Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis

Georg Becker

18.07.08

Seminararbeit
Ruhr-Universität Bochum

Chair for Embedded Security
Prof. Dr.-Ing. Christof Paar

Contents

1 Introduction **1**

2 One-Time Signatures **2**
2.1 Secure hash functions 2
2.2 Lamport One-Time Signature Scheme 3
2.2.1 Key generation 3
2.2.2 Signing a message 3
2.2.3 Signature verification 3
2.3 Winternitz One-time Signature Scheme 4
2.3.1 Key generation 4
2.3.2 Signature generation 5
2.3.3 Signature verification 6
2.3.4 Choosing parameter w 6

3 Merkle-Signature Scheme **8**
3.1 Key generation 8
3.2 Signature generation 9
3.3 Signature verification 9
3.4 Cost analysis 10

4 The Merkle tree traversal problem **12**
4.1 The treehash algorithm 12
4.2 The classic traversal 13
4.3 Merkle tree traversal in log space and time 16
4.4 Fractal tree representation and traversal 18

5 Improvements to Merkle-Signature Scheme **20**

6 Cryptanalysis **22**
6.1 Case 1: $H(Y_i') = H(Y_i)$ 22
6.2 Case 2: $H(Y_i') \neq H(Y_i)$ 22
6.3 Differential Side Channel resistant 23

7 Conclusion **24**

1 Introduction

In todays world, digital signatures are an indispensable element for secured communication applications. They are needed, to ensure the authentication of a communication partner, i.e. in web services like Email or chats. They are also needed, to ensure the authentication of a web server for web services like webshops or online-banking. But digital signatures are not just used in web services. For example, they can also be used to verify the validity of digital passports or other digital documents.

Popular signature schemes are the Digital Signature Scheme (DSA) and the RSA Signature Scheme. The security of these schemes rely on the difficulties of solving the discrete logarithm problem and the problem of factorizing large numbers. Today, no efficient algorithms are known to solve these problems, so that theses schemes are considered secure. However, it is not proven that no such algorithms exist. If the mathematicians are able to find a suitable algorithm, these signature schemes would become insecure. Furthermore, there are already algorithms known to solve these problems in case a quantum computer can be build. Some scientists believe it might be possible to build a quantum computer in about 20 years. Therefore, alternative digital signature schemes are needed, in case the signature schemes based on the discrete logarithm problem or the factorization problem become insecure.

The Merkle Signature Scheme provides such an alternative signature scheme. As we will see in chapter 6, the security of the Merkle Signature Scheme only depends on a secure hash function and a secure one-time signature. The characteristics of secure hash functions are described in section 2.1 and two examples of secure one-time signature schemes are given in section 2.2 and 2.3. In chapter 3 the Merkle Signature Scheme will be introduced. In chapter 4, methods for solving the Merkle Tree Traversal Problem are described. Efficient methods to solve this problem are needed to make the signature scheme feasible. In chapter 5 further improvements to the original Merkle Signature Scheme are introduced, which make the signature scheme more efficient.

2 One-Time Signatures

2.1 Secure hash functions

The security of One-Time Signatures is based on cryptographic secure hash functions. In this section we will define the properties of a cryptographic secure hash function. A hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$ is cryptographic secure, if it is "preimage resistant", "second preimage resistant" and "collision resistant".

- Preimage resistant:

  A hash function H is preimage resistant, if it is hard to find any m for a given h with $h = H(m)$.

- Second preimage resistance:

  A hash function H is second preimage resistant if it is hard to find for a given m_1 any m_2 with $H(m_1) = H(m_2)$.

- Collision resistant:

  A hash function H is collision resistant if it is hard to find a pair of m_1 and m_2 with $H(m_1) = H(m_2)$.

For a good cryptographic secure hash function no algorithm should be known, which solves the preimage resistance and second preimage resistance problem more efficient, than a brute force attack. In a brute force attack against preimage resistance, the attacker chooses m randomly until $h = H(m)$. $H(m)$ can have 2^s different results. Each result h should appear with the same probability P_h with $P_h = 1/(2^s)$. Therefore, an attacker would have to choose on average $2^s/2$ different inputs m, until he finds an m with $h = H(m)$. With the same idea, we find out that the attacker would need on average $2^s/2$) different inputs m_2, until he finds an m_2 with $H(m_1) = H(m_2)$. So the complexity of an attack against preimage and second preimage resistance is $1/2 * 2^s = O(2^s)$, with s being the length of the result of the hash function in bits.

Unfortunately, breaking collision resistance by finding a pair of m_1 and m_2 with $H(m_1) = H(m_2)$ is easier than breaking preimage resistance. This is due to the birthday problem (also referred to as birthday paradox).

Birthday problem: [AMV96] An urn has m balls numbered 1 to m. Suppose that one ball at a time is drawn from the urn with replacement. On average about $\sqrt{\frac{\pi m}{2}}$ balls must be drawn until one ball is drawn twice.

So referred to the complexity to find a pair m_1, m_2 with $H(m_1) = H(m_2)$ this means that in average $\sqrt{\frac{\pi 2^s}{2}} \approx O(\sqrt{2^s})$ hash operations must be performed. To achieve a collision resistance with a security of at least $O(2^{80})$ the size of the hash function $H : \{0,1\}^* \rightarrow \{0,1\}^s$ must be at least 160 bits because $O(\sqrt{2^{160}}) = O(2^{80})$.

2.2 Lamport One-Time Signature Scheme

The Lamport One-Time Signature Scheme (LOTSS) is a signature scheme in which the public key can only be used to sign a single message. The security of the LOTSS is based on cryptographic hash functions. Any secure hash function can be used, which makes this signature scheme very adjustable. If a hash function becomes insecure it can easily be exchanged by another secure hash function. In the following first the key generation, then the signing algorithm and finally the verification algorithm are described.

2.2.1 Key generation

Let $H : \{0,1\}^* - > H : \{0,1\}^s$ be a cryptographic hash function. To sign a message $M = (0,1)^k$ choose $2 * k$ random numbers X_{ij} with $1 \leq i \leq k$ and $j = \{0,1\}$. For each i and j compute $Y_{ij} = H(X_{ij})$. These $2 * k$ values Y_{ij} are the public key, while the X_{ij} values are the private key.

2.2.2 Signing a message

Given is a message $M = m_1, m_2, ..., m_k$ with $m_i \in \{0,1\}$ and the private keys X_{ij} with $1 \leq i \leq k$ and $j = \{0,1\}$. For each i it is checked whether m_i equals 0 or 1. If it equals 0 then $sig_i = X_{i0}$ otherwise $sig_i = X_{i1}$. The signature sig is the concatenation of all sig_i for $i = \{1, ..., k\}$. So $sig = (sig_1||sig_2||...||sig_k)$ with $||$ denotes the concatenation of two values.

2.2.3 Signature verification

Let $sig = (sig_1||sig_2||...||sig_k)$ be the signature of a given message $M = m_1, m_2, ..., m_k$ with $m_i \in \{0,1\}$ and Y_{ij} the corresponding public key of the Lamport One-Time Signature Scheme. For each $1 \leq i \leq k$ the hash value $H(sig_i)$ gets computed. If $m_i = 0$ then $H(sig_i)$ must be $H(sig_i) = Y_{i0}$ otherwise $H(sig_i)$ must be $H(sig_i) = Y_{i1}$ to be a valid signature.

