# CMSS – An Improved Merkle Signature Scheme

Johannes Buchmann[1], Luis Carlos Coronado García[2], Erik Dahmen[1], Martin Döring[1⋆], and Elena Klintsevich[1]

[1] Technische Universität Darmstadt
Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
{buchmann,dahmen,doering,klintsev}@cdc.informatik.tu-darmstadt.de

[2] Banco de México
Av. 5 de Mayo No. 6, 5to piso
Col. Centro C.P. 06059, México, D.F.
coronado@banxico.org.mx

**Abstract.** The Merkle signature scheme (MSS) is an interesting alternative for well established signature schemes such as RSA, DSA, and ECDSA. The security of MSS only relies on the existence of cryptographically secure hash functions. MSS has a good chance of being quantum computer resistant. In this paper, we propose CMSS, a variant of MSS, with reduced private key size, key pair generation time, and signature generation time. We demonstrate that CMSS is competitive in practice by presenting a highly efficient implementation within the Java Cryptographic Service Provider FlexiProvider. We present extensive experimental results and show that our implementation can for example be used to sign messages in Microsoft Outlook.

**Keywords:** Java Cryptography Architecture, Merkle Signatures, One-Time-Signatures, Post-Quantum Signatures, Tree Authentication.

## 1 Introduction

Digital signatures have become a key technology for making the Internet and other IT infrastructures secure. Digital signatures provide authenticity, integrity, and support for non-repudiation of data. Digital signatures are widely used in identification and authentication protocols, for example for software downloads. Therefore, secure digital signature algorithms are crucial for maintaining IT security.

Commonly used digital signature schemes are RSA [RSA78], DSA [Elg85], and ECDSA [JM99]. The security of those schemes relies on the difficulty of factoring large composite integers and computing discrete logarithms. However, it is unclear whether those computational problems remain intractable in the future. For example, Peter Shor [Sho94] proved that quantum computers can

⋆ Author supported by SicAri, a project funded by the German Ministry for Education and Research (BMBF). See http://www.sicari.de.

factor integers and can calculate discrete logarithms in the relevant groups in polynomial time. Also, in the past thirty years there has been significant progress in solving the integer factorization and discrete logarithm problem using classical computers (Lenstra and Verheul). It is therefore necessary to come up with new signature schemes which do not rely on the difficulty of factoring and computing discrete logarithms and which are even secure against quantum computer attacks. Such signature schemes are called post-quantum signature schemes.

A very interesting post-quantum signature candidate is the Merkle signature scheme (MSS) [Mer89]. Its security is based on the existence of cryptographic hash functions. In contrast to other popular signature schemes, MSS can only verify a bounded number of signatures using one public key. Also, MSS has efficiency problems (key pair generation, large secret keys and signatures) and was not used much in practice.

**Our contribution** In this paper, we present CMSS, a variant of MSS, with reduced private key size, key pair generation time, and signature generation time. We show that CMSS is competitive in practice by presenting a highly efficient CMSS Java implementation in the Java Cryptographic Service Provider FlexiProvider [Flexi]. This implementation permits easy integration into applications that use the Java Cryptography Architecture [JCA02]. We present experiments that show: As long as no more that $2^{40}$ documents are signed, the CMSS key pair generation time is reasonable, and signature generation and verification times in CMSS are competitive or even superior compared to RSA and ECDSA. We also show that the CMSS implementation can be used to sign messages in Microsoft Outlook using our `FlexiS/MIME` plug-in [FOP03]. The paper specifies CMSS keys using Abstract Syntax Notation One (ASN.1) [Int02] which guarantees interoperability and permits efficient generation of X.509 certificates and PKCS#12 personal information exchange files. CMSS is based on the Thesis of Coronado [Cor05b] and incorporates the improvements of MSS from [Szy04,DSS05].

**Related Work** Szydlo presents a method for the construction of authentication paths requiring logarithmic space and time in [Szy04]. Dods, Smart and Stam give the first complete treatment of practical implementations of hash based digital signature schemes in [DSS05]. In [NSW05], Naor et. al. propose a C implementation of MSS and give timings for up to $2^{20}$ signatures. A preliminary version of CMSS including security proofs appeared in the PhD thesis of Coronado [Cor05b] and in [Cor05a].

**Organization** The rest of this paper is organized as follows: In Section 2, we describe the Winternitz one-time signature scheme and the Merkle signature scheme. In Section 3, we describe CMSS. Section 4 describes details of the CMSS implementation in the FlexiProvider and the ASN.1 specification of the keys. Section 5 presents experimental data including a comparison with standard signature schemes. Section 6 describes the integration of the CMSS implementation into Microsoft Outlook. Section 7 states our conclusions.

## 2 Preliminaries

Before we describe CMSS in Section 3, we first describe the Winterzitz one-time signature scheme used in CMSS and the Merkle signature scheme (MSS) which CMSS is based on.

### 2.1 The Winternitz One-time Signature Scheme

In this section, we describe the Winternitz one-time signature scheme (OTSS) that was first mentioned in [Mer89] and explicitly described in [DSS05]. It is a generalization of the Merkle OTSS [Mer89], which in turn is based on the Lamport-Diffie OTSS [DH76]. The security of the Winternitz OTSS is based on the existence of a cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^s$ [MOV96]. It uses a block size parameter $w$ that denotes the number of bits that are processed simultaneously. Algorithms 1, 2, and 3 describe the Winternitz OTSS key pair generation, signature generation, and signature verification, respectively.

