm described in Chapter 3, the computation of left authentication nodes requires the computation of a leaf whenever the value s is even (see Line 3 in Algorithm 3.2). The leaf that must be computed is the sth leaf. In round s, the one-time signature key that corresponds to the sth leaf is used to generate the one-time signature of the document. Since the authentication path algorithm is executed after the sth signature has been generated and sent to the verifier, this one-time signature is available. Instead of computing the sth leaf from scratch, it is obtained by verifying this one-time signature which yields the sth one-time verification key Y_s and finally the sth leaf $g(Y_s)$. The saving of this refinement is the time required to generate the one-time signature.

The fourth refinement regards the leaves computed by the authentication path algorithm of Chapter 3 for the generation of right authentication nodes. The one-time signature keys corresponding to these leaves are generated using a PRNG as explained in Section 2.4. During the authentication path computation, leaves which are up to $3 \cdot 2^{H-K-1}$ steps away from the current leaf must be computed by the treehash instances. Calling PRNG that many times to obtain the correct seed is too inefficient. Instead we use the following scheduling strategy that requires $H - K$ calls to PRNG in each round to compute the seeds. Let SEED_s denote the seed required to compute the one-time key pair corresponding to the sth leaf. We have to store two seeds for each height $h = 0, \ldots, H - K - 1$. The first seed is denoted SEEDACTIVE_h and used to successively generate the leaves required for the computation of the height h authentication node currently constructed by TREEHASH_h. The second seed is denoted SEEDNEXT_h and used for upcoming right nodes on height h. SEEDNEXT is updated using PRNG in each round. During the initialization, we set $\text{SEEDNEXT}_h = \text{SEED}_{3 \cdot 2^h}$ for $h = 0, \ldots, H - K - 1$. In each round, at first all seeds SEEDNEXT_h are updated using PRNG. If in round s a new treehash instance is initialized on height h, we copy SEEDNEXT_h to SEEDACTIVE_h. In that case $\text{SEEDNEXT}_h = \text{SEED}_{s+1+3 \cdot 2^h}$ holds and thus is the correct seed to begin computing the next authentication node on height h. The time and space requirements of Algorithm 3.2 change as follows. We have to store additional $2(H - K)$ seeds and require additional $H - K$ calls to PRNG in each round.

5.2 Implementation in Java

GMSS, that is the Merkle signature scheme in combination with the improvements from Section 2.4, Section 2.5, Chapter 3, and Chapter 4, has been implemented as part of the Java Cryptographic Service Provider FlexiProvider [20]. It is therefore possible to integrate GMSS into any application that uses the Java Cryptographic Architecture [40] and Java Cryptography Extension [41]. A detailed description of the implementation can be found in [37, 7]. The implementation supports GMSS

using SHA1 and the whole SHA2 family as hash function. The FlexiProvider also includes an implementation of CMSS which was used for the timings shown in Table 2.2. Further, the FlexiProvider includes implementations of RSA and ECDSA which we use for the comparison. The test platform was an AMD Athlon 64 X2 5200+ EE running at 2.6 GHz with 2GB of memory using MS Windows XP and Java 1.6.0_02.

As described in the last chapter, GMSS can be customized using several parameters that influence the time required for key pair generation, signature generation, and verification as well as the space required for the signatures and the private key. These parameters are summarized in the parameter set

$$P = \big(T, (H_1, \ldots, H_T), (w_1, \ldots, w_T), (K_1, \ldots, K_T)\big), \tag{5.3}$$

where T denotes the number of layers, H_i denotes the height of the Merkle trees on layer i, w_i denotes the Winternitz parameter used on layer i, and K_i denotes the parameter for the authentication path algorithm of Chapter 3 used on layer i, for $i = 1, \ldots, T$. The number of signatures that can be generated using parameter set P is $2^{H_1+\cdots+H_T}$.

Tables 5.1, 5.2, and 5.3 show timings and sizes of GMSS, RSA, and ECDSA using SHA1, SHA224, and SHA256 as hash function, respectively. The key sizes for RSA and ECDSA were chosen such that all three schemes offer a similar security level [18]. The elliptic curves used for ECDSA are the curves secp160k1, secp224k1, and secp256k1 recommended by SECG [36]. In case of RSA and ECDSA the first column indicates the size of the modulus. In case of GMSS, the first column indicates the used parameter set P. We give timings for six parameter sets, two for up to 2^{20}, 2^{30}, and 2^{40} possible signatures, respectively. The parameter sets are

$$\begin{aligned}
2^{20} \text{ signatures:} \quad & P_1 = \big(2, (13,7), (10,7), (3,3)\big) \\
& P_2 = \big(2, (13,7), (7,4), (3,3)\big) \\
2^{30} \text{ signatures:} \quad & P_3 = \big(2, (15,15), (10,7), (3,3)\big) \\
& P_4 = \big(2, (15,15), (7,4), (3,3)\big) \\
2^{40} \text{ signatures:} \quad & P_5 = \big(3, (15,15,10), (10,9,7), (3,3,2)\big) \\
& P_6 = \big(3, (15,15,10), (7,7,4), (3,3,2)\big)
\end{aligned}$$

For each number of possible signatures, there is one parameter set that optimizes the memory requirements and one that optimizes the timings. The parameters K are chosen such that the memory requirements are minimal, i.e. $K = 2$ if H is even and $K = 3$ otherwise. In these tables m_{privkey} corresponds to the sum of m_{privkey} and m_{offline} estimated in Equations (4.11) and (4.18) in Chapter 4. Also, c_{sign} corresponds to the sum of c_{online} and c_{offline} estimated in Equations (4.13) and (4.17) in Chapter 4.

Table 5.1: Timings for GMSS, RSA, and ECDSA using SHA1

	Memory in bytes				Timings in ms	
Scheme	m_{pubkey}	m_{privkey}	$m_{\text{signature}}$	c_{keygen}	c_{sign}	c_{verify}
GMSS-P_1	75	5,967	1,268	2.5 min	12.8	11.5
GMSS-P_2	75	6,525	1,768	0.4 min	2.9	2.1
GMSS-P_3	75	9,619	1,468	13.6 min	25.0	10.9
GMSS-P_4	75	10,178	1,968	2.6 min	5.6	2.3
GMSS-P_5	84	15,098	2,072	21.4 min	17.8	17.2
GMSS-P_6	84	16,155	2,672	5.3 min	4.0	3.8
RSA-1024	162	635	128	0,4 sec	6.9	0.4
RSA-1248	190	763	156	0.7 sec	12.1	0.5
RSA-1536	226	921	192	1.2 sec	21.3	0.8
ECDSA-160	64	52	46	5.0 ms	4.6	5.6

Table 5.2: Timings for GMSS, RSA, and ECDSA using SHA224

	Memory in bytes				Timings in ms	
Scheme	m_{pubkey}	m_{privkey}	$m_{\text{signature}}$	c_{keygen}	c_{sign}	c_{verify}
GMSS-P_1	83	8,600	2,220	6.2 min	30.6	28.5
GMSS-P_2	83	9,607	3,172	1.1 min	6.9	4.8
GMSS-P_3	83	13,279	2,500	33.1 min	60.3	28.3
GMSS-P_4	83	14,286	3,452	6.1 min	13.6	4.6
GMSS-P_5	92	21,458	3,540	51.6 min	42.6	41.5
GMSS-P_6	92	23,442	4,688	12.7 min	9.7	9.1
RSA-2048	294	1217	256	3.8 sec	47.4	1.4
RSA-2432	342	1433	304	6.1 sec	77.4	1.9
RSA-4096	550	2376	512	48.0 sec	348.3	5.2
ECDSA-224	80	60	62	9.5 ms	8.8	10.7

Table 5.3: Timings for GMSS, RSA, and ECDSA using SHA256

	Memory in bytes				Timings in ms	
Scheme	m_{pubkey}	m_{privkey}	$m_{\text{signature}}$	c_{keygen}	c_{sign}	c_{verify}
GMSS-P_1	87	10,021	2,792	7.0 min	35.3	32.0
GMSS-P_2	87	11,427	4,040	1.2 min	7.9	5.6
GMSS-P_3	87	15,209	3,112	37.7 min	69.4	33.2
GMSS-P_4	87	16,616	4,360	7.1 min	15.7	5.6
GMSS-P_5	96	25,010	4,428	58.9 min	49.4	46.7
GMSS-P_6	96	27,694	5,932	14.7 min	11.2	9.9
RSA-2048	294	1,217	256	3.8 sec	47.4	1.4
RSA-3248	444	1,892	406	19.9 sec	178.2	3.3
RSA-4096	550	2,376	512	48.0 sec	348.3	5.2
ECDSA-256	88	64	70	12.7 ms	11.8	14.2

These tables show that GMSS is as efficient as RSA and ECDSA regarding signature generation and verification. RSA signature verification is faster only because small (16-bit) public exponents were used in the experiments. GMSS key pair generation is considerably slower than RSA and ECDSA key pair generation. However, we don't consider this as a setback because key pair generation is not performed frequently. Also, the comparatively large GMSS private keys can be easily stored on desktop PCs. The central issue of GMSS is still the size of the signatures. Even with the Winternitz trade-off that reduces the signature size by a factor w, the signature size is quadratic in the output length of the hash function. However, in many use cases this does not introduce a significant overhead. For example, it does not matter whether a 3,000 byte or a 300 byte signature is attached to a several hundred kilobyte large software update.

Next, we show timings of GMSS that clarify the trade-off between signature generation time and private key size possible using the parameter K in the authentication path algorithm of Chapter 3. Table 5.4 shows timings for the following six parameter sets using SHA1, SHA224, and SHA256 as hash function. The only difference to the previously used parameter sets is the increased value of K.

$$
\begin{aligned}
2^{20} \text{ signatures:} \quad & P'_1 = \big(2, (13,7), (10,7), (5,5)\big) \\
& P'_2 = \big(2, (13,7), (7,4), (5,5)\big) \\
2^{30} \text{ signatures:} \quad & P'_3 = \big(2, (15,15), (10,7), (5,5)\big) \\
& P'_4 = \big(2, (15,15), (7,4), (5,5)\big) \\
2^{40} \text{ signatures:} \quad & P'_5 = \big(3, (15,15,10), (10,9,7), (5,5,4)\big) \\
& P'_6 = \big(3, (15,15,10), (7,7,4), (5,5,4)\big)
\end{aligned}
$$

Table 5.4: Timings for GMSS using larger values for K

Scheme	Memory in bytes			Timings in ms		
	m_{pubkey}	m_{privkey}	$m_{\text{signature}}$	c_{keygen}	c_{sign}	c_{verify}
Using SHA1						
GMSS-P'_1	75	7,090	1,268	2.5 min	10.1	11.5
GMSS-P'_2	75	7,647	1,768	0.4 min	2.3	2.1
GMSS-P'_3	75	10,202	1,468	13.6 min	22.3	10.9
GMSS-P'_4	75	10,761	1,968	2.6 min	5.1	2.3
GMSS-P'_5	84	15,665	2,072	21.4 min	15.7	17.2
GMSS-P'_6	84	16,724	2,672	5.3 min	3.6	3.8
Using SHA224						
GMSS-P'_1	83	10,204	2,220	6.2 min	24.5	28.5
GMSS-P'_2	83	11,210	3,172	1.1 min	5.5	4.8
GMSS-P'_3	83	14,164	2,500	33.1 min	53.8	28.3
GMSS-P'_4	83	15,171	3,452	6.1 min	12.2	4.6
GMSS-P'_5	92	22,372	3,540	51.6 min	37.9	41.5
GMSS-P'_6	92	24,359	4,688	12.7 min	8.7	9.1
Using SHA256						
GMSS-P'_1	87	11,862	2,792	7.0 min	28.2	32.0
GMSS-P'_2	87	13,268	4,040	1.2 min	6.3	5.6
GMSS-P'_3	87	16,243	3,112	37.6 min	62.1	33.2
GMSS-P'_4	87	17,650	4,360	7.1 min	14.0	5.6
GMSS-P'_5	96	26,099	4,428	58.9 min	43.7	46.7
GMSS-P'_6	96	28,786	5,932	14.7 min	9.9	9.9

Code example. The following code example demonstrates how the FlexiProvider implementation of GMSS can be used. This example uses SHA1 as hash function and the parameter set $P_1 = \big(2, (13, 7), (10, 5), (3, 3)\big)$.