2.3 Winternitz One-time Signature Scheme

One major disadvantage of the Lamport One-Time Signature Scheme is the big size of the public and private key. To sign a message $M = \{0,1\}^k$, $2 * k$ hash values have to be saved. To achieve a security of at least $O(2^{80})$, a hash function must have at least 160 bits. Therefore, the private and the public key must have at least $160 * 2 * k = 320 * k$ bits. In most cases a message will be hashed before it is signed, so that the size k of the message M will also be 160 bits long. This results in a total size of the public and private key of $160*2*160$ bits $= 51200$ bits $= 6400$ bytes. Hence, a public key of the Lamport One-Time Signature Scheme is 50 times larger than an equivalent 1024-bit RSA public key. The signature sig is the concatenation of k hash values. For $k = 160$, the signature size is $160 * k$ bits $= 25600$ bits $= 3200$ bytes. Hence a LOTSS signature is about 25 times bigger than an equivalent 1024-bit RSA signature. In the Winternitz One-time Signature Scheme the signature size can be reduced at the cost of hash operations.

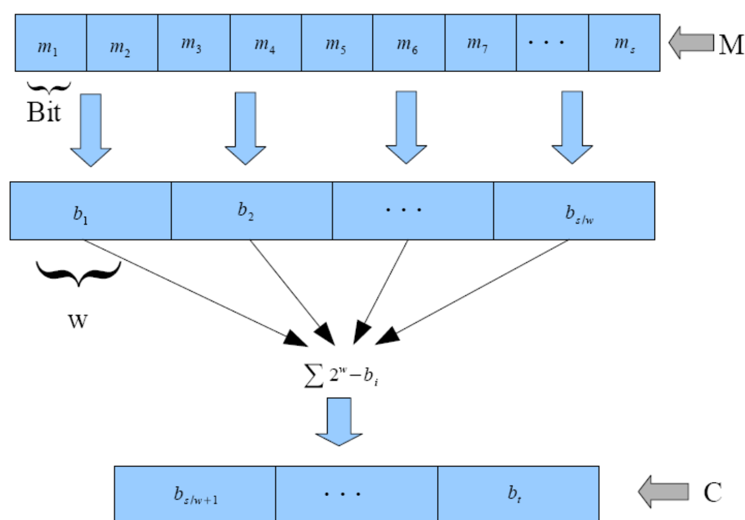

Figure 2.1: Building the values b_i and the checksum C

2.3.1 Key generation

Let $H : \{0,1\}^* \rightarrow H : \{0,1\}^s$ be a cryptographic hash function. At first the parameter w, with $w \in N$, is chosen and and $t = \lceil s/w \rceil + \lceil (\lfloor log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$ gets calculated. A larger parameter w reduces the signature size but increases the calculation time. We now choose t random numbers $X_1, ..., X_t \in \{0,1\}^s$. These random numbers are the private key $X = (X_1||...||X_t)$. In the next step the public key Y is generated by computing $Y_i = H^{2^w-1}(X_i)$ for $i = 1, ..., t$. The

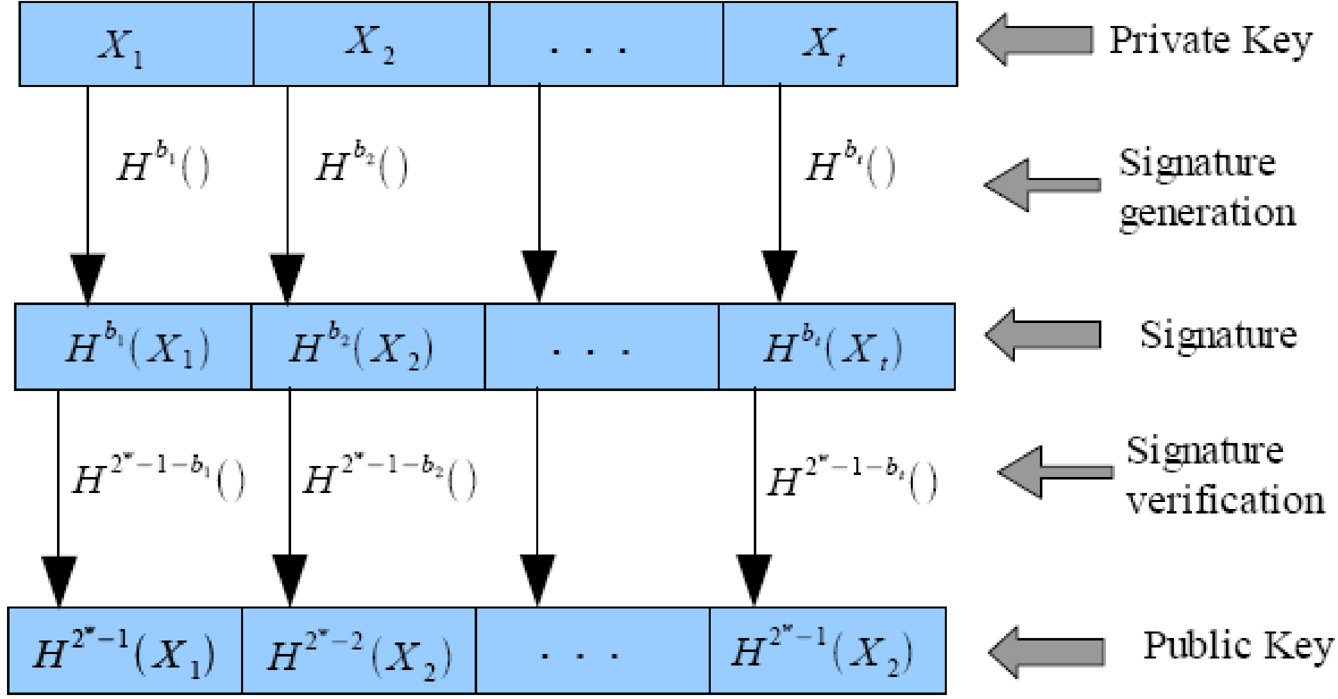

Figure 2.2: Signature generation and verification with the Winternitz One-Time Signature Scheme

Public key $Y = H(Y_1||...||Y_t)$ is the hash value of the concatenation of all Y_i with $i = 1, ..., t$.

2.3.2 Signature generation

Let $M = m_1, ..., m_s \in \{0, 1\}$ be the message to be signed, $X_1, ..., X_t$ the private key and w and t the parameters as described above. The message M is split up into $\lceil s/w \rceil$ blocks $b_1, ..., b_{\lceil s/w \rceil}$ of the length w. If necessary the message is padded with zeros from the left first. We now treat b_i as the integer encoded by the respective block and compute the checksum $C = \sum_{i=1}^{\lceil s/w \rceil} 2^w - b_i$. We then split the binary representation of C into $\lceil (\lfloor log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$ blocks $b_{\lceil s/w \rceil+1}, ..., b_t$ of length w. If necessary C is padded with zeros from the left. We now treat b_i as the integer encoded by the the block b_i and compute $sig_i = H^{b_i}(X_i)$ for $i = 1, ..., t$ with $H^0(X_i) = X_i$. The signature $sig = (sig_1||...||sig_t)$ of the message M is the concatenation of all sig_i for $i = 1, ..., t$.