**Algorithm 1** Winternitz OTSS Key Pair Generation

**System Parameters:** hash function $H : \{0,1\}^* \to \{0,1\}^s$, parameters $w \in \mathbb{N}$ and $t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$

**Output:** signature key $X$, verification key $Y$

1: choose $x_1, \ldots, x_t \in_R \{0,1\}^s$ uniformly at random.
2: set $X = (x_1, \ldots, x_t)$.
3: compute $y_i = H^{2^w-1}(x_i)$ for $i = 1, \ldots, t$.
4: compute $Y = H(y_1 || \ldots || y_t)$, where $||$ denotes concatenation.
5: **return** $(X, Y)$.

**Algorithm 2** Winternitz OTSS Signature Generation

**System Parameters:** hash function $H : \{0,1\}^* \to \{0,1\}^s$, parameters $w \in \mathbb{N}$ and $t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$

**Input:** document $d$, signature key $X$

**Output:** one-time signature $\sigma$ of $d$

1: compute the $s$ bit hash value $H(d)$ of document $d$.
2: split the binary representation of $H(d)$ into $\lceil s/w \rceil$ blocks $b_1, \ldots, b_{\lceil s/w \rceil}$ of length $w$, padding $H(d)$ with zeros from the left if required.
3: treat $b_i$ as the integer encoded by the respective block and compute the checksum

$$C = \sum_{i=1}^{\lceil s/w \rceil} 2^w - b_i.$$

4: split the binary representation of $C$ into $\lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$ blocks $b_{\lceil s/w \rceil + 1}, \ldots, b_t$ of length $w$, padding $C$ with zeros from the left if required.
5: treat $b_i$ as the integer encoded by the respective block and compute $\sigma_i = H^{b_i}(x_i)$, $i = 1, \ldots, t$, where $H^0(x) := x$.
6: **return** $\sigma = (\sigma_1, \ldots, \sigma_t)$.

**Algorithm 3** Winternitz OTSS Signature Verification

**System Parameters:** hash function $H : \{0,1\}^* \to \{0,1\}^s$, parameters $w \in \mathbb{N}$ and $t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$
**Input:** document $d$, signature $\sigma = (\sigma_1, \ldots, \sigma_t)$, verification key $Y$
**Output:** `TRUE` if the signature is valid, `FALSE` otherwise
1: compute $b_1, \ldots, b_t$ as in Algorithm 2.
2: compute $\phi_i = H^{2^w - 1 - b_i}(\sigma_i)$ for $i = 1, \ldots, t$.
3: compute $\Phi = H(\phi_1 || \ldots || \phi_t)$.
4: **if** $\Phi = Y$ **then return** `TRUE` **else return** `FALSE`

The parameter $w$ makes the Winternitz OTSS very flexible. It allows a trade-off between the size of a signature and the signature and key pair generation times. If $w$ is increased, more bits of $H(d)$ are processed simultaneously and the signature size decreases. But more hash function evaluations are required during key and signature generation. Decreasing $w$ has the opposite effect. In [DSS05], the authors show that using $w = 2$ requires the least number of hash function evaluations per bit.

*Example 1.* Let $w = 2$ and $H(d) = 110001110$. Hence $s = 9$ and $t = 8$. Therefore we have $(b_1, \ldots, b_5) = (01, 10, 00, 11, 10)$, $C = 12$ and $(b_6, b_7, b_8) = (00, 11, 00)$. The signature of $d$ is $\sigma = \big(H(x_1), H^2(x_2), x_3, H^3(x_4), H^2(x_5), x_6, H^3(x_7), x_8\big)$.

### 2.2 The Merkle Signature Scheme

The basic Merkle signature scheme (MSS) [Mer89] works as follows. Let $H : \{0,1\}^* \to \{0,1\}^s$ be a cryptographic hash function and assume that a one-time signature scheme (OTSS) is given. Let $h \in \mathbb{N}$ and suppose that $2^h$ signatures are to be generated that are verifiable with one MSS public key.

**MSS Key Pair Generation** At first, generate $2^h$ OTSS key pairs $(X_i, Y_i)$, $i = 1, \ldots, 2^h$. The $X_i$ are the signature keys. The $Y_i$ are the verification keys. The MSS private key is the sequence of OTSS signature keys. To determine the MSS public key, construct a binary authentication tree as follows. Consider each verification key $Y_i$ as a bit string. The leafs of the authentication tree are the hash values $H(Y_i)$ of the verification keys. Each inner node (including the root) of the tree is the hash value of the concatenation of its two children. The MSS public key is the root of the authentication tree.

**MSS Signature Generation** The OTSS key pairs are used sequentially. We explain the calculation of the MSS signature of some document $d$ using the $i$th key pair $(X_i, Y_i)$. That signature consists of the index $i$, the $i$th verification key $Y_i$, the OTSS signature $\sigma$ computed with the $i$th signature key $X_i$, and the authentication path $A$ for the verification key $Y_i$. The authentication path $A$ is a sequence of nodes $(a_h, \ldots, a_1)$ in the authentication tree of length $h$ that is

constructed as follows. The first node in that sequence is the leaf different from the $i$th leaf that has the same parent as the $i$th leaf. Also, if a node $N$ in the sequence is not the last node, then its successor is the node different from $N$ with the same parent as $N$. Figure 1 shows an example of an authentication path for $h = 2$. Here, the authentication path for $Y_2$ is the sequence $A_2 = (a_2, a_1)$.