1. Include the required classes.

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.Signature;
import java.security.PublicKey;
import java.security.Security;
import de.flexiprovider.pqc.hbc.gmss.GMSSParameterSpec;
import de.flexiprovider.pqc.FlexiPQCProvider;

public class example {
  public static void main(String[] args) throws Exception {

    Security.addProvider(new FlexiPQCProvider());
```

2. Generate a `KeyPairGenerator` and `Signature` object.

```
    KeyPairGenerator keyPairGenerator = //
       KeyPairGenerator.getInstance("GMSSwithSHA1","FlexiPQC");

    Signature signature = //
       Signature.getInstance("GMSSwithSHA1","FlexiPQC");
```

3. Generate the parameter set.

```
    int T = 2;
    int[] H = {13, 7};
    int[] w = {10, 5};
    int[] K = {3, 3};

    GMSSParameterSpec P = new GMSSParameterSpec(T, H, w, K);
```

4. Initialize the `KeyPairGenerator` with the parameter set, generate the key pair, and extract the private and public key.

```
keyPairGenerator.initialize(P);

KeyPair gmssKeyPair = keyPairGenerator.genKeyPair();

PublicKey gmssPublicKey = gmssKeyPair.getPublic();
PrivateKey gmssPrivateKey = gmssKeyPair.getPrivate();
```

5. Choose the message to be signed.

```
byte[] messageBytes = "Hello world!".getBytes();
```

6. Initialize the `Signature` object with the private key for signing. Then sign the message.

```
signature.initSign(gmssPrivateKey);
signature.update(messageBytes);

byte[] signatureBytes = signature.sign();
```

7. Initialize the `Signature` object with the public key for verification. Then verify the message.

```
signature.initVerify(gmssPublicKey);
signature.update(messageBytes);

boolean valid = signature.verify(signatureBytes);