2.3.3 Signature verification

To verify a signature $sig = (sig_1||...||sig_t)$ for a given message $M = \{0,1\}^s$ the parameters $b_1, ..., b_t$ are computed first. This is done in the same way as during the signature generation. For $i = 1, ..., t$ $sig'_i = H^{2^w-1-b_i}(sig_i)$ is calculated. $sig'_i = H^{2^w-1-b_i}(sig_i) = H^{2^w-1-b_i}(H^{b_i}(X_i)) = H^{2^w-1}(X_i) = Y_i$.

Hence if $Y' = H(sig'_1||...||sig'_t)$ equals $Y = H(Y_1||...||Y_t)$ the signature is valid. Otherwise the signature is refused.

2.3.4 Choosing parameter w

The Winternitz One-Time Signature Scheme is very flexible due to the parameter w. With the help of this parameter, a trade off between signature size and computation time can be made. Choosing a bigger parameter w will result in a smaller signature size, but the signature generation time and the signature verification time will increase. We will now analyze the signature size depending on the parameter w.

Signature Size: The signature $sig = (sig_1||...||sig_t)$ contains t blocks of sig_i. Each block has the length of one output of the hash function. Hence the bit size of the signature $|sig|$ is $|sig| = t*s = \lceil s/w \rceil + \lceil (\lfloor log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil * s \approx s/w$. So the signature size is about inversely proportional to the parameter w.

We will now analyze the impact of the parameter w on the calculation time in each phase of the Winternitz-Signature Scheme.

Key generation time (gen_{time}): During the key generation t random numbers X_i must be chosen and $H^{2^w-1}(X_i)$ must be computed for $t \approx s/w$ values X_i. Therefore $gen_{time} \approx s/w * (2^w - 1) * hash_{time} + s/w * rand_{time} = O(2^w) * hash_{time} + O(1/w) * rand_{time}$ with $hash_{time}$ being the time for one hash operation and $rand_{time}$ being the time to generate one random number. So the key generation time increases exponentially with the size of w.

Signature time (sig_{time}): To generate the signature $sig = (sig_1||...||sig_t)$ the value sig_i must be computed $t \approx s/w$ times. To generate one $sig_i = H^{b_i}(X_i)$ with $b_i <= 2^w - 1$ in average $(\sum_{j=1}^{w-1} 2^j)/w = \frac{2^w-2}{w}$ hash operations must be performed. This results in a signing cost of $sig_{time} \approx s/w * (2^w - 2)/w * hash_{time} = s * (2^w - 2)/w^2 * hash_{time} = O(2^w)$.

Verification time (ver_{time}): To verify a message sig'_i must be computed $t \approx s/w$ times. To calculate one $sig_i = H^{2^w-1-b_i}$ with $b_i <= 2^w - 1$ on average $(\sum_{j=1}^{w-1} 2^j)/w = \frac{2^w-2}{w}$ hash operations must be performed. So the verification

time is the same as the signature time: $ver_{time} = sig_{time} \approx s * (2^w - 2)/w^2 * hash_{time} = O(2^w)$.

Hence, the optimal value for parameter w depends on the available resources. If signing is fast enough, w can be increased to reduce the signature size. But the signature time increases exponentially, while the signature size decreases linearly, so that choosing a too big value for w is not recommended.

3 Merkle-Signature Scheme

The biggest problem of One-Time Signature Schemes is the key management. Exchanging a public key is very complex. It must be guaranteed, that the public key belongs to the intended communication partner and that the public key has not been modified. Therefore, few public keys should be used and the public keys should be rather short. But in One-Time Signature Schemes, a new public key is used for every signature and the public key is quite big, compared with other signature schemes. To make One-Time Signature Schemes feasible, an efficient key management, that reduces the amount of public keys and their size, is needed. In [Mer79] Merkle introduced the Merkle Signature Scheme (MSS), in which one public key is used to sign many messages.

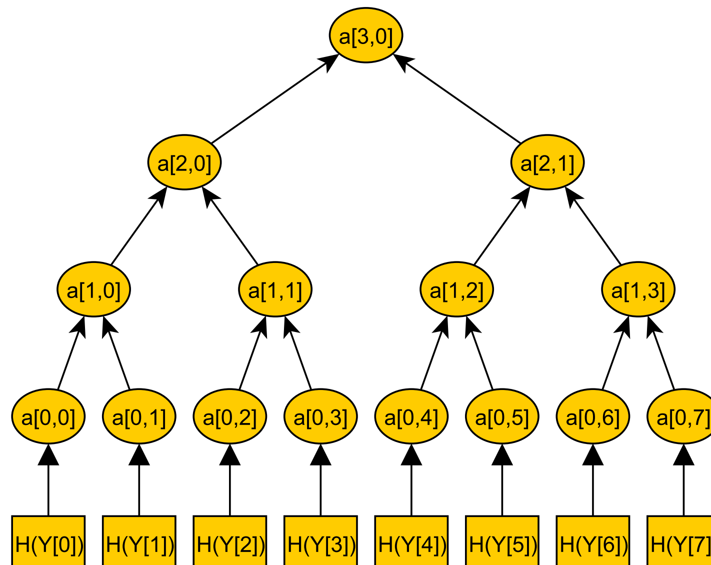

Figure 3.1: Merkle tree with 8 leafs

3.1 Key generation

The Merkle Signature Scheme can only be used to sign a limited number of messages with one public key *pub*. The number of possible messages must be a

power of two, so that we denote the possible number of messages as $N = 2^n$. The first step of generating the public key *pub* is to generate the public keys X_i and private keys Y_i of 2^n one-time signatures, as described in chapter 2. For each public key Y_i, with $1 \leq i \leq 2^n$, a hash value $h_i = H(Y_i)$ is computed. With these hash values h_i a Merkle Tree (also called hash tree) is build. We call a node of the tree $a_{i,j}$, where i denotes the level of the node. The level of a node is defined by the distance from the node to a leaf. Hence, a leaf of the tree has level $i = 0$ and the root has level $i = n$. We number all nodes of one level from the left to the right, so that $a_{i,0}$ is the leftmost node of level i. In the Merkle Tree the hash values h_i are the leafs of a binary tree, so that $h_i = a_{0,i}$. Each inner node of the tree is the hash value of the concatenation of its two children. So $a_{1,0} = H(a_{0,0}||a_{0,1})$ and $a_{2,0} = H(a_{1,0}||a_{1,1})$. An example of a merkle tree is illustrated in figure 3.1 .

In this way, a tree with 2^n leafs and $2^{n+1} - 1$ nodes is build. The root of the tree $a_{n,0}$ is the public key *pub* of the Merkle Signature Scheme.