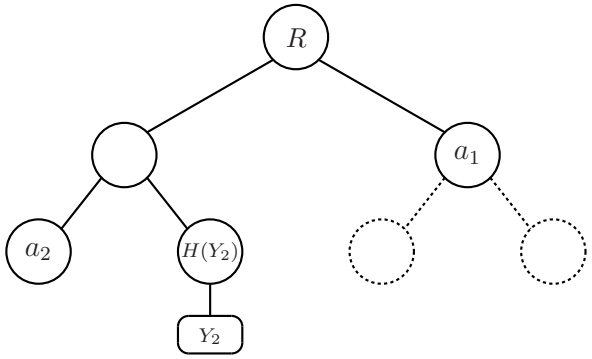


**Fig. 1.** Merkle's Tree Authentication

**MSS Signature Verification** To verify a MSS signature $(i, Y, \sigma, A)$, the verifier first verifies the one-time signature $\sigma$ with the verification key $Y$. If this verification fails, the verifier rejects the MSS signature as invalid. Otherwise, the verifier checks the validity of the verification key $Y$ by using the authentication path $A$. For this purpose, the verifier constructs a sequence of nodes of the tree of length $h+1$. The first node in the sequence is the $i$th leaf of the authentication tree. It is computed as the hash $H(Y)$ of the verification key $Y$. For each node $N$ in the sequence which is not the last node, its successor is the parent $P$ of $N$ in the authentication tree. The verifier can calculate $P$ since the authentication path $A$ included in the signature contains the second child of $P$. The verifier accepts the signature, if the last node in the sequence is the MSS public key.

## 3 CMSS

In this section, we describe CMSS. It is an improvement of the Merkle signature scheme (MSS) [Mer89]. A preliminary version of CMSS including security proofs appeared in the PhD thesis of Coronado [Cor05b] and in [Cor05a].

For any $h \in \mathbb{N}$, MSS signs $N = 2^h$ documents using $N$ key pairs of a one-time signature scheme. Unfortunately, for $N > 2^{25}$, MSS becomes impractical because the private keys are very large and key pair generation takes very long.

CMSS can sign $N = 2^{2h}$ documents for any $h \in \mathbb{N}$. For this purpose, two MSS authentication trees, a main tree and a subtree, each with $2^h$ leafs, are used. The public CMSS key is the root of the main tree. Data is signed using MSS with the subtree. But the root of the subtree is not the public key. That

root is authenticated by an MSS signature that uses the main tree. After the first $2^h$ signatures have been generated, a new subtree is constructed and used to generate the next $2^h$ signatures. In order to make the private key smaller, the OTSS signature keys are generated using a pseudo random number generator (PRNG) [MOV96]. Only the seed for the PRNG is stored in the CMSS private key.

CMSS key pair generation is much faster than that of MSS, since key generation is dynamic. At any given time, only two trees, each with only $2^h$ leafs, have to be constructed. CMSS can efficiently be used to sign up to $N = 2^{40}$ documents. Also, CMSS private keys are much smaller than MSS private keys, since only a seed for the PRNG is stored in the CMSS private key, in contrast to a sequence of $N$ OTSS signature keys in the case of MSS. So, CMSS can be used in any practical application. CMSS is illustrated in Figure 2 for $h = 2$.

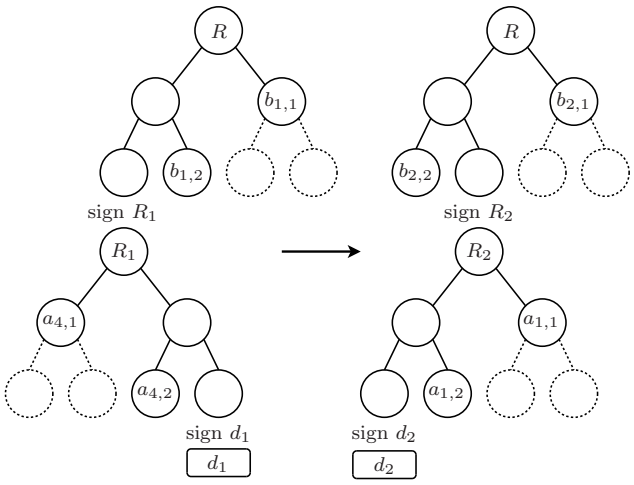


**Fig. 2.** CMSS with $h = 2$

In the following, CMSS is described in detail. First, we describe CMSS key pair generation. Then, we explain the CMSS signature generation process. In contrast to other signature schemes, the CMSS private key is updated after every signature generation. This is necessary in order to keep the private key small and to make CMSS forward secure [Cor05a]. Such signature schemes are called *key-evolving signature schemes* and were first defined in [BM99].

**CMSS Key Pair Generation** Algorithm 6 describes CMSS key pair generation. The algorithm uses two subroutines described in Algorithms 4 and 5. CMSS uses the Winternitz OTSS described in Section 2.1. For the OTSS key pair generation, we use a pseudo random number generator (PRNG) $f : \{0,1\}^s \rightarrow \{0,1\}^s \times \{0,1\}^s$ [MOV96]. In our experiments, we use a PRNG based on SHA1

which is part of the SUN JCE provider [JCA02]. The modified Winternitz OTSS key pair generation process is described in Algorithm 4.