if (valid) System.out.println("Signature valid");
else System.out.println("Signature invalid");
  }
}
```

5.3 Implementation on a microcontroller

We now turn our attention to an implementation of the Merkle signature scheme on a microcontroller [34]. We call this implementation MMSS (Micro Merkle signature scheme). MMSS is designed to run on any 8-bit AVR microcontroller that offers 4 KBytes SRAM, about 4 KBytes EEPROM and at least 8 KBytes of program memory. For the performance measurements we used an Atmel ATmega128 microcontroller [3] that was clocked to 16MHz within the specification limits. Time-critical routines of the implementation, such as the hash function, were implemented in assembly and we used C to glue this routings together.

MMSS incorporates the improvements from Section 2.4 and Chapter 3. Multiple layers of trees are not supported by this implementation. This is because the maximum number of allowed write cycles for the EEPROM of the microcontroller is close to 2^{16} [2, 3] and it is therefore impossible to generate more than 2^{16} signatures during the life-span of the microcontroller. So it is sufficient to use a single Merkle tree of height at most 16 and the time required for the key pair generation is not an issue; especially because it is done on an desktop PC. MMSS uses AES based hash functions; a single block length construction with output length 128 bit for the generation of the one-time signatures and the Merkle tree and a double block length construction with output length 256 bit for the initial hashing of the document.

Remark 5.1. *As we will see in the next chapter, MSS requires a collision resistant hash function to be provably secure. This means that from a provable security point of view, using AES as hash function does not provide a sufficiently large security level. However, the best known attacks on the Merkle scheme require the computation of preimages and second preimages; being able to find collisions does not help. Hence using 128-bit hash functions yields a sufficiently large security level in practice.*

Single block length construction. The single block length hash in our scheme is constructed using the Matyas-Meyer-Oseas (MMO) construction [29]. The MMO construction is recursively defined as $f_{i+1} = E_{f_i}(M_i) \oplus M_i$ with E being the encryption function, M_i the current message block and f_0 an initialization vector (see Figure 5.1). Since we use AES as encryption function, the MMO construction yields a hash function with output length 128 bit.

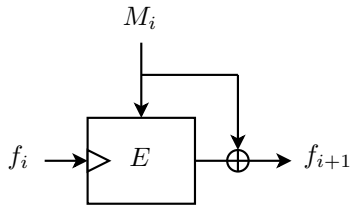

Figure 5.1: Single block length construction due to [29]. The output of the block cipher E is xored with the message block M_i. f_i, f_{i+1}, and M_i are each of bit length 128.

Double block length construction. For applications such as the initial digest generation in a signature scheme, collision resistance is needed and the security of single block length (SBL) constructions is not sufficient. For our implementation, we use the MDC-2 double length construction specified in the ISO/IEC 10118-2 standard. The standard envisions the usage of DES, but there is a variant using AES-128 [44] as depicted in Figure 5.2. This construction takes a block cipher with block length n bit and produces a hash function with $2n$ bit output length. In [39] the authors show, that an adversary needs at least $2^{3n/5}$ oracle queries to find a collision. However, the best practical attacks require 2^n queries. Since we use AES as encryption function, the MDC-2 construction yields a hash function with output length 256 bit.

The double block length construction is only used for initial digest generation. It can easily be replaced by a dedicated hash function resulting in a negligible performance loss but an increased code size.

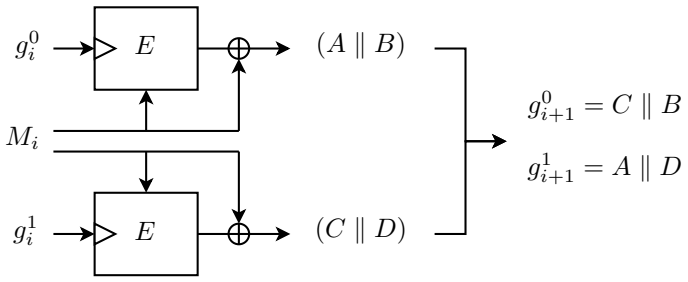

Figure 5.2: Double block length construction due to [44]. The outputs of the block cipher E are xored with the message block M_i and permuted. $g_i^0, g_i^1, g_{i+1}^0 g_{i+1}^1$, and M_i are each of bit length 128.

Performance. Similar to the Java implementation of GMSS, MMSS can be customized using the parameter set

$$P = (H, w, K). \tag{5.4}$$

Here, H denotes the height of the Merkle tree, w denotes the Winternitz parameter, and K denotes the parameter for the authentication path algorithm of Chapter 3. The number of signatures that can be generated using parameter set P is 2^H. We give timings for six parameter sets, three for up to 2^{10} and 2^{16} possible signatures, respectively. The parameter sets are

$$\begin{array}{lll} & P_1 = (10,2,2) & \\ 2^{10} \text{ signatures:} & P_2 = (10,2,4) & 2^{16} \text{ signatures:} \\ & P_3 = (10,4,4) & \end{array} \qquad \begin{array}{l} P_4 = (16,2,2) \\ P_5 = (16,2,4) \\ P_6 = (16,4,4) \end{array}$$

Table 5.5 show timings and sizes of our MMSS implementation and reference implementations of RSA and ECDSA. In case of MMSS, the first column indicates

the used parameter set P. In case of RSA and ECDSA the first column indicates the size of the modulus. Since the key pair generation is not performed on the microcontroller but on a desktop PC, we removed the column for the key pair generation time. We just remark that key pair generation for these parameters takes only a few minutes. New is the column m_{ROM}, which denotes the code size of the implementation.

Table 5.5: Timings for MMSS, RSA and ECDSA on an Atmel ATmega128 microcontroller.

	Memory in Bytes				Time in msec	
Scheme	m_{pubkey}	m_{privkey}	$m_{\text{signature}}$	m_{ROM}	c_{sign}	c_{verify}
MMSS-P_1	16	848	2290	6600	756	82
MMSS-P_2	16	876	2290	6600	598	82
MMSS-P_3	16	876	1234	6600	946	124
MMSS-P_4	16	1440	2350	6600	1230	85
MMSS-P_5	16	1472	2350	6600	1072	85
MMSS-P_6	16	1472	1330	6600	1665	127
RSA-1024 [22]	131	128	128	7400	5495	215
RSA-2048 [22]	259	256	256	10600	41630	970
ECDSA-160 [15]	40	21	40	43200	423	423
ECDSA-160 [27]	40	21	40	17900	1001	1218

This table shows that MMSS compares excellently to state of the art implementations of RSA and ECDSA. It provides the fastest signature verification times, even though the RSA timings were measured using a small public exponent. Signature generation is faster than RSA and comparable to ECDSA. As in case of GMSS, the MMSS private key and signature sizes are large compared to RSA and ECDSA. However, it is no problem to store them in the microcontroller's memory. Another benefit of MMSS is the small code size, which is also an issue when implementing on resource constrained devices.

In summary, the MMSS implementation shows that the improvements of the last chapters allow the Merkle signature scheme to be integrated in resource constrained devices while being highly competitive to currently used signature schemes.

6 Provable security

This section deals with the security of the Merkle signature scheme (MSS). One great advantage of the Merkle signature scheme is that it is provably secure. Based on the work of Coronado [13, 14], we will show in Section 6.1 that the Lamport–Diffie one-time signature scheme (LD-OTS) is existentially unforgeable under adaptive chosen message attacks as long as the used one-way function is preimage resistant. Then we show that the Merkle signature scheme is existentially unforgeable under adaptive chosen message attacks as long as the used hash function is collision resistant and the underlying one-time signature scheme is existentially unforgeable under adaptive chosen message attacks. On the basis of these reductions we estimate the security level of MSS combined with LD-OTS.

The security level of MSS combined with LD-OTS is essentially determined by the collision resistance of the hash function. Recent attacks on popular hash functions like MD5 [45] and SHA1 [46] have shown that collision resistance is a hard to achieve property. In Section 6.2 we describe a different construction method for Merkle trees. We call the signature scheme that uses the new construction SPR-MSS. We will show that SPR-MSS is existentially unforgeable under adaptive chosen message attacks as long as the used hash function is second-preimage resistant. This is a drastic improvement compared to the collision resistance requirement of the original Merkle scheme, since generic attacks to compute second preimages cannot exploit the birthday paradox. Thus, the new scheme has a significantly higher security level. After presenting the security reduction for SPR-MSS, we also estimate its security level. In Section 6.3, we compare the security level of MSS and SPR-MSS for different output lengths n of the hash function. We begin with some security notions and definitions.

Security notions for hash functions. We present three security notions for hash functions: preimage resistance, second preimage resistance, and collision resistance. The definitions are taken from [35]. We write $x \xleftarrow{\$} S$ for the experiment of choosing a random element from the finite set S with the uniform distribution. Let $\mathcal{G}$ be a family of hash functions, that is, a parameterized set

$$\mathcal{G} = \{g_k : \{0,1\}^* \to \{0,1\}^n | k \in K\} \tag{6.1}$$

where $n \in \mathbb{N}$ and K is a finite set. The elements of K are called *keys*. An *adversary* Adv is a probabilistic algorithm that takes any number of inputs.

We define *preimage resistance*. In fact, our notion of preimage resistance is a special case of the preimage resistance defined in [35] which is useful in our context.

Consider an adversary that attempts to find preimages of the hash functions in $\mathcal{G}$. The adversary takes as input a key $k \in K$ and the image $y = g_k(x)$ of a string $x \in \{0,1\}^n$. Both k and x are chosen randomly with the uniform distribution. The adversary outputs a *preimage* x' of y or `failure`. The success probability of this adversary is denoted by

$$\Pr[k \xleftarrow{\$} K, x \xleftarrow{\$} \{0,1\}^n, y \longleftarrow g_k(x), x' \xleftarrow{\$} \textsc{Adv}(k,y) : g_k(x') = y]. \tag{6.2}$$

Let t, ϵ be positive real numbers. The family $\mathcal{G}$ is called (t,ϵ) preimage resistant, if the success probability (6.2) of any adversary Adv that runs in a time t is at most ϵ.

Next, we define *second preimage resistance*. Consider an adversary that attempts to find second preimages of the hash functions in $\mathcal{G}$. The adversary takes as input a key $k \in K$ and a string $x \in \{0,1\}^{2n}$, both chosen randomly with the uniform distribution. He outputs a *second preimage* x' under g_k of $g_k(x)$ which is different from x or `failure`. The success probability of this adversary is denoted by

$$\Pr[k \xleftarrow{\$} K, x \xleftarrow{\$} \{0,1\}^{2n}, x' \xleftarrow{\$} \textsc{Adv}(k,x) : x \neq x' \wedge g_k(x) = g_k(x')]. \tag{6.3}$$

Let t, ϵ be positive real numbers. The family $\mathcal{G}$ is called (t,ϵ) second-preimage resistant, if the success probability (6.3) of any adversary Adv that runs in a time t is at most ϵ.

Finally, we define *collision resistance*. Consider an adversary that attempts to find collisions of the hash functions in $\mathcal{G}$. The adversary takes as input a key $k \in K$, chosen randomly with the uniform distribution. He outputs a *collision* of g_k, that is a pair $x, x' \in \{0,1\}^*$ with $x \neq x'$ and $g(x) = g(x')$ or `failure`. The success probability of this adversary is denoted by

$$\Pr[k \xleftarrow{\$} K, (x,x') \xleftarrow{\$} \textsc{Adv}(k) : x \neq x' \wedge g_k(x) = g_k(x')]. \tag{6.4}$$

Let t, ϵ be positive real numbers. The family $\mathcal{G}$ is called (t,ϵ) collision resistant, if the success probability (6.4) of any adversary Adv that runs in a time t is at most ϵ.

Signature schemes. Let Sign be a signature scheme. So Sign is a triple (Gen, Sig, Ver). Gen is the key pair generation algorithm. It takes as input 1^n, the string of n successive 1s where $n \in \mathbb{N}$ is a security parameter. It outputs a pair $(\mathsf{sk}, \mathsf{pk})$ consisting of a private key sk and a public key pk. Sig is the signature generation algorithm. It takes as input a message M and a private key sk. It outputs a signature σ for the message M. Finally, Ver is the verification algorithm. Its input is a message M, a signature σ and a public key pk. It checks whether σ is a valid signature for M using the public key pk. It outputs `true` if the signature is valid and `false` otherwise.

Existential unforgeability. Let $\textsc{Sign} = (\textsc{Gen}, \textsc{Sig}, \textsc{Ver})$ be a signature scheme and let $(\mathsf{sk}, \mathsf{pk})$ be a key pair generated by $\textsc{Gen}$. We define *existential unforgeability under adaptive chosen message attacks* of $\textsc{Sign}$. This security model assumes a very powerful forger. The forger has access to the public key and a signing oracle $\mathcal{O}(\mathsf{sk}, \cdot)$ that, in turn, has access to the private key. On input of a message the oracle returns the signature of that message. It is the goal of the forger to win the following game. The forger chooses at most q messages and lets the signing oracle find the signatures of those messages. The maximum number q of queries is also an input of the forger. The oracle queries may be adaptive, that is, they may depend on the oracles answers to previously queried messages. The forger outputs a pair (M', σ'). The forger wins if M' is different from all the messages in the oracle queries and if $\textsc{Ver}(M', \sigma', \mathsf{pk}) = \texttt{true}$. We denote such a forger by $\textsc{For}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$.