3.2 Signature generation

To sign a message M with the Merkle Signature Scheme, the message M is signed with a one-time signature scheme, resulting in a signature sig', first. This is done, by using one of the public and private key pairs $(X_i, Y_i,)$. The corresponding leaf of the hash tree to a one-time public key Y_i is $a_{0,i} = H(Y_i)$. We call the path in the hash tree from $a_{0,i}$ to the root A. The path A consists of $n + 1$ nodes, $A_0, ...A_n$, with $A_0 = a_{0,i}$ being the leaf and $A_n = a_{n,0} = pub$ being the root of the tree. To compute this path A, we need every child of the nodes $A_1, ..., A_n$. We know that A_i is a child of A_{i+1}. To calculate the next node A_{i+1} of the path A, we need to know both children of A_{i+1}. So we need the brother node of A_i. We call this node $auth_i$, so that $A_{i+1} = H(A_i||auth_i)$. Hence, n nodes $auth_0, ..., auth_{n-1}$ are needed, to compute every node of the path A. We now calculate and save these nodes $auth_0, ..., auth_{n-1}$. How this is done efficiently is discussed in chapter 4. These nodes, plus the one-time signature sig' of M is the signature $sig = (sig'||auth_2||auth_3||...||auth_{n-1})$ of the Merkle Signature Scheme. An example of an authentication path is illustrated in figure 3.2.

3.3 Signature verification

The receiver knows the public key *pub*, the message M, and the signature $sig = (sig'||auth_0||auth_1||...||auth_{n-1})$. At first, the receiver verifies the one-time signature sig' of the message M. If sig' is a valid signature of M, the receiver computes $A_0 = H(Y_i)$ by hashing the public key of the one-time signature. For $j = 1, .., n-1$, the nodes of A_j of the path A are computed with $A_j = H(a_{j-1}||b_{j-1})$.

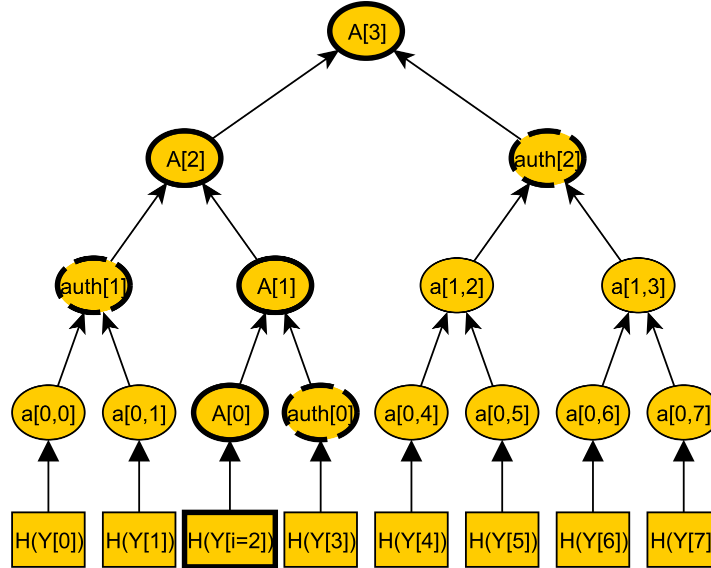

Figure 3.2: Merkle tree with path A and authentication path for i=2

If A_n equals the public key *pub* of the merkle signature scheme, the signature is valid.

3.4 Cost analysis

The big advantage of the Merkle Signature Scheme is, that many signatures can be generated with using only one public key. However, this advantage comes with an increase of computation time and signature length. In the following we will examine the computation time of each part of the signature process. To generate the public key *pub*, 2^n one-time signature keys must be generated. Then every node of the hash tree must be computed. The tree consists of $2^{n+1}-1$ nodes. One hash operation is needed to calculate a node, so that $2^{n+1}-1$ hash operations are needed to generate the public key. It is obvious, that the size of such a tree is limited. To compute 2^{40} nodes is very costly, to compute 2^{80} nodes is impossible.

To generate a signature the nodes $auth_0, ..., auth_{n-1}$ are needed. If you do not store the nodes of the tree, the nodes must be generated again for every signature. Generating the tree is very expensive, so that generating the entire tree for every signature is impracticable for bigger trees. But saving all $2^{n+1}-1$ nodes would result in huge storage requirements. Hence, a good strategy is needed, to generate the signature without saving too many nodes, at a still efficient time. This problem is called The Merkle tree traversal problem and is described in chapter 4.

The verification time is quite fast, compared to the signature time. At first,

the one-time signature must be verified. After that, the path $A = A_1, ..., A_n$ must be computed. To do this, only n hash operations are needed, one for every node.

The signature of the Merkle Signature Scheme consists of the one-time signature sig' and n nodes $auth_0, ..., auth_{n-1}$. If a 160 bit hash function is used, the signature size would be $|sig| = |sig'| + n * 160$ bits.

4 The Merkle tree traversal problem

In this chapter, we will focus on the problem, of efficiently computing the next authentication path, needed for the Merkle Signature Scheme. At first, we will introduce the treehash algorithm, to efficiently compute a node in a hash tree. This algorithm will be used later, to generate the public key and to generate the next authentication path.

4.1 The treehash algorithm

For the traversal techniques, we need an algorithm, that computes efficiently the nodes of the tree. Assume a binary tree with 2^n leafs. The height H of a node, is defined by the distance of the node to a leaf. So the root has the height $H = n$, while the leafs have the height $H = 0$. We define the node $a_{i,j}$ as the jth node from the left (starting with $j = 0$) of the height i. So $a_{0,0}$ is the leftmost leaf of the tree, and $a_{n,0}$ the root. To compute a node of the hight $H = h$, $2^h - 1$ nodes must be computed. The treehash algorithm needs $2^h - 1$ operations, to calculate a node of the hight h, while saving as few nodes at once as possible.

The main idea of the treehash algorithm is to calculate the needed subtree from left to right and only saving the nodes, that are still needed. This is done, by using a stack. At first the stack only consists of the leftmost leaf. Then the next leaf is added. The algorithm now checks whether the last two nodes on the stack are of the same height or not. If they are of the same height, the two nodes are removed from the stack, and their parent is built and pushed on the stack. If the last two nodes on the stack are of different height, then a new leaf is pushed on the stack. This step is repeated, until the node of the wanted height has been generated.

Algorithm: TREEHASH (start, maxheight)

1. Set $leaf = start$ and create empty stack.
2. Consolidate: If top 2 nodes on the stack are equal height:
   - Pop node value $P(n_{right})$ from stack.
   - Pop node value $P(n_{left})$ from stack.

- Compute $P(n_{parent}) = f(P(n_{left}||P(n_{right}))$.
- If height of $P(n_{parent}) = maxheight$, output $P(n_{parent})$.
- Push $P(n_{parent})$ onto the stack.

3. New Leaf: Otherwise:
   - Compute $P(n_l) = LEAFCALC(leaf)$.
   - Push $P(n_l)$ onto the stack.
   - Increment $leaf$.
4. Loop to step 2.