**Algorithm 4** Winternitz OTSS key pair generation using a PRNG

**System Parameters:** PRNG $f : \{0,1\}^s \rightarrow \{0,1\}^s \times \{0,1\}^s$, hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$, parameters $w \in \mathbb{N}$ and $t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$
**Input:** a seed $seed_{in} \in_R \{0,1\}^s$ chosen uniformly at random
**Output:** a Winternitz OTSS key pair $(X, Y)$ and a seed $seed_{out} \in \{0,1\}^s$

1: compute $(seed_{out}, s_0) = f(seed_{in})$
2: **for** $i = 1, \ldots, t$ **do**
3: compute $(s_i, x_i) = f(s_{i-1})$
4: set $X = (x_1, \ldots, x_t)$
5: compute the verification key $Y$ as in steps 3 and 4 of Algorithm 1
6: **return** $(X, Y)$ and $seed_{out}$

Algorithm 5 is used to construct a binary authentication tree and its first authentication path. This is done leaf-by-leaf, using a stack for storing intermediate results. Algorithm 5 carries out the computation for one leaf. It is assumed that in addition to the node value, the height of a node is stored. The algorithm is inspired by [Mer89] and [Szy04].

**Algorithm 5** Partial construction of an authentication tree

**System Parameters:** hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$
**Input:** a leaf value $H(Y)$, algorithm stack *stack*, sequence of nodes $A$
**Output:** updated stack *stack* and updated sequence $A$

1: set $in = H(Y)$
2: **while** $in$ has same height as top node from *stack* **do**
3: **if** $in$ has greater height than last node in $A$ or $A$ is empty **then**
4: append $in$ to $A$
5: pop top node *top* from *stack*
6: compute $in = H(top||in)$
7: push $in$ onto *stack*
8: **return** *stack*, $A$

CMSS key pair generation is carried out in two parts. First, the first subtree and its first authentication path are generated using Algorithms 4 and 5. Then, the main tree and its first authentication path are computed. The CMSS public key is the root of the main tree. The CMSS private key consists of two indices $i$ and $j$, three seeds for the PRNG, three authentication paths (of which one is constructed during signature generation), the root of the current subtree and three algorithm stacks for subroutines. The details are described in Algorithm 6.

**Algorithm 6** CMSS key pair generation

**System Parameters:** hash function $H : \{0,1\}^* \to \{0,1\}^s$, PRNG $f : \{0,1\}^s \to \{0,1\}^s \times \{0,1\}^s$, Winternitz parameter $w$

**Input:** parameter $h \in \mathbb{N}$, two seeds $seed_{main}$ and $seed_{sub}$ chosen uniformly at random in $\{0,1\}^s$

**Output:** a CMSS key pair $(priv, R)$

1: set $N = 2^h$ and $seed_0 = seed_{sub}$
2: initialize empty stack $stack_{sub}$ and empty sequence of nodes $A_1$
3: **for** $i = 1, \ldots, N$ **do**
4: compute $((X_i, Y_i), seed_i) \leftarrow$ `Algorithm 4`$(seed_{i-1})$
5: compute $(stack_{sub}, A_1) \leftarrow$ `Algorithm 5`$(H(Y_i), stack_{sub}, A_1)$
6: let $R_1$ be the single node in $stack_{sub}$; $R_1$ is the root of the first subtree
7: set $seed_{next} = seed_N$ and $seed_0 = seed_{main}$
8: initialize empty stack $stack_{main}$ and empty sequence of nodes $B_1$
9: **for** $j = 1, \ldots, N$ **do**
10: compute $((X_j, Y_j), seed_j) \leftarrow$ `Algorithm 4`$(seed_{j-1})$
11: compute $(stack_{main}, B_1) \leftarrow$ `Algorithm 5`$(H(Y_j), stack_{main}, B_1)$
12: let $R$ be the single node in $stack_{main}$; $R$ is the root of the main tree
13: initialize empty stacks $stack_{main}$, $stack_{sub}$, and $stack_{next}$ and empty sequence of nodes $C_1$
14: set $priv = (1, 1, seed_{\{main,sub,next\}}, A_1, B_1, C_1, R_1, stack_{\{main,sub,next\}})$
15: **return** $(priv, R)$

**CMSS Signature Generation** CMSS signature generation is carried out in four parts. First, the MSS signature of document $d$ is computed using the subtree. Then, the MSS signature of the root of the subtree is computed using the main tree. Then, the next subtree is partially constructed. Finally, the CMSS private key is updated.

The CMSS signature generation algorithm uses an algorithm of Szydlo for the efficient computation of authentication paths. We do not explain this algorithm here but we refer to [Szy04] for details. We call the algorithm `Szydlo.auth`. Input to `Szydlo.auth` are the authentication path of the current leaf, the seed for the current tree and an algorithm stack. Output are the next authentication path and the updated stack. `Szydlo.auth` needs to compute leaf values of leafs with higher index than the current leaf. For this purpose, Algorithm 7 is used. The details of CMSS signature generation are described in Algorithm 8.