Let t and ϵ be positive real numbers and let q be a positive integer. The signature scheme $\textsc{Sign}$ is (t, ϵ, q) *existentially unforgeable under adaptive chosen message attacks* if for any forger that runs in a time t, the success probability for winning the above game (which depends on q) is at most ϵ. If $\textsc{Sign}$ has the above property it is also called a (t, ϵ, q) *signature scheme*.

For one-time signatures we must have $q = 1$ since the signature key of a one-time signature scheme must be used only once. For the Merkle signature scheme we must have $q \leq 2^H$.

6.1 Security of the Merkle signature scheme

We begin with a security reduction for the Lamport–Diffie one-time signature scheme (LD-OTS) followed by a security reduction for the Merkle signature scheme (MSS). Finally, we estimate the security level of the Merkle signature scheme combined with the Lamport–Diffie one-time signature scheme.

6.1.1 Security reduction for the Lamport–Diffie OTS

In this section we discuss the security of LD–OTS from Section 2.1. We slightly modify this scheme. Select a security parameter $n \in \mathbb{N}$. Let $K = K(n)$ be a finite set of parameters. Let

$$\mathcal{F} = \left\{ f_k : \{0,1\}^n \to \{0,1\}^n | k \in K \right\}$$

be a family of one-way functions. The key generation of the modified LD–OTS works as follows. On input of 1^n for a security parameter n a key $k \in K(n)$ is selected randomly with the uniform distribution. Then LD–OTS is used with the one-way function f_k. The secret and public keys are generated as described in Section 2.1. The key k is included in the public key. We show that the existential unforgeability under adaptive chosen message attacks of this LD-OTS variant can be reduced to the preimage resistance of the family $\mathcal{F}$.

Suppose that there exists a forger $\textsc{For}^{\mathcal{O}(X,\cdot)}(Y)$ of LD-OTS. Then an adversary $\textsc{Adv}_{\mathsf{Pre}}$ that determines preimages of functions in $\mathcal{F}$ can be constructed as follows.

Fix a security parameter n. The adversary $\text{ADV}_{\mathsf{Pre}}$ takes as input a key k and the image $y = f_k(x)$ of a string $x \in \{0,1\}^n$. Both k and x are selected randomly with the uniform distribution. A LD–OTS key pair (X, Y) is generated using the one-way function f_k. The public key Y is of the form $Y = (y_{n-1}[0], y_{n-1}[1], \ldots, y_0[0], y_0[1])$. The adversary selects indices $a \in \{0, \ldots, n-1\}$ and $b \in \{0,1\}$ randomly with the uniform distribution. He replaces the string $y_a[b]$ with the target string y. Next, $\text{ADV}_{\mathsf{Pre}}$ runs the forger $\text{FOR}^{\mathcal{O}(X,\cdot)}(Y)$ with the modified public key. If the forger asks its oracle to sign a message $M = (m_{n-1}, \ldots, m_0)$ and if $m_a = 1 - b$, then the adversary, playing the role of the oracle, signs the message and returns the signature. The adversary can sign this message since he knows the original key pair and because of $m_a = 1 - b$, the modified string in the public key is not used. However, if $m_a = b$ then the adversary cannot sign M. So his answer to the oracle query is `failure` which also causes the forger to abort. If the forger's oracle query was successful or if the forger does not ask the oracle at all the forger may produce a message $M' = (m'_{n-1}, \ldots, m'_0)$ and the signature $(\sigma'_{n-1}, \ldots, \sigma'_0)$ of that message. If $m'_a = b$, then σ'_a is the preimage of y which the adversary returns. Otherwise, the adversary returns `failure`. More formally, the adversary is presented in Algorithm 6.1.

Algorithm 6.1 $\text{ADV}_{\mathsf{Pre}}$

Input: $k \xleftarrow{\$} K$ and $y = f_k(x)$, where $x \xleftarrow{\$} \{0,1\}^n$
Output: x' such that $y = f_k(x)$ or `failure`

1. Generate an LD–OTS key pair (X, Y).
2. Choose $a \xleftarrow{\$} \{0, \ldots, n-1\}$ and $b \xleftarrow{\$} \{0,1\}$.
3. Replace $y_a[b]$ by y in the LD–OTS verification key Y.
4. Run $\text{FOR}^{\mathcal{O}(X,\cdot)}(Y)$.
5. When $\text{FOR}^{\mathcal{O}(X,\cdot)}(Y)$ asks its only oracle query with $M = (m_{n-1}, \ldots, m_0)$:
   a) **if** $m_a = (1 - b)$ **then** sign M and respond to the forger $\text{FOR}^{\mathcal{O}(X,\cdot)}(Y)$ with the signature σ.
   b) **else return** `failure`.
6. When $\text{FOR}^{\mathcal{O}(X,\cdot)}(Y)$ outputs a valid signature $\sigma' = (\sigma'_{n-1}, \ldots, \sigma'_0)$ for message $M' = (m'_0, \ldots, m'_{n-1})$:
   a) **if** $m'_a = b$ **then return** σ'_a as preimage of y.
   b) **else return** `failure`.

We now compute the success probability of the adversary $\text{ADV}_{\mathsf{Pre}}$. We denote by ϵ the forger's success probability for producing an existential forgery of the LD–OTS and by t its running time. By t_{GEN} and t_{SIG} we denote the times the LD–OTS requires for key and signature generation, respectively.

The adversary $\text{ADV}_{\mathsf{Pre}}$ is successful in finding a preimage of y if and only if the forger $\text{FOR}^{\mathcal{O}(X,\cdot)}(Y)$ queries the oracle with a message $M = (m_{n-1}, \ldots, m_0)$ with $m_a = (1 - b)$ (Line 5a) or if he queries the oracle not at all and if the forger returns a valid signature for message $M' = (m'_0, \ldots, m'_{n-1})$ with $m'_a = b$ (Line 6a). Since b is selected randomly with the uniform distribution, the probability for $m_a = (1 - b)$

is $1/2$. Since M' must be different from the queried message M, there exists at least one index c such that $m'_c = 1 - m_c$. $\text{ADV}_{\mathsf{Pre}}$ is successful if $c = a$, which happens with probability at least $1/2n$. Hence, the adversary's success probability for finding a preimage in a time $t_{\text{OW}} = t + t_{\text{SIG}} + t_{\text{GEN}}$, is at least $\epsilon/4n$. We have proved the following theorem.

Theorem 6.1. *Let $n \in \mathbb{N}$, let K be a finite parameter set, let t_{OW}, ϵ_{OW} be positive real numbers, and $\mathcal{F} = \{f_k : \{0,1\}^n \to \{0,1\}^n | k \in K\}$ be a family of $(t_{\text{OW}}, \epsilon_{\text{OW}})$ one-way functions. Then the LD–OTS variant that uses $\mathcal{F}$ is $(t_{\text{OTS}}, \epsilon_{\text{OTS}}, 1)$ existentially unforgeable under adaptive chosen message attacks with $\epsilon_{\text{OTS}} \leq 4n \cdot \epsilon_{\text{OW}}$ and $t_{\text{OTS}} = t_{\text{OW}} - t_{\text{SIG}} - t_{\text{GEN}}$ where t_{GEN} and t_{SIG} are the key generation and signing times of LD–OTS, respectively.*

6.1.2 Security reduction for Merkle's tree authentication scheme

This section discusses the security of the Merkle signature scheme. We modify the Merkle scheme slightly. Select a security parameter $n \in N$. Let $K = K(n)$ be a finite set of parameters. Let

$$\mathcal{G} = \{g_k : \{0,1\}^* \to \{0,1\}^n | k \in K\}$$

be a family of hash functions. The key generation of the modified MSS works as follows. On input of 1^n for a security parameter n a key $k \in K(n)$ is selected randomly with the uniform distribution. Then the Merkle signature scheme is used with the hash function g_k and some one-time signature scheme. The secret and public keys are generated as described in Section 2.2. The parameter k is included in the public key. We show that the existential unforgeability of this MSS variant under adaptive chosen message attacks can be reduced to the collision resistance of the family $\mathcal{G}$ and the existential unforgeability of the underlying one-time signature scheme.

We explain how an existential forger for the Merkle signature scheme can be used to construct an adversary that is either an existential forger for the underlying one-time signature scheme or a collision finder for a hash function in $\mathcal{G}$. The adversary takes as input a one-time signature scheme, a key $k \in K$ chosen randomly with the uniform distribution, and the Merkle tree height H. The adversary also takes as input a verification key Y_{OTS} and a signing oracle $\mathcal{O}_{\text{OTS}}(X_{\mathsf{OTS}}, \cdot)$, where $(X_{\mathsf{OTS}}, Y_{\mathsf{OTS}})$ is a key pair of the one-time signature scheme.

The adversary is allowed to query the oracle $\mathcal{O}_{\text{OTS}}(X_{\mathsf{OTS}}, \cdot)$ once. His goal is to output a collision for the hash function g_k or an existential forgery (M', σ') for the one-time signature scheme that can be verified using the verification key Y_{OTS}. He has access to an adaptive chosen message forger $\text{FOR}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$ for the MSS with hash function g_k and tree height H. The forger is allowed to ask 2^H queries to its signature oracle. The adversary is supposed to impersonate that oracle.

The adversary selects randomly with the uniform distribution an index c in the set $\{0, \ldots, 2^H - 1\}$. He generates a MSS key pair as usual with the only exception that

as the cth one-time verification key the one-time verification key Y_{OTS} from the input is used. Then the adversary invokes the adaptive chosen message forger for MSS with the hash function g_k and the MSS public key which he just generated. Without loss of generality, we assume that the forger queries the oracle 2^H times. The oracle answers are given by the adversary. When the forger asks for the ith signature, $i \neq c$, then the adversary produces this signatures using the signature keys which he generated before. However, when the forger asks for the cth signature, the adversary queries the oracle $\mathcal{O}_{\mathrm{OTS}}(X_{\mathsf{OTS}}, \cdot)$. Suppose that the forger is successful and outputs an existential forgery $(M', (s, \sigma', Y', A'))$ where s is the index of the one-time key pair used for this signature, σ' is the one-time signature, Y' is the verification key and A' is the authentication path. The adversary examines the Merkle signature (s, σ, Y, A) of M he returned in response to the forgers sth oracle query.

If $s = c$ and $(Y, A) = (Y', A')$, then the adversary returns (M', σ'). We show that this is an existential forgery of the one-time signature scheme with verification key Y_{OTS}. Since $s = c$ we have $Y' = Y = Y_{\mathsf{OTS}}$. So the one-time signature σ' can be verified using the one-time verification key Y_{OTS}. We also know that $M \neq M'$ holds and therefore (M', σ') is an existential forgery of the one-time signature scheme with verification key Y_{OTS}.

Algorithm 6.2 $\mathrm{ADV}_{\mathsf{CR},\mathsf{OTS}}$

Input: Key for the hash function $k \xleftarrow{\$} K$, height of the tree $H \geq 2$, one instance of the underlying OTS consisting of a verification key Y_{OTS} and the corresponding signing oracle $\mathcal{O}_{\mathrm{OTS}}(X_{\mathsf{OTS}}, \cdot)$.
Output: A collision of g_k, an existential forgery for the supplied instance of the OTS, or `failure`

1. Set $c \xleftarrow{\$} \{0, \ldots, 2^H - 1\}$.
2. Generate OTS key pairs $(X_j, Y_j), j = 0, \ldots, 2^H - 1, j \neq c$ and set $Y_c \leftarrow Y_{\mathsf{OTS}}$.
3. Complete the Merkle key pair generation and obtain $(\mathsf{sk}, \mathsf{pk})$.
4. Run $\mathrm{FOR}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$.
5. When $\mathrm{FOR}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$ asks its qth oracle query $(0 \leq q \leq 2^H - 1)$:
   a) **if** $q = c$ **then** query the signing oracle $\mathcal{O}_{\mathrm{OTS}}(X_{\mathsf{OTS}}, \cdot)$.
   b) **else** compute the one-time signature σ using the qth signature key X_q.
   c) Return the corresponding Merkle signature to the forger.
6. If the forger outputs an existential forgery $(M', (s, \sigma', Y', A'))$, examine the Merkle signature (s, σ, Y, A) returned in response to the forgers sth oracle query.
   a) **if** $(Y', A') \neq (Y, A)$ **then return** a collision of g_k.
   b) **else**
      i. **if** $s = c$ **then return** (M', σ') as forgery for the supplied instance of the one-time signature scheme.
      ii. **else return** `failure`.

If $(Y, A) \neq (Y', A')$, then the adversary can construct a collision for the hash function g_k as follows. Consider the path $B = (B_0 = g_k(Y), B_1, \ldots, B_H)$ from Y in the Merkle tree to its root constructed using the hash function g_k and the authentication path $A = (A_0, \ldots, A_{H-1})$. Compare it to the path $B' = (B'_0 = g_k(Y'), B'_1, \ldots, B'_H)$ from Y' in the Merkle tree to its root constructed using the authentication path $A' = (A'_0, \ldots, A'_{H-1})$. First assume that B and B' are different. For example, this is true when $Y \neq Y'$. Since $B_H = B'_H$ is the MSS public key, there is an index $0 \leq i < H$ with $B_{i+1} = B'_{i+1}$ and $B_i \neq B'_i$. Since B_{i+1} is the hash value of the concatenation of B_i and A_i (in the appropriate order), and since B'_{i+1} is the hash value of the concatenation of B'_i and A'_i (in the appropriate order), a collision of g_k is found. Next, assume that B and B' are equal. Therefore $g_k(Y) = B_0 = B'_0 = g_k(Y')$ holds. If $Y \neq Y'$ a collision is found. If $Y = Y'$ then A and A' are different. Assume that $A_i \neq A'_i$ for some index $i < H$. Since B_{i+1} is the hash value of the concatenation of B_i and A_i (in the appropriate order), and since B'_{i+1} is the hash value of the concatenation of B'_i and A'_i (in the appropriate order) again a collision is found. That collision is returned by the adversary. In all other cases the adversary returns failure. Algorithm 6.2 summarizes our description.