To be able to run multiple instances of treehash, we define an object $stack_h$ with two methods, $stack_h.initialize(startnode, h)$ and $stack_h.update(t)$. With the *initialize* method we simply define the start leaf and the height of the resulting node. The method *update* runs the steps 2 or 3 of the treehash algorithm t times. For example $stack_2.initialize(0, 2)$ means, that in $stack_2$ we compute nodes up to the height $h = 2$, beginning with the 0th node. $stack_2.update(3)$ will now perform 3 operations of treehash on $stack_2$. The first operation will be to push node $a_{0,0}$ on the stack. The second operation will be to push the node $a_{0,1}$ on the stack. Now the last two nodes on the stack are of equal height. So in the third operation these two nodes are removed and $a_{1,0}$ gets computed and push on the stack. Because the treehash should only perform three operations, the algorithm stops at this point. When $stack_2.update(t)$ is called again, the algorithm will continue at this point, by pushing the node $a_{0,2}$ on the stack.

4.2 The classic traversal

In the first step of the Merkle Signature Scheme, the public key, which is the root of the tree, gets computed. This is done, by using the treehash algorithm. During this computation, every node of the tree is generated, so that we can easily save the first authentication path $auth$. We do this, by saving all nodes $auth_i$ with $auth_i = a_{i,1}$ for $i = 1, ..., n-1$. These nodes $auth = \{auth_1, ..., auth_{n-1}\}$ are the right brothers of the nodes of the leftmost path. In addition to the $auth_i$ nodes we also store the nodes of the leftmost path in the objects $stack_i$, with $stack_i = a_{i,0}$ for $i = 1, ..., n-1$. We will need these objects, to efficiently generate the next authentication path.

The next phase is the output and update phase. In this phase, we output the leaf value together with the authentication path. After that, we generate the next authentication path. Generating the output is quite simple. We use the function $LEAFCALC$ to calculate the value of the leaf (The leaf values is the hash value of the public key of the one-time signature. So $LEAFCALC$ builds

the hash value of the one-time signature public key). The authentication path $auth = auth_1, ..., auth_{n-1}$ has been already computed. So the important part is to calculate the next authentication path. To do this, we need a counter $leaf$, which points to the current leaf to be calculated, and we need the old authentication path $auth$. In addition to that, we also have the objects $stack_i$ for $i = 0, ..., n-1$. We can modify these by the functions $stack_i.initialize(startnode, h)$ and $stack_i.update(t)$, as described above.

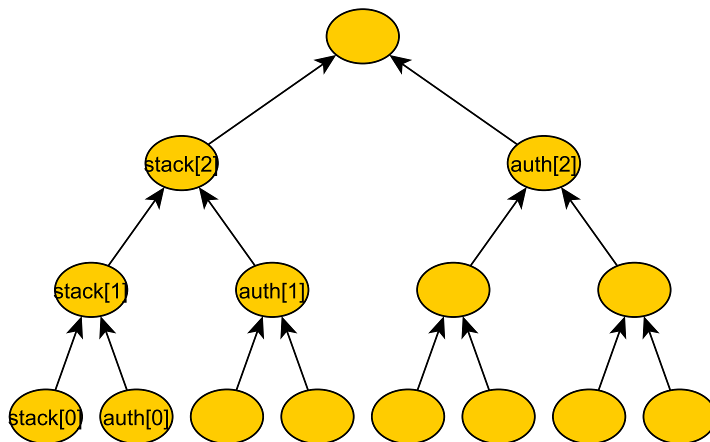

Figure 4.1: Merkle Tree before the first output and update phase

We now have to determine which authentication nodes $auth_h$ have to be changed, so that $auth = auth_o, ..., auth_{H-1}$ is the authentication path for the next leaf $leaf + 1$. The authentication node of the height h only needs an update, if 2^h divides $leaf + 1$ without remainder. The new authentication node $auth_h$ has already been generated and is saved in the stack $stack_h$. So if 2^h divides $leaf + 1$, $auth_h = POP(stack_h)$. Then $stack_h$ is empty and we use this stack to precalculate the next authentication node. In 2^h steps, when $leaf = leaf + 1 + 2^h$, $auth_h$ needs an update again. So we search for the leftmost leaf, $startnode$, of the next authentication node, of the height h. This is $startnode = leaf + 1 + 2^h + 2^h$ if the current $auth_h$ is a left-node and $startnode = leaf + 1$ if the current $auth_h$ is a right-node. So $startnode = leaf + 1 + 2^h \oplus 2^h$. Hence we set $stack_h.initialize(startnode, h)$.

This illustrated in figure 4.3. In this figure $auth_1$ has changed. The next change of $auth_1$ will be when $leaf = leaf + 1 + 2^1$. Hence, we need the authentication node of level 1 for the $leaf + 1 + 2^1$. This node is $sack_1$. The leftmost leaf of this node $stack_1$ is $leaf + 1 + 2^1 + 2^1 = startnode$. SO $stack_1.initialize(leaf + 1 + 2^1 + 2^1, 1)$.

We could now use the treehash algorithm to compute $stack_h$ at once. But this would take $2^{h+1} - 1$ steps. In the worst case, $H - 1$ nodes $auth_h$ can change at once, so that we would need $\sum_{h=0}^{H-1} 2^{h+1} - 1$ operations to compute one signature.