**Algorithm 7** `leafcalc`

**System Parameters:** hash function $H : \{0,1\}^* \to \{0,1\}^s$, PRNG $f : \{0,1\}^s \to \{0,1\}^s \times \{0,1\}^s$

**Input:** current leaf index $i$, current seed $seed$, leaf index $j > i$

**Output:** leaf value $H(Y_j)$ of $j$th leaf

1: set $seed_0 = seed$
2: **for** $k = 1, \ldots, j - i$ **do** compute $(seed_k, s_0) = f(seed_{k-1})$
3: compute $((X_j, Y_j), seed_{out}) \leftarrow$ `Algorithm 4`$(seed_{j-i})$
4: **return** $H(Y_j)$

**Algorithm 8** CMSS signature generation

**System Parameters:** hash function $H : \{0,1\}^* \to \{0,1\}^s$
**Input:** document $d$, CMSS private key $priv = (i, j, seed_{main}, seed_{sub}, seed_{next}, A_i, B_j, C_1, R_j, stack_{main}, stack_{sub}, stack_{next})$
**Output:** signature $sig$ of $d$, updated private key $priv$, or `STOP` if no more signatures can be generated

1: **if** $j = 2^h + 1$ **then** `STOP`
2: obtain an OTSS key pair: $((X_i, Y_i), seed_{sub}) \leftarrow$ `Algorithm 4`$(seed_{sub})$
3: compute the one-time signature of $d$: $\sigma_i \leftarrow$ `Algorithm 2`$(d, X_i)$
4: obtain second OTSS key pair: $((X_j, Y_j), seed_{temp}) \leftarrow$ `Algorithm 4`$(seed_{main})$
5: compute the one-time signature of $R_j$: $\tau_j \leftarrow$ `Algorithm 2`$(R_j, X_j)$
6: set $sig = (i, j, \sigma_i, \tau_j, A_i, B_j)$
7: compute the next authentication path for the subtree:
$(A_{i+1}, stack_{sub}) \leftarrow$ `Szydlo.auth`$(A_i, seed_{sub}, stack_{sub})$
and replace $A_i$ in $priv$ by $A_{i+1}$
8: partially construct the next subtree:
$((X_i, Y_i), seed_{next}) \leftarrow$ `Algorithm 4`$(seed_{next})$
$(stack_{next}, C_1) \leftarrow$ `Algorithm 5`$(H(Y_i), stack_{next}, C_1)$
9: **if** $i < 2^h$ **then** set $i = i + 1$
10: **else**
11: let $R_{j+1}$ be the single node in $stack_{next}$; $R_{j+1}$ is the root of the $(j{+}1)$th subtree.
12: compute the next authentication path for the main tree:
$(B_{j+1}, stack_{main}) \leftarrow$ `Szydlo.auth`$(B_j, seed_{main}, stack_{main})$
and replace $B_j$ in $priv$ by $B_{j+1}$
13: replace $R_j$ in $priv$ by $R_{j+1}$, $seed_{main}$ by $seed_{temp}$, and $A_i$ by $C_1$
14: set $i = 1$ and $j = j + 1$
15: **return** the CMSS signature $sig$ of $d$ and the updated private key $priv$

**CMSS Signature Verification** CMSS signature verification proceeds in two steps. First, the two authentication paths are validated, then the validity of the two one-time signatures is verified. The details are described in Algorithm 9.

**Algorithm 9** CMSS signature verification

**System Parameters:** hash function $H : \{0,1\}^* \to \{0,1\}^s$
**Input:** document $d$, CMSS signature $sig = (i, j, \sigma_i, \tau_j, A_i, B_j)$, CMSS public key $R$
**Output:** `TRUE` if the signature is valid, `FALSE` otherwise.

1: repeat steps 1 to 3 of Algorithm 3 with input $d$ and $\sigma_i$ to obtain an alleged verification key $\Phi_i$
2: using $\Phi_i$ and $A_i$, compute the root $R_j$ of the current subtree as in the case of MSS signature verification (see Section 2.2).
3: repeat steps 1 to 3 of Algorithm 3 with input $R_j$ and $\tau_j$ to obtain an alleged verification key $\Psi_j$
4: using $\Psi_j$ and $B_j$, compute the root $Q$ of the main tree as in the case of MSS.
5: **if** $Q$ is not equal to the CMSS public key $R$ **then return** `FALSE`
6: verify the one-time signature $\sigma_i$ of $d$ using Algorithm 3 and verification key $\Phi_i$
7: verify the one-time signature $\tau_j$ of $R_j$ using Algorithm 3 and verification key $\Psi_j$
8: **if** both verifications succeed **return** `TRUE` **else return** `FALSE`

## 4 Specification and Implementation

This section describes parameter choices and details of our CMSS implementation. CMSS is implemented as part of the Java Cryptographic Service Provider (CSP) FlexiProvider [Flexi]. It is therefore possible to integrate the implementation into any application that uses the Java Cryptographic Architecture [JCA02] and Java Cryptography Extension [JCE02]. Our CMSS implementation is available at [Flexi] as open source software.

**Scheme Parameters** The hash function $H$ used in the OTSS and the authentication trees can be chosen among SHA1, SHA256, SHA384, and SHA512. The Winternitz parameter $w$ can be chosen among $1, 2, 3$, and 4. As PRNG $f$, we use a PRNG based on SHA1 which is part of the SUN JCE provider [JCA02]. For each choice, there exists a distinct object identifier (OID) that can be found in Appendix B.