We now estimate the success probability of the adversary $\text{ADV}_{\mathsf{CR},\mathsf{OTS}}$. In the following, ϵ denotes the success probability and t the running time of the forger. Also, $t_{\text{GEN}}, t_{\text{SIG}}$, and t_{VER} denote the times MSS requires for key generation, signature generation, and verification, respectively.

If $(Y', A') \neq (Y, A)$, then the adversary returns collision. His (conditional) probability ϵ_{CR} for returning a collision in a time $t_{\text{CR}} = t + 2^H \cdot t_{\text{SIG}} + t_{\text{VER}} + t_{\text{GEN}}$ is at least ϵ. If $(Y', A') = (Y, A)$ the adversary returns an existential forgery if $s = c$. His (conditional) probability ϵ_{OTS} for finding an existential forgery with verification key Y_{OTS} in a time $t_{\text{OTS}} = t + 2^H \cdot t_{\text{SIG}} + t_{\text{VER}} + t_{\text{GEN}}$ is at least $\epsilon \cdot 1/2^H$. Since both cases are mutually exclusive, one of them occurs with probability at least $1/2$. So we have proved the following theorem.

Theorem 6.2. *Let K be a finite set, let $H \in \mathbb{N}$, $t_{\text{CR}}, t_{\text{OTS}}, \epsilon_{\text{CR}}, \epsilon_{\text{OTS}} \in \mathbb{R}_{>0}$, $\epsilon_{\text{CR}} \leq 1/2$, $\epsilon_{\text{OTS}} \leq 1/2^{H+1}$, and let $\mathcal{G} = \{g_k : \{0,1\}^* \to \{0,1\}^n | k \in K\}$ be a family of $(t_{\text{CR}}, \epsilon_{\text{CR}})$ collision resistant hash functions. Consider MSS using a $(t_{\text{OTS}}, \epsilon_{\text{OTS}}, 1)$ one-time signature scheme. Then MSS is a $(t, \epsilon, 2^H)$ signature scheme with*

$$\epsilon \leq 2 \cdot \max\{\epsilon_{\text{CR}}, 2^H \cdot \epsilon_{\text{OTS}}\} \tag{6.5}$$

$$t = \min\{t_{\text{CR}}, t_{\text{OTS}}\} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}. \tag{6.6}$$

This theorem tell us that if there is no adversary that can break the collision resistance of the family $\mathcal{G}$ in a time at most t_{CR} with probability greater than ϵ_{CR} and there is no adversary that is able to produce an existential forgery for the one-time signature scheme used in MSS in a time at most t_{OTS} with probability greater than ϵ_{OTS}, then there exists no forger for MSS running in a time at most $\min\{t_{\text{CR}}, t_{\text{OTS}}\} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}$ and success probability greater then $2 \cdot \max\{\epsilon_{\text{CR}}, 2^H \cdot \epsilon_{\text{OTS}}\}$.

6.1.3 Security level of the Merkle signature scheme

The goal of this section is to estimate the security level of the Merkle signature scheme when used with the Lamport–Diffie one-time signature scheme for a given output length n of the hash function. Let $b \in \mathbb{N}$. We say that MSS has security level 2^b if the expected number of hash function evaluations required for the generation of an existential forgery is at least 2^b. This security level can be computed as t/ϵ where t is the running time of an existential forger and ϵ is its success probability. We also say that the signature scheme has b *bits of security* or that the *bit security* is b. In this section let $\epsilon_{\text{CR}}, t_{\text{CR}}, \epsilon_{\text{OW}}, t_{\text{OW}} \in \mathbb{R}_{>0}$, let K be a finite set, and let

$$\mathcal{G} = \left\{ g_k : \{0,1\}^* \to \{0,1\}^n | k \in K \right\} \tag{6.7}$$

be a family of $(t_{\text{CR}}, \epsilon_{\text{CR}})$ collision resistant and $(t_{\text{OW}}, \epsilon_{\text{OW}})$ preimage resistant hash functions.

Since we consider MSS using LD-OTS, we first combine Theorems 6.1 and 6.2. This is achieved by substituting the values for ϵ_{OTS} and t_{OTS} from Theorem 6.1 in Equations (6.5) and (6.6) from Theorem 6.2. This yields

$$\begin{aligned} \epsilon &\leq 2 \cdot \max\left\{\epsilon_{\text{CR}}, 2^H \cdot 4n \cdot \epsilon_{\text{OW}}\right\} && (6.8) \\ t &= \min\left\{t_{\text{CR}}, t_{\text{OW}}\right\} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}. && (6.9) \end{aligned}$$

Note that we can replace t_{OTS} by t_{OW} rather than $t_{\text{OW}} - t_{\text{SIG}} - t_{\text{GEN}}$, since the time LD-OTS requires for signature and key generation is already included in the signature and key generation time of the MSS in Theorem 6.2. We also require $\epsilon_{\text{CR}} \leq 1/2$ and $\epsilon_{\text{OW}} \leq 1/(2^{H+1} \cdot 4n)$ to ensure $\epsilon \leq 1$.

To estimate the security level, we need explicit values for the key pair generation, signature generation and verification times of MSS using LD-OTS. We will use the following upper bounds.

$$t_{\text{GEN}} \leq 2^H \cdot 6n, \quad t_{\text{SIG}} \leq 4n(H+1), \quad t_{\text{VER}} \leq n + H \tag{6.10}$$

We also make assumptions for the values of $(t_{\text{CR}}, \epsilon_{\text{CR}})$ and $(t_{\text{OW}}, \epsilon_{\text{OW}})$. We distinguish between attacks that use classical computers only and attacks with quantum computers.

Using classical computers. In our security analysis of MSS we assume that the hash functions under consideration have output length n and only admit generic attacks against their preimage and collision resistance. Those generic attacks are exhaustive search and the birthday attack. When classical computers are used, then a birthday attack that inspects $2^{n/2}$ hash values has a success probability of approximately $1/2$. We therefore assume that our hash functions are

$$(t_{\text{CR}}, \epsilon_{\text{CR}}) = (2^{n/2}, 1/2) \tag{6.11}$$

collision resistant. Also, an exhaustive search of $2^{n/2}$ random strings yields a preimage of a given hash value with probability $1/2^{n/2}$. We therefore assume that our hash functions are

$$(t_{\mathrm{OW}}, \epsilon_{\mathrm{OW}}) = (2^{n/2}, 1/2^{n/2}) \tag{6.12}$$

preimage resistant. In this situation, we prove the following theorem.

Theorem 6.3 (Classical case)**.** *The security level of the Merkle signature scheme combined with the Lamport-Diffie one-time signature scheme is at least*

$$b = n/2 - 1 \tag{6.13}$$

if the height of the Merkle tree is at most $H \leq n/3$ and the output length of the hash function is at least $n \geq 87$.

To prove Theorem 6.3 we use our assumption and Equations (6.8) and (6.9) and obtain the following estimate for the security level.

$$\frac{t}{\epsilon} \geq \frac{2^{n/2} - 2^H \cdot t_{\mathrm{SIG}} - t_{\mathrm{VER}} - t_{\mathrm{GEN}}}{2 \cdot \max\{1/2, 2^H \cdot 4n \cdot 1/2^{n/2}\}}. \tag{6.14}$$

Using $H \leq n/3$, the maximum in the denominator is $1/2$ as long as

$$n/3 \leq n/2 - \log_2 4n - 1 \tag{6.15}$$

which holds for $n \geq 53$. Using the upper bounds for t_{SIG}, t_{VER}, and t_{GEN} estimated above, Equation (6.14) implies

$$\frac{t}{\epsilon} \geq 2^{n/2} - 2^H \cdot 4n(H+1) - (n+H) - 2^H \cdot 6n. \tag{6.16}$$

Using $H \leq n/3$, the desired lower bound for the security level of $2^{n/2-1}$ holds as long as

$$2^{n/3}(4/3 \cdot n^2 + 4n) + 4/3 \cdot n + 2^{n/3} \cdot 6n \leq 2^{n/2-1} \tag{6.17}$$

which is true for $n \geq 87$.

Using quantum computers. Again, we assume that our hash functions have output length n and only admit generic attacks against their collision and preimage resistance. However, when quantum computers are available, the Grover algorithm [21] can be used in those generic attacks. Grovers algorithm requires $2^{n/3}$ evaluations of the hash function to find a collision with probability at most $1/2$. So we assume that our hash functions are

$$(t_{\mathrm{CR}}, \epsilon_{\mathrm{CR}}) = (2^{n/3}, 1/2) \tag{6.18}$$

collision resistant. By virtue of Grover's algorithm we may also assume that our hash functions are

$$(t_{\mathrm{OW}}, \epsilon_{\mathrm{OW}}) = (2^{n/3}, 1/2^{n/3}) \tag{6.19}$$

preimage resistant, see [6] Chapter 2 "Quantum computing". In this situation, we prove the following theorem.

Theorem 6.4 (Quantum case)**.** *The security level of the Merkle signature scheme combined with the Lamport-Diffie one-time signature scheme is at least*

$$b = n/3 - 1 \tag{6.20}$$

if the height of the Merkle tree is at most $H \leq n/4$ and the output length of the hash function is at least $n \geq 196$.

To prove Theorem 6.4 we use the same approach as for the proof of Theorem 6.3. We use our assumption on the hash function and Equations (6.8) and (6.9) and obtain the following estimate for the security level.

$$\frac{t}{\epsilon} \geq \frac{2^{n/3} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}}{2 \cdot \max\{1/2, 2^H \cdot 4n \cdot 1/2^{n/3}\}}. \tag{6.21}$$

Using $H \leq n/4$, the maximum in the denominator is $1/2$ as long as

$$n/4 \leq n/3 - \log_2 4n - 1 \tag{6.22}$$

which holds for $n \geq 119$. Using the upper bounds for t_{SIG}, t_{VER}, and t_{GEN} estimated above, Equation (6.21) implies

$$\frac{t}{\epsilon} \geq 2^{n/3} - 2^H \cdot 4n(H+1) - (n+H) - 2^H \cdot 6n. \tag{6.23}$$

Using $H \leq n/4$, the desired lower bound for the security level of $2^{n/3-1}$ holds as long as

$$2^{n/4}(n^2 + 4n) + 5/4 \cdot n + 2^{n/4} \cdot 6n \leq 2^{n/3-1} \tag{6.24}$$

which is true for $n \geq 196$.

6.2 Merkle signatures based on second-preimage resistance

According to the last section, the security level of MSS combined with LD-OTS is determined by the collision resistance of the hash function family. Recent attacks on the collision resistance of popular hash functions such as MD5 and SHA1 indicate that collision resistance is a hard to achieve goal. We will now describe a different construction method for Merkle trees that adds randomness to each inner node. We call the resulting signature scheme SPR-MSS [17]. Then we show that SPR-MSS is existentially unforgeable under adaptive chosen message attacks as long as the used hash function is second-preimage resistant and the underlying one-time signature scheme is existentially unforgeable under adaptive chosen message attacks. In the following, let

$$\mathcal{G} = \left\{ g_k : \{0,1\}^* \to \{0,1\}^n | k \in K \right\}$$

be a family of hash functions.

6.2.1 The SPR-MSS construction

In this section we describe the new construction method for the Merkle authentication tree, which is inspired by the XOR-tree proposed in [5]. We call the resulting tree SPR-Merkle tree. The new construction differs from the original construction in two ways.

1. A leaf of the SPR-Merkle tree is not the hash value of the concatenation of the bit strings in the one-time verification key, but the bit strings themselves. The SPR-Merkle tree is constructed starting directly from these bit strings.

2. Before applying the hash function to the concatenation of two child nodes to compute their parent, both child nodes are xored with a randomly chosen mask.

SPR-MSS also uses the parameter $H \geq 2$ to decide on the number of signatures that can be generated with one key pair, i.e. 2^H many. In the following, we assume that each one-time verification key consists of 2^l bit strings of length n. Since there are 2^H one-time verification keys, a SPR-Merkle tree has 2^{H+l} leaves and height $H + l$. The nodes are denoted by $\nu_h[j]$, where $h = 0, \ldots, H + l$ denotes the height of the node in the tree (leaves have height 0 and the root has height $H + l$) and $j = 0, \ldots, 2^{H+l-h} - 1$ denotes the position of the node on that height, counting from left to right.

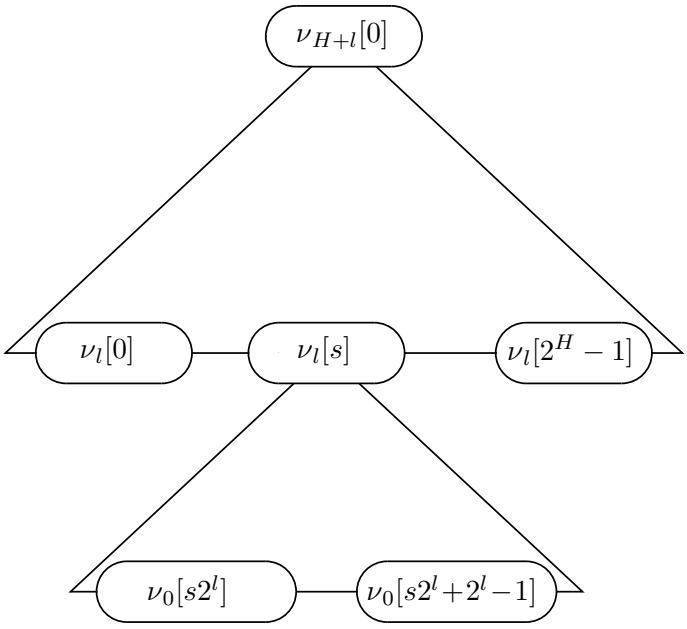

Figure 6.1: SPR-Merkle tree

The sth verification key Y_s contains 2^l leaves of the SPR-Merkle tree denoted by $\nu_0[s2^l], \ldots, \nu_0[s2^l+2^l-1]$. These leaves can be used to compute the inner node $\nu_l[s]$ on height l. Unlike in the original MSS, where the signer includes the authentication path for the sth leaf in the signature, the signer now has to include an authentication path for the inner node $\nu_l[s]$. Figure 6.1 shows an example of an SPR-Merkle tree.

Remark 6.5. *In case the number of bit strings L in the verification keys of the chosen OTS is not a power of 2, the resulting SPR-Merkle tree has height $H + \lceil \log_2 L \rceil$. The SPR-Merkle tree is constructed such that the subtrees below the 2^H nodes $\nu_h[j]$ are unbalanced trees of height $\lceil \log_2 L \rceil$.*

For the construction of inner nodes we choose two masks

$$v_h[0], v_h[1] \in_R \{0,1\}^n \tag{6.25}$$

for each height $h = 1, \ldots, H + l$ uniform at random. When constructing an inner node on height h, its left child on height is xored with $v_h[0]$ and its right child is xored with $v_h[1]$. Then the results are concatenated and the hash function is applied. The construction rule for inner nodes of an SPR-Merkle tree is