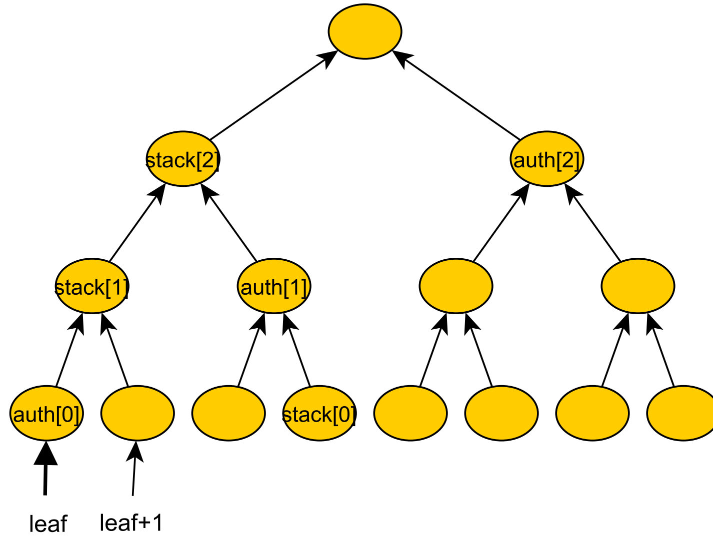

Figure 4.2: Merkle Tree after the first output and update phase

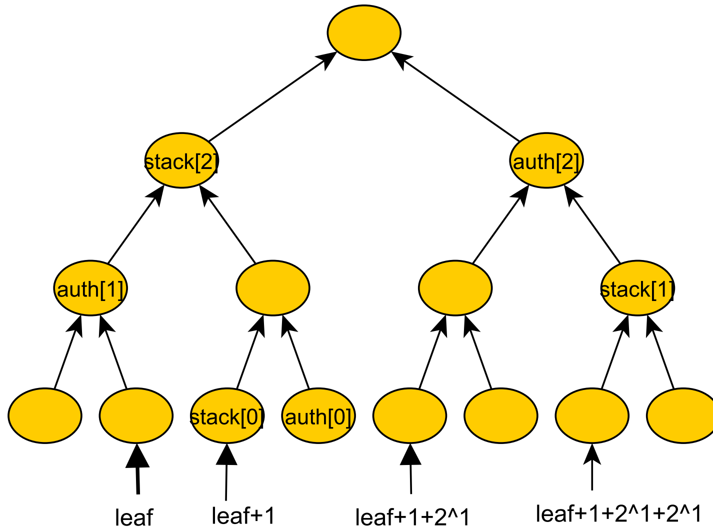

Figure 4.3: Merkle Tree after the second output and update phase

We know, that we do not need to change $auth_h$ for the next 2^h signatures. Hence, we have 2^h signatures time, to make the $2^{h+1}-1$ operations which generate the next node. Therefore, we only do two operations of updating for $h = 0, ..., H-1$ per signature, by calling $stack_h.update(2)$ for $h = 0, ..., H-1$. In this way, we only perform $(H-1)*2$ operations per signature in the worst case.

Algorithm: Classic Merkle Tree Traversal

1. Set $leaf = 0$.
2. **Output:**
   - Compute and output $leaf$ with $LEAFCALC(leaf)$
   - For each $h \in [0, H-1]$ output $\{auth_h\}$.
3. **Refresh Auth Nodes:**

   For h such that 2^h divides $leaf + 1$:
   - Set $auth_h$ be the sole node value in $stack_h$.
   - Set $startnode = (leaf + 1 + 2^h) \oplus 2^h$.
   - $stack_h.initialize(startnode, h)$.
4. **Build Stacks:**
   For all $h \in [0, H-1]$:
   - $stack_h.update(2)$.
5. **Loop**
   - Set $leaf = leaf + 1$.
   - If $leaf < 2^H$ go to Step 2.

4.3 Merkle tree traversal in log space and time

In [Szy04] an improvement to the Classic Merkle Traversal Algorithm was introduced. The main goal of this improved algorithm is to reduce the memory requirements. In the classic algorithm up to H instances of *treehash* may be concurrently active, one for each height less than H. In one treehash, up to $h+1$ nodes must be stored at once. Hence, up to $\sum_{h=0}^{H-1} h+1 = \frac{H*(H+1)}{2}$ nodes must be stored during one signature generation. The main idea of the improved algorithm is, to reduce the memory requirements, by reducing the number of active treehash instances during the signature generation.

To generate the next authentication node in $stack_h$, $2^{h+1} - 1$ operations are needed. To generate the next authentication node in $stack_{h+1}$, $2^{h+2} - 1$ operations are needed. In the classic algorithm, during one signature generation, $stack_h.update(2)$ and $stack_{h+1}.update(2)$ are called once. But if for the first $2^h/2$ signatures only $stack_h.update(4)$ is called, then $stack_h$ will be computed after $2^h/2$ signatures. For the next $2^h/2$ signatures, $stack_{h+1}.update(4)$ can be called, so that at the end, after 2^h signatures, the same values have been computed, as in the classic algorithm. However, in the first $2^h/2$ signatures $stack_{h+1}$ was empty and in the next $2^h/2$ signatures only one node was stored in the $stack_h$. Hence, we could reduce the memory requirements. The Logarithmic Merkle Tree Traversal algorithm below is based on this idea.

Algorithm: Logarithmic Merkle Tree Traversal

1. Set $leaf = 0$.
2. **Output:**
   - Compute and output $leaf$ with $LEAFCALC(leaf)$
   - For each $h \in [0, H-1]$ output $\{auth_h\}$.
3. **Refresh Auth Nodes:**
   For h such that 2^h divides $leaf + 1$:
   - Set $auth_h$ be the sole node value in $stack_h$.
   - Set $startnode = (leaf + 1 + 2^h) \oplus 2^h$.
   - $stack_h.initialize(startnode, h)$.
4. **Build Stacks:**
   Repeat the following $2H - 1$ times:
   - Let l_{min} be the minimum of $\{stack_h.low\}$ for all $h = 0, ..., H-1$.
   - Let focus be the least h so that $stack_h.low = l_{min}$.
   - $Stack_{focus}.update(1)$.
5. **Loop**
   - Set $leaf = leaf + 1$.
   - If $leaf < 2^H$ go to Step 2.

The algorithm prefers to complete $stack_h$ for the lowest h first, unless another stack has a lower tail node. Hence, the algorithm will not start to compute a new $stack_h$, until there is no node stored in any stack, that has a height below h.

The presented algorithm just stores $3log(N)$ nodes in the worst case and needs to compute $2log(N)$ operations in the worst case. (With N being the number of available signatures, hence $N = 2^n$) In the preprint version of the paper [Szy03], Szydlo presented an algorithm, which is based on the same idea, but achieves this with only $log(N)$ operations and $3log(N)$ stored nodes. This is due to a better, but more complex, scheduling algorithm.

4.4 Fractal tree representation and traversal

The main idea of the fractal traversal [MJS03] is, to split up the merkle tree in subtrees and to save and compute these subtrees, instead of single nodes. To describe the algorithm, we need some notations. Each subtree t has the same height h. Each root of one subtree is signed with a signature of the superior subtree. There are $L = H/h$ levels of subtrees from the bottom to the top of the merkle tree. If a node $a_{i*h,j}$ is a leaf of subtree t_i, then i denotes the level of the subtree, with $1 \leq i \leq L$. We define the subtree $t_{i,j}$, as the jth subtree from the left of level i.

During the key generation, all subtrees $Exit_i$, which contain a node of the next authentication path, are stored. At the beginning, these subtrees $Exits_i$ are the leftmost subtrees $Exits_i = t_{i,0}$ of each level $i = 1, ..., L$. If $a_{i*h,j}$ is the root of subtree $Exits_i$, then the next subtree of level i, which will be needed, is the subtree $Desire_i$ with the root $a_{ih,j+1}$. These subtrees $Desire_i$ are also saved during the key generation.

The signature generation phase consists again of two phases. In the output phase, the signature and the authentication path is outputted. In the update phase, the subtrees with the next authentication node $Exist_i$ are set and the subtrees $Desire_i$ are computed. To generate the subtree $Desire_i$, we use a slightly modified treehash algorithm. In this treehash algorithm, we save all nodes of the hight smaller than ih, because these nodes are nodes of the subtree. We also stop one step earlier, because we do not need to calculate the root (Instead of the root, the leaf of the subtree of level $i + 1$ is used as an authentication node).

Algorithm: Stratified Merkle Tree Traversal

1. Set $leaf = 0$.

2. **Output:**

   - Compute and output $leaf$ with $LEAFCALC(leaf)$
   - For each $j \in [0, H - 1]$ output $\{auth_j\}$.