As described earlier, CMSS makes use of the Winternitz OTSS. However, it is possible to replace the Winternitz OTSS by any other one-time signature scheme. If unlike in the case of Winternitz OTSS the verification keys can not be computed from the signature keys, they have to be part of the CMSS signature. Also, the PRNG based on SHA1 can be replaced by any other PRNG.

**Key Generation** The CMSS private and public keys are stored using Abstract Syntax Notation One (ASN.1) [Int02]. ASN.1 ensures interoperability between different applications and also allows efficient generation of X.509 certificates and PKCS#12 personal information exchange files. The ASN.1 encoding of the keys can be found in Appendix A. In addition to what was described in Section 3, both the CMSS public and private key contain the OID of the algorithm they can be used with.

**Signature Generation and Verification** For the computation of authentication paths, we use the preprint version of the algorithm `Szydlo.auth` which is more efficient than the conference version. See [Szy04] for details.

Each time a new CMSS signature is computed, the signature of the root of the current subtree is recomputed. This reduces the size of the CMSS private key. The time required to recompute this MSS signature is tolerable.

## 5 Experimental Results

This section compares the CMSS implementation with RSA, DSA, and ECDSA. We compare the times required for key pair generation, signature generation, and signature verification as well as the sizes of the private key, public key, and signatures. For RSA, DSA, and ECDSA, the implementations provided by the Java CSP FlexiProvider are used, which is available at [Flexi] as open source software.

The results are summarized in Table 1. In case of CMSS, the first column denotes the logarithm to the base 2 of the number of possible signatures $N$. For RSA, DSA, and ECDSA, the column *mod* denotes the size of the modulus. The size of the keys is the size of their DER encoded ASN.1 structure.

The experiments were made using a computer equipped with a Pentium M 1.73GHz CPU, 1GB of RAM and running Microsoft Windows XP.

| $\log N$ | $s_{public\ key}$ | $s_{private\ key}$ | $s_{signature}$ | $t_{keygen}$ | $t_{sign}$ | $t_{verify}$ |
|---|---|---|---|---|---|---|
| CMSS with SHA1, $w = 1$ | | | | | | |
| 20 | 46 bytes | 1900 bytes | 7168 bytes | 2.9 s | 10.2 ms | 1.2 ms |
| 30 | 46 bytes | 2788 bytes | 7368 bytes | 1.5 min | 13.6 ms | 1.2 ms |
| 40 | 46 bytes | 3668 bytes | 7568 bytes | 48.8 min | 17.5 ms | 1.2 ms |
| CMSS with SHA1, $w = 2$ | | | | | | |
| 20 | 46 bytes | 1900 bytes | 3808 bytes | 2.6 s | 9.2 ms | 1.3 ms |
| 30 | 46 bytes | 2788 bytes | 4008 bytes | 1.4 min | 12.4 ms | 1.4 ms |
| 40 | 46 bytes | 3668 bytes | 4208 bytes | 43.8 min | 14.9 ms | 1.3 ms |
| CMSS with SHA1, $w = 3$ | | | | | | |
| 20 | 46 bytes | 1900 bytes | 2688 bytes | 3.1 s | 9.7 ms | 1.5 ms |
| 30 | 46 bytes | 2788 bytes | 2888 bytes | 1.5 min | 13.2 ms | 1.5 ms |
| 40 | 46 bytes | 3668 bytes | 3088 bytes | 47.8 min | 16.9 ms | 1.6 ms |
| CMSS with SHA1, $w = 4$ | | | | | | |
| 20 | 46 bytes | 1900 bytes | 2128 bytes | 4.1 s | 12.5 ms | 2.0 ms |
| 30 | 46 bytes | 2788 bytes | 2328 bytes | 2.0 min | 17.0 ms | 2.0 ms |
| 40 | 46 bytes | 3668 bytes | 2528 bytes | 62.3 min | 21.7 ms | 2.0 ms |
| *mod* | $s_{public\ key}$ | $s_{private\ key}$ | $s_{signature}$ | $t_{keygen}$ | $t_{sign}$ | $t_{verify}$ |
| RSA with SHA1 | | | | | | |
| 1024 | 162 bytes | 634 bytes | 128 bytes | 0.4 s | 13.8 ms | 0.8 ms |
| 2048 | 294 bytes | 1216 bytes | 256 bytes | 3.4 s | 96.8 ms | 3.0 ms |
| DSA with SHA1 | | | | | | |
| 1024 | 440 bytes | 332 bytes | 46 bytes | 18.2 s | 8.2 ms | 16.2 ms |
| ECDSA with SHA1 | | | | | | |
| 192 | 246 bytes | 231 bytes | 55 bytes | 5.1 ms | 5.1 ms | 12.9 ms |
| 256 | 311 bytes | 287 bytes | 71 bytes | 9.6 ms | 9.8 ms | 24.3 ms |
| 384 | 441 bytes | 402 bytes | 102 bytes | 27.3 ms | 27.3 ms | 66.9 ms |

**Table 1.** Timings for CMSS, RSA, DSA, and ECDSA

The table shows that the CMSS implementation offers competitive signing and verifying times compared to RSA, DSA, and ECDSA. The table also shows that a CMSS public key is significantly smaller than a RSA or a DSA public key.