$$\nu_h[j] = g_k\Big(\big(\nu_{h-1}[2j] \oplus v_h[0]\big) \parallel \big(\nu_{h-1}[2j+1] \oplus v_h[1]\big)\Big) \tag{6.26}$$

for $h = 1, \ldots, H + l$ and $j = 0, \ldots, 2^{H+l-h} - 1$. The computation of inner nodes is visualized in Figure 6.2.

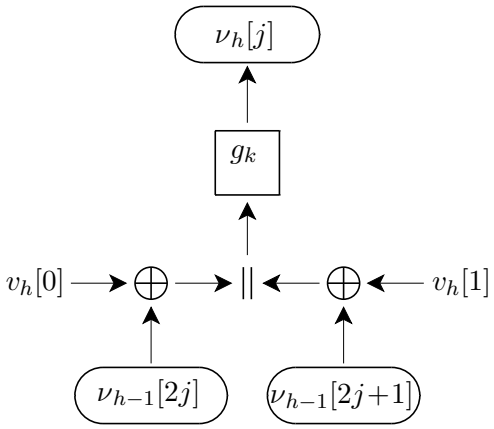

Figure 6.2: Construction of inner nodes in a SPR-Merkle tree

SPR-MSS key pair generation. The key pair generation works as follows. First choose $H \geq 2$. Next compute 2^H one-time key pairs (X_j, Y_j), for $j = 0, \ldots, 2^H - 1$, where each one-time verification key consists of 2^l bit stings of length n. Then choose a key for the hash function $k \in_R K$ and masks $v_h[0], v_h[1] \in_R \{0,1\}^n$ uniformly at random for $h = 1, \ldots, H + l$. The SPR-MSS private key consists of the 2^H one-time signature keys X_j or the seed for PRNG used to generate them (see Section 2.4). The SPR-MSS public key consists of the key for the hash function k, the XOR masks $v_1[0], v_1[1], \ldots, v_{H+l}[0], v_{H+l}[1]$, and the root $\nu_{H+l}[0]$. The root of the SPR-Merkle tree is generated using the treehash algorithm of Section 2.3. This algorithm must be adjusted such that it correctly computes inner nodes using the masks $v_h[0], v_h[1]$.

Compared to the original MSS, SPR-MSS key pair generation requires $2^H(2^l - 2)$ additional evaluations of the hash function and $2^H(2^{l+1} - 2)$ additional XOR computations to compute the inner nodes $\nu_l[0], \ldots, \nu_l[2^H - 1]$. Further, $2^{H+1} - 2$ additional XOR operations are required to compute the root $\nu_{H+l}[0]$. The size of the private key doesn't change. The SPR-MSS public key also contains the XOR masks. Hence its size increases by $2(H + l) \cdot n$ bits.

SPR-MSS signature generation. The SPR-MSS signature generation is very similar to the MSS signature generation described in Section 2.2. To sign a message M, the signer first computes its n-bit digest d. Then he generates the one-time signature σ_{OTS} of the digest using the sth one-time signature key X_s, $s \in \{0, \ldots, 2^H - 1\}$. The SPR-MSS signature contains this one-time signature and the corresponding one-time verification key Y_s. The signature also contains the authentication path $A_s = (a_0, \ldots, a_{H-1})$ for inner node $\nu_l[s]$ on height l given as

$$a_h = \begin{cases} \nu_{h+l}[s/2^h - 1] & \text{, if } \lfloor s/2^h \rfloor \equiv 1 \bmod 2 \\ \nu_{h+l}[s/2^h + 1] & \text{, if } \lfloor s/2^h \rfloor \equiv 0 \bmod 2 \end{cases} \tag{6.27}$$

for $h = 0, \ldots H - 1$. The respective algorithm used for the computation of the authentication paths must be adjusted such that it correctly computes inner nodes using the masks $v_h[0], v_h[1]$. In total, the sth SPR-MSS signature is given as

$$\sigma_s = \big(s, \sigma_{\text{OTS}}, Y_s, A_s)\big). \tag{6.28}$$

Compared to the original MSS, SPR-MSS signature generation requires two additional XOR operations for each inner node of the SPR-Merkle tree that must be computed during signature generation, e.g. by the authentication path algorithm.

SPR-MSS signature verification. The verification of a SPR-MSS signature is also quite similar to the MSS signature verification. First the verifier verifies the one-time signature of message M using the supplied verification key Y_s and the verification algorithm of the respective one-time signature scheme.

Then he verifies the authenticity of Y_s as follows: first he uses the 2^l bit strings in $Y_s = (\nu_0[s2^l], \ldots, \nu_0[s2^l + 2^l - 1])$ to compute the inner node $\nu_l[s]$ on height l as follows

$$\nu_h[j] = g_k\big(\nu_{h-1}[2j] \oplus v_h[0] \parallel \nu_{h-1}[2j+1] \oplus v_h[1]\big) \tag{6.29}$$

for $h = 1, \ldots, l$ and $j = s2^{l-h}, \ldots, s2^{l-h} + 2^{l-h} - 1$. Then he uses the authentication path A_s and recomputes the path from $\nu_l[s]$ to the root $\nu_{H+l}[0]$ as

$$p_h = \begin{cases} g_k\Big(\big(a_{h-1} \oplus v_h[0]\big) \parallel \big(p_{h-1} \oplus v_h[1]\big)\Big) & \text{, if } \lfloor s/2^{h-1} \rfloor \equiv 1 \bmod 2 \\ g_k\Big(\big(p_{h-1} \oplus v_h[0]\big) \parallel \big(a_{h-1} \oplus v_h[1]\big)\Big) & \text{, if } \lfloor s/2^{h-1} \rfloor \equiv 0 \bmod 2 \end{cases} \tag{6.30}$$

for $h = 1, \ldots H$ and $p_0 = \nu_l[s]$. The signature is valid if p_H equals the signers public root $\nu_{H+l}[0]$ and the verification of the one-time signature is successful.

Compared to the original MSS, SPR-MSS signature verification requires $2^l - 2$ additional evaluations of the hash function and $2^{l+1} - 2$ additional XOR computations to compute the node $\nu_l[s]$. Further, $2H$ additional XOR operations are required to recompute the path from $\nu_l[s]$ to the root $\nu_{H+l}[0]$.

6.2.2 Security reduction for SPR-MSS

We now reduce the existential unforgeability of SPR-MSS under adaptive chosen message attacks to the second preimage resistance of the hash function family

$$\mathcal{G} = \big\{ g_k : \{0,1\}^* \to \{0,1\}^n | k \in K \big\}$$

and the existential unforgeability of the underlying one-time signature scheme. The reduction is similar to the reduction for the original MSS explained in Section 6.1.2. The difference is, that the randomly chosen masks $v_h[0], v_h[1] \in_R \{0,1\}^n, h = 1, \ldots, H + l$ allow us to insert a predefined first preimage into the SPR Merkle tree. If a collision is found at the right position, this yields the desired second preimage.

We now explain the details of the reduction, i.e. how an existential forger for SPR-MSS can be used to construct an adversary that is either an existential forger for the underlying one-time signature scheme or a second preimage finder for a hash function in $\mathcal{G}$. The adversary takes as input a key $k \in K$ and a first preimage $x \in_R \{0,1\}^{2n}$ both chosen randomly with the uniform distribution and the SPR-Merkle tree height H. The adversary also takes as input a one-time signature scheme, a verification key Y_{OTS} and a signing oracle $\mathcal{O}_{\mathrm{OTS}}(X_{\mathsf{OTS}}, \cdot)$, where $(X_{\mathsf{OTS}}, Y_{\mathsf{OTS}})$ is a key pair of the one-time signature scheme.

The adversary is allowed to query the oracle $\mathcal{O}_{\mathrm{OTS}}(X_{\mathsf{OTS}}, \cdot)$ once. His goal is to output a second preimage of x under g_k or an existential forgery (M', σ') for the one-time signature scheme that can be verified using the verification key Y_{OTS}. He has access to an adaptive chosen message forger $\textsc{For}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$ for the SPR-MSS with hash function g_k and tree height H. The forger is allowed to ask 2^H queries to its signature oracle. The adversary is supposed to impersonate that oracle.

The adversary selects randomly with the uniform distribution an index c in the set $\{0, \ldots, 2^H - 1\}$. He generates one-time verification keys $(X_j, Y_j), j = 0, \ldots, 2^H - 1, j \neq c$ in the usual manner with the exception that as cth one-time verification key the one-time verification key Y_{OTS} from the input is used. The adversary now executes the necessary steps to include the first preimage x in the SPR-Merkle

tree. He chooses randomly with the uniform distribution a tuple (a, b) in the set $\big\{(h, j) : h \in \{1, \ldots, H + l\}, j \in \{0, \ldots, 2^{H+l-h} - 1\}\big\}$. The adversary then chooses masks $v_h[0], v_h[1] \in_R \{0,1\}^n, h = 1, \ldots, H + l,\ h \neq a$ randomly with the uniform distribution. The adversary then constructs the SPR-Merkle tree up to height $a - 1$ such that he knows the nodes $\nu_{a-1}[2b], \nu_{a-1}[2b + 1]$. He then computes the masks $v_a[0], v_a[1]$ as

$$v_a[0] \parallel v_a[1] = x \oplus \big(\nu_{a-1}[2b] \parallel \nu_{a-1}[2b + 1]\big). \tag{6.31}$$

Then $x = \big(\nu_{a-1}[2b] \oplus v_a[0]\big) \parallel \big(\nu_{a-1}[2b+1] \oplus v_a[1]\big)$ and therefore $y_a[b] = g_k(x)$ holds. Finally the adversary uses $v_a[0], v_a[1]$ to complete the key pair generation. Note that the masks $v_a[0], v_a[1]$ are indistinguishable from bit strings chosen randomly with the uniform distribution because x was chosen chosen randomly with the uniform distribution. As a consequence, the adversary $\textsc{Adv}_{\mathsf{SPR},\mathsf{OTS}}$ creates an environment identical to the signature forging game played by the forger.

The next steps are the same as in the security reduction for the original MSS explained in Section 6.1.2 The adversary invokes the adaptive chosen message forger for SPR-MSS with hash function g_k and the public SPR-MSS key which he generated before. Without loss of generality, we assume that the forger queries the oracle 2^H times. The oracle answers are given by the adversary. When the forger asks for the ith signature, $i \neq c$, then the adversary produces this signatures using the signature keys which he generated before. However, when the forger asks for the cth signature, the adversary queries the oracle $\mathcal{O}_{\mathrm{OTS}}(X_{\mathsf{OTS}}, \cdot)$. Suppose that the forger is successful and outputs an existential forgery $(M', (s, \sigma', Y', A'))$ where s is the index of the one-time key pair used for this signature, σ' is the one-time signature, Y' is the verification key and A' is the authentication path. The adversary examines the SPR-MSS signature (s, σ, Y, A) of M he returned in response to the forgers sth oracle query.

If $s = c$ and $(Y, A) = (Y', A')$, then the adversary returns (M', σ') as existential forgery of the one-time signature scheme with verification key Y_{OTS} as described in Section 6.1.2.

If $(Y, A) \neq (Y', A')$ the adversary can find a collision as described in Section 6.1.2. This collision can occur either during the computation of the inner node $\nu_l[s]$ from the leaves $\nu_0[s2^l], \ldots, \nu_0[s2^l + 2^l - 1]$ or during the computation of the path from $\nu_l[s]$ to the root $\nu_{H+l}[0]$. If the collision occurs at node $\nu_a[b]$, i.e. $\nu_a[b] = g_k(x')$ with $x' \neq x$, the adversary outputs x' as second preimage of x under g_k.

In all other cases, the adversary returns `failure`. Algorithm 6.3 summarizes our description.

We now estimate the success probability of the adversary $\textsc{Adv}_{\mathsf{SPR},\mathsf{OTS}}$. As before, ϵ denotes the success probability and t the running time of the forger. Also, $t_{\textsc{Gen}}, t_{\textsc{Sig}}$, and $t_{\textsc{Ver}}$ denote the times SPR-MSS requires for key generation, signature generation, and verification, respectively.

If $(Y', A') \neq (Y, A)$, then the adversary finds a collision. Since the position of node $\nu_a[b]$ was chosen at randomly with the uniform distribution, the probability that the collision occurs precisely at this position is at least $1/(2^{H+l} - 1)$. The

Algorithm 6.3 $\text{ADV}_{\mathsf{SPR,OTS}}$

Input: Key for the hash function $k \xleftarrow{\$} K$, height of the tree $H \geq 2$, first-preimage $x \in_R \{0,1\}^{2n}$, one instance of the underlying OTS consisting of a verification key Y_{OTS} and the corresponding signing oracle $\mathcal{O}_{\text{OTS}}(X_{\mathsf{OTS}}, \cdot)$.
Output: Second-preimage $x' \in \{0,1\}^{2n}$ with $x' \neq x$ and $g_k(x) = g_k(x')$, an existential forgery for the supplied instance of the OTS, or `failure`

1. Set $c \xleftarrow{\$} \{0, \ldots, 2^h - 1\}$.
2. Generate OTS key pairs $(X_j, Y_j), j = 0, \ldots, 2^H - 1, j \neq c$ and set $Y_c \leftarrow Y_{\mathsf{OTS}}$.
3. Set $(a, b) \xleftarrow{\$} \big\{(h, j) : h \in \{1, \ldots, H + l\}, j \in \{0, \ldots, 2^{H+l-h} - 1\}\big\}$.
4. Choose random masks $v_h[0], v_h[1] \xleftarrow{\$} \{0,1\}^n, h = 1, \ldots, H + l,\ h \neq a$.
5. Construct the SPR-Merkle tree up to height $a - 1$.
6. Compute $(v_a[0] \parallel v_a[1]) \leftarrow x \oplus \big(\nu_{a-1}[2b] \parallel \nu_{a-1}[2b + 1]\big)$.
7. Use $v_a[0], v_a[1]$ to complete the key pair generation.
8. Run $\text{FOR}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$.
9. When $\text{FOR}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$ asks its qth oracle query ($0 \leq q \leq 2^H - 1$):
   a) **if** $q = c$ **then** query the signing oracle $\mathcal{O}(\mathsf{sk}, \cdot)$.
   b) **else** compute the one-time signature σ using the qth signature key X_q.
   c) Return the corresponding SPR-Merkle signature to the forger.
10. If the forger outputs an existential forgery $(M', (s, \sigma', Y', A'))$, examine the Merkle signature (s, σ, Y, A) returned in response to the forgers sth oracle query.
    a) **if** $(Y', A') \neq (Y, A)$:
       i. **if** $\nu_a[b]$ is computed during the verification as $\nu_a[b] = g_k(x')$ and $x' \neq x$ holds **then return** x' as second-preimage of x.
       ii. **else return** `failure`.
    b) **else**
       i. **if** $s = c$ **then return** (M', σ') as forgery for the supplied instance of the one-time signature scheme.
       ii. **else return** `failure`.