3. **Next Subtree:**
For each i for which $Exist_i$ is no longer needed, i.e., for $i \in \{1, 2, ..., L\}$ with $leaf = 1(mod 2^{hi})$:
   - Set $Exist_i = Desire_i$.
   - Create new empty $Desire_i$ (if $leaf + 2^{ih} < 2^H$).
4. **Grow Subtrees**
For each $i \in \{1, 2, ..., h\}$: Grow tree $Desire_i$ by applying 2 units to modified treehash (unless $Desire_i$ is completed)
5. Increment $leaf$ and loop back to step 2 (while $leaf < 2^H$).

So the algorithm performs up to two operations for every subtree, in each round. There are $L = H/h$ subtrees. This gives an upper bound of the computation time of $sig_{time} = 2(L-1) \leq 2H/h$. The maximum space required during the computation is sig_{space}. There are L existing subtrees and $L-1$ desired subtrees. Each tree consists of $2^{h+1} - 1 - 1$ nodes, because the root value must not be stored. During the computation of a desired tree, up to $h*(i-1)$ tail nodes must be saved. So $sig_{space} \leq (2L-1)(2^{h+1}-2) + h(L-2)(L-1)/2$.

A good value for h, in which the space requirements are minimal, would be $h = logH = loglogN$.(See [FRACTAL]) Using this parameter would result in a time and space bound of $sig_{time} = 2logN/loglogN$ and $sig_{space} = 5/2log^2N/loglogN$.

5 Improvements to Merkle-Signature Scheme

In [JB06] and [JB07] improvements to the original Merkle Signature Scheme were proposed. The improvements consist of two main ideas. The first idea is, to use a pseudo random number generator (PRNG) with a seed value to generate the private keys of the one-time signatures. As a result, just the seed of the PRNG needs to be stored, instead of all private keys.

The other idea is, to use many smaller merkle trees instead of one big tree. One disadvantage of the Merkle Signature Scheme is the still limited number of signatures. Building a merkle tree for 2^{80} signatures is not possible, due to the enormous calculation costs to compute the public key. So instead of building one Merkle Tree with 2^{80} signatures, a tree $t_{1,0}$ with only 2^{20} leafs and another tree $t_{2,0}$ with 2^{20} signatures is build. The public key of $t_{2,0}$ is signed with a signature of $t_{1,0}$. Hence, $t_{1,0}$ is the parent tree of $t_{2,0}$. In the same way the trees $t_{3,0}$ and $t_{4,0}$ are build. The parent tree of $t_{3,0}$ is $t_{2,0}$ and the parent of $t_{4,0}$ is $t_{3,0}$. A message is signed with the merkle tree $t_{4,0}$. The public key of $t_{4,0}$ is signed by the tree $t_{3,0}$, which is signed by $t_{2,0}$. $t_{2,0}$ is signed by $t_{1,0}$. The public key of $t_{1,0}$ is the public key of this signature scheme. In this way 2^{20} messages can be signed with $t_{4,0}$. After these 2^{20} signatures all one-time keys of $t_{4,0}$ are used and a new tree $t_{4,1}$ is build. This tree is signed again by $t_{3,0}$. After 2^{20} different trees $t_{4,i}$ have been generated, a new tree $t_{3,1}$ is needed to sign $t_{4,2^{20}}$. This new tree is signed again by $t_{2,0}$. 2^{20} different trees $t_{3,i}$ can be signed by $t_{2,0}$. So that after 2^{20} different trees $t_{3,i}$ a new tree $t_{2,1}$ is build. 2^{20} trees $t_{2,i}$ can be signed by $t_{1,0}$, so that in this way $2^{20} * 2^{20} * 2^{20} * 2^{20} = 2^{80}$ signatures can be generated with only one public key (the one of t_1). But nevertheless only four trees of the size 2^{20} have been generated and stored at once. Hence, a in practice endless number of signatures can be generated with one public key, without making the signature cost impracticable. This principle is illustrated in figure 5.1.

In [JB07] it is shown, that using this strategy, it is possible to sign a message on a Pentium dualcore 1.8GHz in 10.1 ms and verify a signature in 10.1 ms. In the Merkle Signature Scheme many trade offs between signature size and speed can be made. The following table 5.1 shows the signature time and memory requirements for 4 different parameters. The table shows the memory requirements during the output phase mem_{output} and the update phase mem_{update}. The signature length is $mem_{signature}$. To generate the public key t_{keygen} minutes are needed. To sign a message t_{sign} ms are needed and to verify a signature t_{verify} ms are needed. For

$signatures$	mem_{update}	mem_{output}	$mem_{signature}$	t_{keygen}	t_{sign}	t_{verify}
2^{40}	3160 bytes	1640 bytes	1860 bytes	723 min	26.0 ms	19.6 ms
2^{40}	3200 bytes	1680 bytes	2340 bytes	390 min	10.7 ms	10.7 ms
2^{80}	7320 bytes	4320 bytes	3620 bytes	1063 min	26.1 ms	18.1 ms
2^{80}	7500 bytes	4500 bytes	4240 bytes	592 min	10.1 ms	10.1 ms

Table 5.1: Timings and memory requirements

details on the used algorithms and parameters see [JB07].