In the case of $N = 2^{40}$, key pair generation takes quite long. However, this does not affect the usability of the implementation, since key pair generation has to be performed only once. Also, the size of the signature and the private key is larger compared to RSA and DSA. While this might lead to concerns regarding memory constrained devices, those sizes are still reasonable in an end-user scenario.

To summarize, CMSS offers a very good trade-off concerning signature generation and verification times compared to RSA and DSA while preserving a reasonable signature and private key size. Appendix C contains a table showing timings for CMSS with SHA256.

## 6 Signing Messages in Microsoft Outlook with CMSS

Section 5 showed that the space and time requirements of our CMSS implementation are sufficiently small for practical usage. Also, the number of signatures that can be generated is large enough for practical purposes.

The implementation can be easily integrated in applications that use the JCA. An example for such an application is the `FlexiS/MIME` Outlook plug-in [FOP03], which enables users to sign and encrypt emails using any Java Cryptographic Service Provider in a fast and easy way. The plugin is available at [FOP03] as a free download and is compatible with Microsoft Outlook 98, 2000, 2002, XP and 2003.

In addition to the basic functions like key pair generation, signature generation and verification, the plug-in also supports the generation of self-signed X.509 certificates and PKCS#10 conform certification requests for a certification authority. Furthermore, it is possible to import and export X.509 certificates and PKCS#12 personal information exchange files.

Using the `FlexiS/MIME` Outlook plug-in in conjunction with the FlexiProvider implementation, we are able to sign emails with CMSS. Furthermore, CMSS can be easily integrated into existing public-key infrastructures.

## 7 Conclusion

In this paper, we present CMSS, an improved Merkle signature scheme with significantly reduced private key size, key pair generation, and signature generation times. We describe an efficient CMSS FlexiProvider implementation. The implementation provides competitive or even superior timings compared to the commonly used signature schemes RSA, DSA, and ECDSA. This demonstrates that it is already possible today to use quantum computer resistant signature schemes without any loss of efficiency concerning signature generation and verification times and with reasonable signature and key lengths. Using CMSS, it is possible to sign up to $2^{40}$ messages, while preserving moderate key pair generation times. Because CMSS is implemented as part of a Java Cryptographic Service Provider, it can be used with any application that uses the JCA, e.g. the `FlexiS/MIME` plug-in, which can be used to sign emails with Microsoft Outlook.

## References


[BM99] M. Bellare and S. Miner. A Forward-Secure Digital Signature Scheme. In *Advances in Cryptology – CRYPTO '99*, number 1666 in LNCS, pages 431–448. Springer, 1999.

[Cor05a] L. C. Coronado García. On the security and the efficiency of the Merkle signature scheme. Technical Report 2005/192, Cryptology ePrint Archive, 2005. Available at `http://eprint.iacr.org/2005/192/`.

[Cor05b] L. C. Coronado García. *Provably Secure and Practical Signature Schemes*. PhD thesis, Computer Science Departement, Technical University of Darmstadt, 2005. Available at `http://elib.tu-darmstadt.de/diss/000642/`.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[DSS05] C. Dods, N. P. Smart, and M. Stam. Hash Based Digital Signature Schemes. In *Cryptography and Coding*, number 3796 in LNCS, pages 96–115. Springer, 2005.

[Elg85] T. Elgamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology – CRYPTO '84*, number 196 in LNCS, pages 10–18. Springer, 1985.

[Flexi] The FlexiProvider group at Technische Universität Darmstadt. *FlexiProvider, an open source Java Cryptographic Service Provider*, 2001–2006. Available at `http://www.flexiprovider.de/`.

[FOP03] The FlexiPKI research group at Technische Universität Darmstadt. *The FlexiS/MIME Outlook Plugin*, 2003. Available at `http://www.informatik.tu-darmstadt.de/TI/FlexiPKI/FlexiSMIME/FlexiSMIME.html`.

[Int02] International Telecommunication Union. *X.680: Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation*, 2002. Available at `http://www.itu.int/rec/T-REC-X.680/`.

[JCA02] Sun Microsystems. *The Java Cryptography Architecture API Specification & Reference*, 2002. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html`.

[JCE02] Sun Microsystems. *The Java Cryptography Extension (JCE) Reference Guide*, 2002. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html`.

[JM99] D. Johnson and A. Menezes. The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical Report CORR 99-34, University of Waterloo, 1999. Available at `http://www.cacr.math.uwaterloo.ca`.

[Mer89] R. Merkle. A certified digital signature. In *Advances in Cryptology – CRYPTO '89*, number 1462 in LNCS, pages 218–238. Springer, 1989.

[MOV96] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, Boca Raton, Florida, 1996. Available at `http://cacr.math.uwaterloo.ca/hac/`.

[NSW05] D. Naor, A. Shenhav, and A. Wool. One-Time Signatures Revisited: Have They Become Practical? Technical Report 2005/442, Cryptology ePrint Archive, 2005. Available at `http://eprint.iacr.org/2005/442/`.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Sho94] P. W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1994)*, pages 124–134. IEEE Computer Society Press, 1994.