adversary's (conditional) probability ϵ_{SPR} for returning a second preimage in a time $t_{\text{SPR}} = t + 2^H \cdot t_{\text{SIG}} + t_{\text{VER}} + t_{\text{GEN}}$ is at least $\epsilon/(2^{H+l} - 1)$. If $(Y', A') = (Y, A)$ the adversary returns an existential forgery if $s = c$. His (conditional) probability ϵ_{OTS} for finding an existential forgery in a time $t_{\text{OTS}} = t + 2^H \cdot t_{\text{SIG}} + t_{\text{VER}} + t_{\text{GEN}}$ is at least $\epsilon \cdot 1/2^H$. Since both cases are mutually exclusive, one of them occurs with probability at least $1/2$. So we have proved the following theorem.

Theorem 6.6. *Let K be a finite set, let $H \in \mathbb{N}$, $t_{\text{SPR}}, t_{\text{OTS}}, \epsilon_{\text{SPR}}, \epsilon_{\text{OTS}} \in \mathbb{R}_{>0}$, $\epsilon_{\text{SPR}} \leq 1/(2^{H+l+1} - 2)$, $\epsilon_{\text{OTS}} \leq 1/2^{H+1}$, and let $\mathcal{G} = \{g_k : \{0,1\}^* \to \{0,1\}^n | k \in K\}$ be a family of $(t_{\text{SPR}}, \epsilon_{\text{SPR}})$ second preimage resistant hash functions. Consider SPR-MSS using a $(t_{\text{OTS}}, \epsilon_{\text{OTS}}, 1)$ one-time signature scheme. Then SPR-MSS is a $(t, \epsilon, 2^H)$ signature scheme with*

$$\begin{aligned} \epsilon &\leq 2 \cdot \max\left\{(2^{H+l} - 1) \cdot \epsilon_{\text{SPR}}, 2^H \cdot \epsilon_{\text{OTS}}\right\} && (6.32) \\ t &= \min\left\{t_{\text{SPR}}, t_{\text{OTS}}\right\} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}. && (6.33) \end{aligned}$$

6.2.3 Security level of SPR-MSS

In this section we compute the security level of SPR-MSS when used with the Lamport–Diffie one-time signature scheme (LD-OTS). The computation is similar to what was done in Section 6.1.3. We express security level as t/ϵ where t is the running time of an existential forger and ϵ is its success probability. In this section let $\epsilon_{\text{SPR}}, t_{\text{SPR}}, \epsilon_{\text{OW}}, t_{\text{OW}} \in \mathbb{R}_{>0}$, let K be a finite set, and let

$$\mathcal{G} = \{g_k : \{0,1\}^* \to \{0,1\}^n | k \in K\} \tag{6.34}$$

be a family of $(t_{\text{SPR}}, \epsilon_{\text{SPR}})$ second preimage resistant and $(t_{\text{OW}}, \epsilon_{\text{OW}})$ preimage resistant hash functions.

Since we consider SPR-MSS using LD-OTS, we first combine Theorem 6.1 and Theorem 6.6. This is achieved by substituting the values for ϵ_{OTS} and t_{OTS} from Theorem 6.1 in Equations (6.32) and (6.33) from Theorem 6.6. An LD-OTS verification key consists of $2n = 2^{\log_2 2n}$ bit strings of length n (see Section 2.1.1). We therefore have $l = \log_2 2n$. In summary we get

$$\begin{aligned} \epsilon &\leq 2 \cdot \max\left\{(2^{H+\log_2 2n} - 1) \cdot \epsilon_{\text{SPR}}, 2^{H+\log_2 4n} \cdot \epsilon_{\text{OW}}\right\} && (6.35) \\ t &= \min\left\{t_{\text{SPR}}, t_{\text{OW}}\right\} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}. && (6.36) \end{aligned}$$

Again we replace t_{OTS} by t_{OW} rather than $t_{\text{OW}} - t_{\text{SIG}} - t_{\text{GEN}}$, since the time LD-OTS requires for signature and key generation is already included in the signature and key generation time of SPR-MSS in Theorem 6.6. To ensure $\epsilon \leq 1$, we require $\epsilon_{\text{SPR}} \leq 1/(2^{H+\log_2 2n+1} - 2)$, $\epsilon_{\text{OW}} \leq 1/(2^{H+\log_2 4n})$.

To estimate the security level, we need explicit values for the key pair generation, signature generation and verification times of SPR-MSS using LD-OTS. We use the upper bounds from Equation (6.10).

$$t_{\text{GEN}} \leq 2^H \cdot 6n, \quad t_{\text{SIG}} \leq 4n(H+1), \quad t_{\text{VER}} \leq n + H$$

Again we make assumptions for the values of $(t_{\text{SPR}}, \epsilon_{\text{SPR}})$ and $(t_{\text{OW}}, \epsilon_{\text{OW}})$ and distinguish between attacks that use classical computers only and attacks with quantum computers.

Using classical computers. In our security analysis of SPR-MSS we assume that the hash functions under consideration have output length n and only admit generic attacks against their preimage and second preimage resistance. The best generic attack to find preimages or second preimages is exhaustive search. An exhaustive search of $2^{n-H-\log_2 4n-1}$ random strings yields a preimage of a given hash value with probability $1/2^{H+\log_2 4n+1}$. We therefore assume that our hash functions are

$$(t_{\text{OW}}, \epsilon_{\text{OW}}) = (2^{n-H-\log_2 4n-1}, 1/2^{H+\log_2 4n+1}) \tag{6.37}$$

preimage resistant. Also, an exhaustive search of $2^{n-H-\log_2 4n-1}$ random strings yields a second preimage of a given first preimage with probability $1/2^{H+\log_2 4n+1}$. We therefore assume that our hash functions are

$$(t_{\text{SPR}}, \epsilon_{\text{SPR}}) = (2^{n-H-\log_2 4n-1}, 1/2^{H+\log_2 4n+1}) \tag{6.38}$$

second preimage resistant. In this situation, we prove the following theorem.

Theorem 6.7 (Classical case)**.** *The security level of SPR-MSS combined with the Lamport–Diffie one-time signature scheme is at least*

$$b = 2/3 \cdot n - \log_2 n - 4 \tag{6.39}$$

if the height of the Merkle tree is at most $H \leq n/3$ and the output length of the hash function is at least $n \geq 69$.

To prove Theorem 6.7 we use our assumption and Equations (6.35) and (6.36) and obtain the following estimate for the security level.

$$\frac{t}{\epsilon} \geq \frac{2^{n-H-\log_2 4n-1} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}}{2 \cdot \max\left\{\frac{2^{H+\log_2 2n}-1}{2^{H+\log_2 4n+1}}, \frac{2^{H+\log_2 4n}}{2^{H+\log_2 4n+1}}\right\}}. \tag{6.40}$$

The maximum in the denominator is $1/2$ for any choice of n and H. Using the upper bounds for t_{SIG}, t_{VER}, and t_{GEN} estimated above, Equation (6.40) implies

$$\frac{t}{\epsilon} \geq 2^{n-H-\log_2 4n-1} - 2^H \cdot 4n(H+1) - (n+H) - 2^H \cdot 6n. \tag{6.41}$$

Using $H \leq n/3$, the desired lower bound for the security level of $2/3 \cdot n - \log_2 n - 4$ holds as long as

$$2^{n/3}(4/3 \cdot n^2 + 4n) + 4/3 \cdot n + 2^{n/3} \cdot 6n \leq 2^{2n/3-\log_2 n-4} \tag{6.42}$$

which is true for $n \geq 69$.

Using quantum computers. Again, we assume that our hash functions have output length n and only admit generic attacks against their preimage and second preimage resistance. Again, we use the Grover algorithm [21] in those generic attacks. By virtue of Grover's algorithm we may assume that our hash functions are

$$(t_{\mathrm{OW}}, \epsilon_{\mathrm{OW}}) = (2^{n/2-(H+\log_2 4n+1)/2}, 1/2^{H+\log 4n+1}) \tag{6.43}$$

preimage resistant and

$$(t_{\mathrm{SPR}}, \epsilon_{\mathrm{SPR}}) = (2^{n/2-(H+\log_2 4n+1)/2}, 1/2^{H+\log 4n+1}) \tag{6.44}$$

second preimage resistant, see [6] Chapter 2 "Quantum computing". In this situation, we prove the following theorem.

Theorem 6.8 (Quantum case)**.** *The security level of SPR-MSS combined with the Lamport-Diffie one-time signature scheme is at least*

$$b = 3/8 \cdot n - \log_4 n - 2 \tag{6.45}$$

if the height of the Merkle tree is at most $H \leq n/4$ and the output length of the hash function is at least $n \geq 176$.

To prove Theorem 6.8 we use the same approach as for the proof of Theorem 6.7. We use our assumption on the hash function and Equations (6.35) and (6.36) and obtain the following estimate for the security level.

$$\frac{t}{\epsilon} \geq \frac{2^{n/2-(H+\log_2 4n+1)/2} - 2^H \cdot t_{\mathrm{SIG}} - t_{\mathrm{VER}} - t_{\mathrm{GEN}}}{2 \cdot \max\left\{\frac{2^{H+\log_2 2n}-1}{2^{H+\log_2 4n+1}}, \frac{2^{H+\log_2 4n}}{2^{H+\log_2 4n+1}}\right\}}. \tag{6.46}$$

Again, the maximum in the denominator is $1/2$ for any choice of n and H. Using the upper bounds for t_{SIG}, t_{VER}, and t_{GEN} estimated above, Equation (6.46) implies

$$\frac{t}{\epsilon} \geq 2^{n/2-(H+\log_2 4n+1)/2} - 2^H \cdot 4n(H+1) - (n+H) - 2^H \cdot 6n. \tag{6.47}$$

Using $H \leq n/4$, the desired lower bound for the security level of $3/8 \cdot n - \log_4 n - 2$ holds as long as

$$2^{n/4}(n^2+4n) + 5/4 \cdot n + 2^{n/4} \cdot 6n \leq 2^{3n/8-\log_4 n-3/2} - 2^{3n/8-\log_4 n-2} \tag{6.48}$$

which is true for $n \geq 176$.

6.2.4 Signing arbitrarily long messages

In the security reduction of the Lamport–Diffie one-time signature scheme the length of the message to be signed is limited to n bits (see Section 6.1.1). In real-world applications this is not sufficient. In order to sign arbitrarily long messages, the message is hashed to a n-bit string prior to signing and the hash value is signed (see

Sections 2.1 and 2.2). To prevent trivial existential forgeries, the used hash function must be collision resistant. In case of the original Merkle signature scheme this is no additional restriction, because collision resistance is required for the security reduction of the tree authentication scheme anyway (see Section 6.1.2). In other words, Theorems 6.3 and 6.4 remain valid if the message to be signed is hashed using a collision resistant hash function and the hash value is signed.

This does not hold in case of SPR-MSS which is specifically designed to eliminate the necessity of collision resistance. So using a collision resistant hash function for the initial hash of the message would decrease the security level estimated in Theorems 6.7 and 6.8. In the following, we review a security notion that is weaker than collision resistance but strong enough to prevent trivial existential forgeries. Let

$$\mathcal{G} = \left\{ g_k : \{0,1\}^* \to \{0,1\}^n | k \in K \right\} \tag{6.49}$$

be a family of hash functions and m a positive integer.

We define *universal one-way hash functions* (UOWHF), also known as *target collision resistance* (TCR) hash functions [32, 5]. We use the definition from [35] where this property is called *everywhere second preimage resistant* (eSec). TCR is defined using a two stage adversary. In the beginning the adversary Adv commits to a first preimage $x \in \{0,1\}^m$. In the next step a key $k \in K$ is chosen randomly with the uniform distribution and passed to the adversary that continues where it left off. The adversary outputs a second preimage x' of x or `failure`. The success probability of this adversary is denoted by

$$\Pr[x \xleftarrow{\$} \textsc{Adv}(), k \xleftarrow{\$} K, x' \xleftarrow{\$} \textsc{Adv}(k) : x \neq x' \wedge g_k(x) = g_k(x')]. \tag{6.50}$$

The TCR property is stronger that second preimage resistance because the adversary gets to choose the first preimage. However, TCR is weaker than collision resistance, because the adversary has to choose the first preimage before knowing under which element of the family $\mathcal{G}$ it is supposed to find a second preimage. For that reason, the birthday paradox cannot be used in generic attacks to break the TCR property [32]. We finally remark that collision resistance implies TCR which in turn implies second preimage resistance [35].

When using a TCR hash function with output length n for the initial hashing of the message, the key k of this function must be signed as well. In [5], Bellare and Rogaway show how a TCR hash function can be built from a $d-$to-1 TCR compression function using the XOR construction we used for the SPR-Merkle tree. In the following we use $d = 2$. Using this construction, the length of the key k depends on the message length. If the message to be signed consists of b blocks of length n, i.e. has bit length $b \cdot n$, then the key k consists of is $2 \cdot \lceil \log_2 b \rceil + 1$ blocks of length n. We need one n bit key for the compression function and $2 \cdot \lceil \log_2 b \rceil$ random masks of bit length n for the XOR construction. Together with the n bit digest of the message, $2n \cdot (\lceil \log_2 b \rceil + 1)$ bits must be signed. Even though the key size depends only logarithmically on the message length it can be quite large. This affects the size of the one-time signature as shown in the following example.

Example 6.9. *Suppose we use a TCR hash function with output length* $n = 128$ *and want to sign a message of size 16 MByte, i.e. a message that consists of* 2^{20} *blocks of length 128 bit. The size of the key then is* $128 + 2 \cdot 128 \cdot 20 = 5248$ *bits and together with the* 128 *bit digest of the message, a total of 5376 bits must be signed. When using the Lamport-Diffie one-time signature scheme, the size of the one-time signature is 86,016 Bytes.*

In order to solve the problem of long keys, Bellare and Rogaway suggested iterating TCR hash functions [5]. For example, the TCR hash function can be iterated with three different keys k_1, k_2 and k_3 as explained in the following. Assume that a message of length $b \cdot n$ bits is to be signed. As described above, the key k_1 for the first iteration consists of $|k_1| = 2 \cdot \lceil \log_2 b \rceil + 1$ blocks of length n. This key is hashed in the second iteration together with the n bit output of the first iteration. So the key k_2 for the second iteration consists of

$$|k_2| = 2 \cdot \lceil \log_2(|k_1| + 1) \rceil + 1$$

blocks of length n. This key and the output of the second iteration are hashed in the third iteration and therefore the key k_3 consists of

$$|k_3| = 2 \cdot \lceil \log_2(|k_2| + 1) \rceil + 1$$

blocks of length n. Although the three keys must be transmitted with the signature, only the third key k_3 must be signed. So the size of the data to be signed is $(|k_3|+1) \cdot n$ bits. This process is depicted in Figure 6.3.