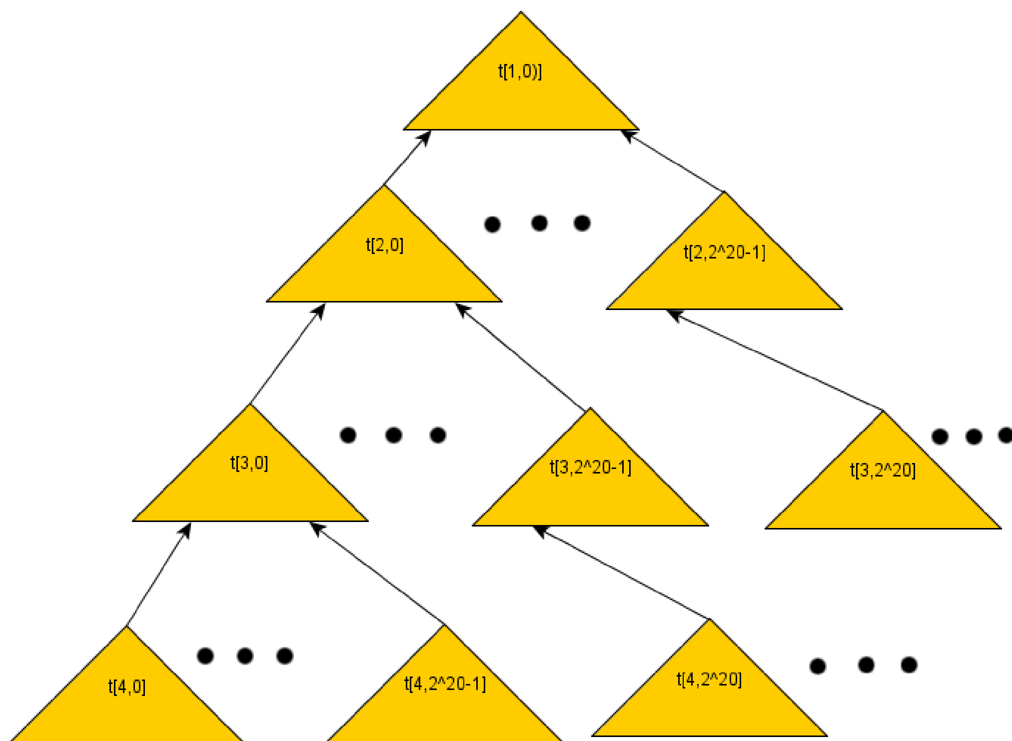

Figure 5.1: GMSS using four levels of trees

6 Cryptanalysis

In this chapter we will analyze the security of the Merkle Signature Scheme. To do this, we will consider what an attacker would have to do, to forge a signature. We assume that $sig = (sig'||auth_0||auth_1||...||auth_{n-1})$ is a valid signature of the message m, of the Merkle Signature Scheme, with the valid public key pub. sig' is a valid one-time signature and $auth_0, ..., auth_{n-1}$ the authentication path for the leaf $A_n = H(Y_i)$, with A_n being the hash value of the public key Y_i of the one-time signature. To verify the message M, at first the one-time signature sig' will be verified. If sig' is a valid signature, then A_0 gets computed with $A_0 = H(Y_i)$. Then the nodes $A_i = H(A_{i-1}||auth_{i-1})$ for $i = 1, ..., n$ are computed. If $A_n = pub$, then the signature is valid.

If an attacker, who knows message m and the signature sig, wants to counterfeit a signature of the message m', he has two options.

6.1 Case 1: $H(Y_i') = H(Y_i)$

The first option would be, that the attacker finds a valid one-time signature sig_a' with the public key Y_i' and $H(Y_i') = H(Y_i) = A_0$. An attacker could archive this, by finding a valid one-time signature of the message m' with the public key $Y_i' = Y_i$. Finding such a signature would mean to break the one-time signature. Therefore, if the attacker is able to break the one-time signature, he is able to break the Merkle Signature Scheme. If the attacker is not able to break the one-time signature, the attacker needs to find a signature sig_a' with the public key $Y_i' \neq Y_i$ and $H(Y_i') = H(Y_i)$. Hence, the attacker needs to find for a given input of a hash function, another input, which has the same hash value. A hash function in which this is possible is not second preimage resistant. Therefore, if the used hash function is second preimage resistant, the attacker will not be able to find such a one-time signature public key Y_i'.

So the Merkle Signature Scheme is secure in case 1, if the one-time signature is secure and the used hash function is second preimage resistant.

6.2 Case 2: $H(Y_i') \neq H(Y_i)$

The other option would be, to generate a valid one-time signature sig_a' with the public key Y_i' and $A_0' = H(Y_i') \neq H(Y_i) = A_0$. In this case, the attacker

needs to change the authentication path, so that $A'_n = A_n = pub$ with $A'_i = H(a'_{i-1}||auth'_{i-1})$ for $i = 1, ..., n$ to make the signature valid. If the attacker finds one $auth'_i$ so that $H(A'_i||auth'_i) = A'_{i+1} = A_{i+1}$ he has found a valid authentication path. Hence, to counterfeit a signature, the attacker needs to find one $auth'_i$ so that $H(A'_i||auth'_i) = H(A_i||auth_i)$. If the used hash function is not second preimage resistant, then such an attack is possible. But if the used hash function is second preimage resistant, an attacker will not be able to find such an $auth_i$.

So the Merkle Signature Scheme is secure, if the used one-time signature is secure and the used hash function is second preimage resistant. In chapter 2 we defined a cryptographic secure hash function, as a hash function, which is second preimage resistant and collision resistant. A hash function, which is collision resistant with a security of $O(2^{80})$, needs at least 2^{160} bits (see chapter 2). Note, that we do not need a collision resistant hash function for the Merkle Signature Scheme. The used hash function just needs to be second preimage resistant. Hence, it is possible to use a 80 bit hash function which is preimage resistant, to archive a security of $O(2^{80})$.

6.3 Differential Side Channel resistant

The Merkle Signature Scheme has an interesting characteristic. It is resistant against differential side channel attacks. In a differential side channel attack, the attacker gains extra information by eavesdropping a side channel during the computation of the signature. Classical side channels are the power consumption, the time the algorithm needed, or electromagnetic leaks. The attacker collects these information for many different signatures with the same public key. The goal is to gain extra information of the secret by comparing these information.

However, this strategy will not be successful against the Merkle Signature Scheme. The secret of the Merkle Signature Scheme are the private keys of the one-time signatures. But for each signature a new private key is used. Hence, an attacker can not gather information about a secret by comparing the informations gathered during the computation of two signatures, because the secrets have no relation to each other. Everything else of the Merkle Signature Scheme is public. All nodes of the tree can be published, because they all will be part of at least one signature and therefore will be public anyways.

7 Conclusion

The Merkle Signature Scheme is known for 30 years, but most improvements to the Merkle Signature Scheme like [Szy04], [JB06], [JB07], and [MJS03] were published within the past five years. These improvements begin to make the Merkle Signature Scheme a reasonable alternative to conventional signature schemes. As seen in chapter 5, it is possible to sign and verify a message with the Merkle Signature Scheme in a reasonable time, with a public key that can be used for 2^{80} signatures. Unfortunately, the signature size and the storage requirements are still very big compared to other digital signature schemes such as DSA or the RSA Signature Scheme. Hence, if necessary, conventional signature schemes can be replaced by the Merkle Signature Scheme in applications which have enough storage available and in which the big size of the signature does not matter.

The big advantage of the Merkle Signature Scheme is, that the security does not rely on the difficulty of any mathematic problem. The security of the Merkle Signature Scheme depends on the availability of a secure hash function and a secure one-time digital signature. Even if a one-time signature or a hash function becomes insecure, it can be easily exchanged. This makes it very likely that the Merkle Signature Scheme stays secure even if the conventional signature schemes become insecure.

Bibliography

[AMV96] P. Van Oorschot, A. Menezes, S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[Ede07] Boris Ederov. Merkle tree traversal techniques. *Bachelor Thesis, Darmstadt University of Technology Department of Computer Science Cryptography and Computer Algebra*, 2007.

[JB06] E. Dahmen, M. Dring, E. Klintsevich, J. Buchmann, L.C. Coronado Garca. CMSS - an improved merkle signature scheme. *Progress in Cryptology - Indocrypt 2006*, 2006.

[JB07] E. Klintsevich, K. Okeya, C.Vuillaume, J. Buchmann, E.Dahmen. Merkle signatures with virtually unlimited signature capacity. *5th International Conference on Applied Cryptography and Network Security - ACNS07*, 2007.

[Mer79] Ralph Merkle. Secrecy, authentication and public key systems/ A certified digital signature. *Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University*, 1979.

[MJS03] S. Micali, M. Jakobsson, T. Leighton, M. Szydlo. Fractal merkle tree representation and traversal. *RSA-CT 03*, 2003.

[Szy03] Michael Szydlo. Merkle tree traversal in log space and time. *(preprint version, 2003)*, 2003.

[Szy04] Michael Szydlo. Merkle tree traversal in log space and time. *Eurocrypt 2004*, 2004.