[Szy04] M. Szydlo. Merkle Tree Traversal in Log Space and Time. In *Advances in Cryptology – EUROCRYPT 2004*, number 3027 in LNCS, pages 541–554. Springer, 2004. Preprint version (2003) available at `http://szydlo.com/`.

## A ASN.1 Encoding

This section describes the specification of the CMSS public and private keys using Abstract Syntax Notation number One (ASN.1) [Int02].

```
CMSSPublicKey  ::= SEQUENCE {
    algorithm      OBJECT IDENTIFIER
    height         INTEGER
    root           OCTET STRING
}

CMSSPrivateKey ::= SEQUENCE {
    algorithm      OBJECT IDENTIFIER
    counterSub     INTEGER
    counterMain    INTEGER
    seedMain       OCTET STRING
    seedSub        OCTET STRING
    seedNext       OCTET STRING
    authMain       AuthPath
    authSub        AuthPath
    authNext       AuthPath
    stackMain      Stack
    stackSub       Stack
    stackNext      Stack
}

AuthPath       ::= SEQUENCE OF OCTET STRING
Stack          ::= SEQUENCE OF OCTET STRING
```

## B Object Identifiers

This section lists the object identifiers (OIDs) assigned to our CMSS implementation. The main OID for CMSS as well as the OID for the CMSSKeyFactory is

`1.3.6.1.4.1.8301.3.1.3.2`

The OIDs for CMSS are summarized in the following table, where the column "Hash function" denotes the hash function used in the OTSS and the authentication trees, and the column "$w$" denotes the Winternitz parameter $w$.

| Hash function | $w$ | Object Identifier (OID) |
|---|---|---|
| SHA1 | 1 | 1.3.6.1.4.1.8301.3.1.3.2.1 |
| SHA1 | 2 | 1.3.6.1.4.1.8301.3.1.3.2.2 |
| SHA1 | 3 | 1.3.6.1.4.1.8301.3.1.3.2.3 |
| SHA1 | 4 | 1.3.6.1.4.1.8301.3.1.3.2.4 |
| SHA256 | 1 | 1.3.6.1.4.1.8301.3.1.3.2.5 |
| SHA256 | 2 | 1.3.6.1.4.1.8301.3.1.3.2.6 |
| SHA256 | 3 | 1.3.6.1.4.1.8301.3.1.3.2.7 |
| SHA256 | 4 | 1.3.6.1.4.1.8301.3.1.3.2.8 |
| SHA384 | 1 | 1.3.6.1.4.1.8301.3.1.3.2.9 |
| SHA384 | 2 | 1.3.6.1.4.1.8301.3.1.3.2.10 |
| SHA384 | 3 | 1.3.6.1.4.1.8301.3.1.3.2.11 |
| SHA384 | 4 | 1.3.6.1.4.1.8301.3.1.3.2.12 |
| SHA512 | 1 | 1.3.6.1.4.1.8301.3.1.3.2.13 |
| SHA512 | 2 | 1.3.6.1.4.1.8301.3.1.3.2.14 |
| SHA512 | 3 | 1.3.6.1.4.1.8301.3.1.3.2.15 |
| SHA512 | 4 | 1.3.6.1.4.1.8301.3.1.3.2.16 |

**Table 2.** OIDs assigned to CMSS

## C CMSS Timings Using SHA256

| $\log N$ | $s_{public\ key}$ | $s_{private\ key}$ | $s_{signature}$ | $t_{keygen}$ | $t_{sign}$ | $t_{verify}$ |
|---|---|---|---|---|---|---|
| CMSS with SHA256, $w = 1$ | | | | | | |
| 20 | 58 bytes | 2884 bytes | 17672 bytes | 7.0 s | 23.4 ms | 2.9 ms |
| 30 | 58 bytes | 4244 bytes | 17992 bytes | 3.8 min | 32.3 ms | 3.3 ms |
| 40 | 58 bytes | 5604 bytes | 18312 bytes | 120.9 min | 41.3 ms | 3.3 ms |
| CMSS with SHA256, $w = 2$ | | | | | | |
| 20 | 58 bytes | 2884 bytes | 9160 bytes | 6.3 s | 19.6 ms | 2.8 ms |
| 30 | 58 bytes | 4244 bytes | 9480 bytes | 3.2 min | 27.3 ms | 2.8 ms |
| 40 | 58 bytes | 5604 bytes | 9800 bytes | 101.3 min | 34.9 ms | 2.9 ms |
| CMSS with SHA256, $w = 3$ | | | | | | |
| 20 | 58 bytes | 2884 bytes | 6408 bytes | 7.5 s | 23.3 ms | 3.7 ms |
| 30 | 58 bytes | 4244 bytes | 6728 bytes | 3.8 min | 31.9 ms | 3.7 ms |
| 40 | 58 bytes | 5604 bytes | 7048 bytes | 120.7 min | 40.9 ms | 3.7 ms |
| CMSS with SHA256, $w = 4$ | | | | | | |
| 20 | 58 bytes | 2884 bytes | 4936 bytes | 10.2 s | 31.6 ms | 5.1 ms |
| 30 | 58 bytes | 4244 bytes | 5256 bytes | 5.2 min | 43.4 ms | 5.1 ms |
| 40 | 58 bytes | 5604 bytes | 5576 bytes | 165.5 min | 55.8 ms | 5.1 ms |

**Table 3.** Timings for CMSS with SHA256