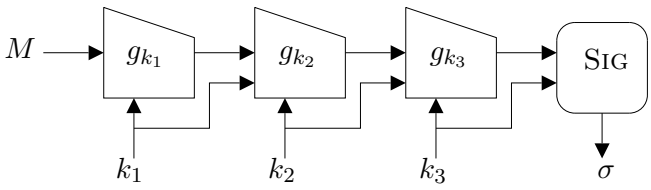

Figure 6.3: Iterating TCR hash functions

Iterating the TCR hash function drastically reduces the size of the one-time signature as shown in the following example.

Example 6.10. *Suppose we use a TCR hash function with output length* $n = 128$ *and want to sign a 16 MByte message. We iterate the hash function three times. The sizes of the keys* k_1, k_2, k_3 *are*

$$\begin{aligned} |k_1| &= 41 \cdot 128 = 5248 \text{ bits.} \\ |k_2| &= 13 \cdot 128 = 1664 \text{ bits.} \\ |k_3| &= 9 \cdot 128 = 1152 \text{ bits.} \end{aligned}$$

The total size of the data to be signed is 1280 bits and when using the Lamport–Diffie one-time signature scheme the size of the one-time signature is $20,480$ *Bytes.*

6.3 Comparison of the security level

To conclude this chapter, we present explicit values for the security level of MSS and SPR-MSS combined with the Lamport–Diffie one-time signature scheme for fixed output lenghts of the hash function. These values are obtained using Theorems 6.3, 6.4, 6.7, and 6.8 and are summarized in Table 6.1. This table also shows the maximum value for the height H of the tree such that the security level holds.

Table 6.1: Security level of MSS and SPR-MSS combined with LD-OTS in bits.

Output length n	128	160	224	256	384	512
Classical case						
MSS bit security b	63	79	111	127	191	255
SPR-MSS bit security b	74	95	137	158	243	328
Maximum value for H	42	53	74	85	128	170
Quantum case						
MSS bit security b	–	–	73	84	127	169
SPR-MSS bit security b	–	–	78	90	137	185
Maximum value for H	–	–	56	64	96	128

This table shows that the security reduction for the original MSS is very tight. Only one bit of security is lost compared to the security level of the underlying hash function against generic collision attacks. Unfortunately this is not true for SPR-MSS. Although its security level is significantly higher than that of MSS, it does not inherit the security level of the underlying hash function against generic preimage and second preimage attacks. The reason for that is the low success probability of the adversary described in Section 6.2.2, which in turn is the result of the low probability that the forger produces a collision in the exact same tree node where the first preimage was inserted. However, in practice second preimage resistance is easier to achieve than collision resistance which is a great advantage of SPR-MSS.

To summarize, state of the art hash functions can be used to ensure a high security level of the Merkle signature scheme, even against attacks by quantum computers. For all practical applications the maximum height of the Merkle tree and the resulting number of messages that can be signed with one key pair is sufficiently large.

7 Conclusion

This thesis introduces two enhancements for the Merkle signature scheme drastically reducing the worst case signature generation time. The signature scheme using these enhancements is called GMSS. The worst case signature generation time of GMSS corresponds to the average case signature generation time of the Merkle signature scheme combined with Szydlo's and Coronado's improvements. Further, the worst case signature generation time of GMSS is extremely close to its average case signature generation time. Hence, GMSS is the first Merkle type signature scheme providing truly balanced timings. The improved signature generation times of GMSS are exploited to provide a noticeable reduction of the signature sizes, while maintaining timings highly competitive to RSA and ECDSA. This is illustrated by explicit timings of two implementations, one in Java and one for 8-bit AVR microcontrollers.

This thesis also introduces SPR-MSS, a signature scheme that in contrast to the original MSS requires only second preimage resistant hash functions in order to be existentially unforgeable under adaptive chosen message attacks. Reducing the requirements on the hash function significantly increases the security level.

In summary, there is now a signature scheme at our disposal not only efficient enough to take over the place of RSA and ECDSA but also secure as long as cryptographically secure hash functions exist. Contrary to RSA and ECDSA, large scale quantum computers and innovative ideas leading to polynomial time algorithms for solving number theoretic problems pose no threat to GMSS.

Further research. The main problem of the Merkle signature scheme still is the signature size which is dominated by the size of the one-time signature. Although the signature size is tolerable in most applications, it is desirable to have signature sizes comparable to RSA. Therefore it is an important research goal to explore one-time signature schemes providing small signature sizes and being resistant against quantum computer attacks. The lattice based one-time signature scheme proposed by Lyubashevsky and Micciancio [28] presents a promising example.

Further research also includes more efficient implementations. GMSS is highly parallelizable. This can be exploited when a multi-core CPU is available, which is the case in state of the art PCs. For example, generating leaves essentially requires t bit strings of length n to be hashed 2^w times. So the computation of a leaf can be done by t independent processes running in parallel. On dual-core systems this would halve the time required for key pair generation, signature generation, and verification. This speed-up can again be used to decrease the signature size.

References

[1] Aaronson, S., Shi, Y.:. Quantum lower bounds for the collision and the element distinctness. Journal of the ACM, 51(4):595–605, 2004.

[2] Atmel: Overview of secure avr microcontrollers 8-/16-bit risc cpu, 2007. Available at `http://www.atmel.com/products/SecureAVR/`.

[3] Atmel: Specifications of the atmega128 microcontroller, 2007. Available at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`.

[4] Bellare, M., Rogaway, P.: Optimal asymmetric encryption. Advances in Cryptology - EUROCRYPT'94, LNCS 950, pages 92–111, Springer, 1995.

[5] Bellare, M., Rogaway, P.: Collision-resistant hashing: Towards making UOWHFs practical. Advances in Cryptology - CRYPTO '97, LNCS 1294, pages 470–484, Springer, 1997.

[6] Bernstein, D. J., Buchmann, J., Dahmen, E. (Eds.): Post-Quantum Cryptography. Springer, 2009, ISBN: 978-3-540-88701-0.

[7] Blume, S.: Efficient Java Implementation of GMSS diploma thesis, Technische Universität Darmstadt, 2007. Available at `http://www.cdc.informatik.tu-darmstadt.de`

[8] Berman, P., Karpinski, M., Nekrich, Y.: Optimal Trade-Off for Merkle Tree Traversal. Theoretical Computer Science, volume 372, issue 1, pages 26–36, 2007.

[9] Boneh, D., Mironov, I., Shoup, V.: A secure signature scheme from bilinear maps. In *Topics in Cryptology - CT-RSA 2003*, LNCS 2612, pages 98–110. Springer, 2003.

[10] Buchmann, J., Coronado, C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – an improved Merkle signature scheme. Progress in Cryptology - INDOCRYPT 2006, LNCS 4329, pages 349–363, Springer, 2006.

[11] Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle signatures with virtually unlimited signature capacity. Applied Cryptography and Network Security - ACNS 2007, LNCS 4521, pages 31–45, Springer, 2007.

[12] Buchmann, J., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. Post-Quantum Cryptography - PQCrypto 2008, LNCS 5299, pages 63–77, Springer, 2008.

[13] Coronado, C.: On the security and the efficiency of the Merkle signature scheme. Cryptology ePrint Archive, Report 2005/192, 2005. Available at `http://eprint.iacr.org/`.

[14] Coronado, C.: Provably Secure and Practical Signature Schemes. PhD thesis, Technische Universität Darmstadt, 2005. Available at `http://elib.tu-darmstadt.de/diss/000642/`.

[15] Driessen, B., Poschmann, A., Paar, C.: Comparison of Innovative Signature Algorithms for WSNs. Proceedings of the First ACM Conference on Wireless Network Security - WISEC 2008, pages 30–35, 2008.

[16] Dods, C., Smart, N., Stam, M.: Hash based digital signature schemes. Cryptography and Coding, LNCS 3796, pages 96–115, Springer, 2005.

[17] Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital Signatures out of Second-Preimage Resistant Hash Functions, 2nd International Workshop on Post-Quantum Cryptography - PQCrypto 2008, LNCS 5299, pages 109-123, Springer, 2008.

[18] European Network of Excellence in Cryptology - ECRYPT: D.SPA.28 - Yearly Report on Algorithms and Keysizes, 2007. Available at `http://www.ecrypt.eu.org/documents.html`

[19] National Institute of Standards and Technology: Digital signature standard (DSS), FIPS PUB 186-3, 2006. Available at `http://csrc.nist.gov/publications/fips/`.

[20] The FlexiProvider group at Technische Universität Darmstadt: FlexiProvider, an open source Java Cryptographic Service Provider, 2001–2008. Available at `http://www.flexiprovider.de/`.

[21] Grover, L. K.: A fast quantum mechanical algorithm for database search. Proceedings of the Twenty-Eighth Annual Symposium on the Theory of Computing, pages 212–219, ACM Press New York, 1996.

[22] Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S. C.: Comparing elliptic curve cryptography and rsa on 8-bit cpus. Cryptographic Hardware and Embedded Systems - CHES 2004, LNCS 3156, pp 119–132. Springer, 2004.

[23] Jakobsson, M., Leighton, T., Micali, S., Szydlo, M.: Fractal Merkle tree representation and traversal. Topics in Cryptology CT-RSA 2003, LNCS 2612, pages 314–326, Springer, 2003.

[24] Johnson, D. and Menezes, A.: The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical Report CORR 99-34, University of Waterloo, 1999. Available at `http://www.cacr.math.uwaterloo.ca`.

[25] Lamport, L.: Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.

[26] Lenstra, A. K., Verheul, E. R.:. Selecting cryptographic key sizes. Journal of Cryptology, 14(4):255–293, 2001. Updated version from 2004 available at `http://plan9.bell-labs.com/who/akl/index.html`.

[27] Liu, A., Ning, P.: TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. Technical Report TR-2007-36, North Carolina State University, Department of Computer Science, November 2007.

[28] Lyubashevsky, V., Micciancio, D.: Asymptotically Efficient Lattice-Based Digital Signatures. Theory of Cryptography - TCC 2008, LNCS 4948, pages 37–54, Springer, 2008.

[29] Menezes, A. J., Vanstone, S. A., van Oorschot, P. C.: Handbook of Applied Cryptography. CRC Press, 1996.

[30] Merkle, R.C.: A certified digital signature. Advances in Cryptology – CRYPTO '89 Proceedings, LNCS 435, pages 218–238, Springer, 1990.

[31] Naor, D., Shenhav, A., Wool, A.: One-time signatures revisited: Practical fast signatures using fractal merkle tree traversal. IEEE – 24th Convention of Electrical and Electronics Engineers in Israel, pages 255–259, 2006.

[32] Naor, M., Yung, M.: Universal one-way hash functions and their cryptographic applications. 21st Annual ACM Symposium on Theory of Computing - STOC'89, pages 33–43. ACM Press, 1989.

[33] Rivest, R. L., Shamir, A., and Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, 21(2):120–126, 1978.

[34] Rohde, S., Eisenbarth, T., Dahmen, E., Buchmann, J., Paar, C.: Fast Hash-Based Signatures on Constrained Devices, Eighth Smart Card Research and Advanced Application Conference - CARDIS 2008, LNCS 5189, pages 104-117, Springer, 2008.

[35] Rogaway, P., Shrimpton, T.: Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. Fast Software Encryption - FSE 2004, LNCS 3017, pages 371–388, Springer, 2004.

[36] The Standards for Efficient Cryptography Group: SEC 2: Recommended Elliptic Curve Domain Parameters, 2000. Available at `http://www.secg.org`

[37] Schneider, M.: Improved Authentication Path Computation for Merkle Trees, diploma thesis, Technische Universität Darmstadt, 2008. Available at `http://www.cdc.informatik.tu-darmstadt.de`

[38] Shor, P. W.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science - FOCS 1994, pages 124–134, IEEE Computer Society Press, 1994.

[39] Steinberger, J. P.: The collision intractability of mdc-2 in the ideal-cipher model. Advances in Cryptology - EUROCRYPT 2007, LNCS 4515, pages 34–51, Springer, 2007.

[40] Sun Microsystems: The Java Cryptography Architecture API Specification & Reference, 2002. Available at `http://java.sun.com`.

[41] Sun Microsystems: The Java Cryptography Extension (JCE) Reference Guide. Available at `http://java.sun.com`

[42] Szydlo, M.: Merkle Tree Traversal in Log Space and Time. Advances in Cryptology - EUROCRYPT 2004, LNCS 3027, pages 541–554, Springer, 2004

[43] Szydlo, M.: Merkle Tree Traversal in Log Space and Time. Preprint, 2003. Available at `http://www.szydlo.com`

[44] Viega, J.: The AHASH Mode of Operation. Manuscript, 2004. Available at `http://www.cryptobarn.com/`

[45] Wang, X., Yu, H.: How to break MD5 and other hash functions. Advances in Cryptology - EUROCRYPT 2005, LNCS 3494, pages 19–35, Springer, 2005.

[46] Wang, X., Yin, Y.-L., Yu, H.; Finding collisions in the full SHA-1. Advances in Cryptology - CRYPTO 2005, LNCS 3621, pages 17–36, Springer, 2005.