Further Analysis of a Proposed Hash-Based Signature Standard

Scott Fluhrer

Cisco Systems, USA
sfluhrer@cisco.com

Abstract. We analyze the concrete security of a hash-based signature scheme described in the most recent Internet Draft by McGrew, Fluhrer and Curcio. We perform this analysis in the random-oracle model, where the Merkle-Damgård hash compression function is models as the random oracle. We show that, even with a large number of different keys the attacker can choose from, and a huge computational budget, the attacker succeeds in creating a forgery with negligible probability ($< 2^{-129}$).

1 Introduction

McGrew et al have recently proposed a series of standards for hash-based signatures. Along with these standards, there is a proof by Jonathan Katz of security [1]; this proof shows that forgery attempts against version 4 [2] will succeed with only negligible probability. McGrew et al has since updated their proposal with version 6 [3], the proof is also applicable there. This proof is in the random oracle model, specifically, it assumed that the hash function was a random function, with the attacker having oracle access.

With this assumption, the proof is valid. However, if we examine a stronger attack model, specifically, if the attacker has oracle access to the hash compression function within the Merkle-Damgård hash function, we find that the proof no longer applies. Specifically, the conclusion in [1, Lemma 9] would be incorrect in this attack model, as the attacker can attack all the `0x04` hashes from a single Merkle tree at once, as follows. The attacker extracts the hash computations from all the interior Merkle tree nodes (which can be extracted from the authentication paths contained within the valid sigantures), and arranges these hash computations as follows:

$$
\begin{array}{llll}
\text{H(I,} & \text{T}[2*1], & \text{T}[2*1+1], & \text{ustr32}(1), \quad \text{0x04)} \\
\text{H(I,} & \text{T}[2*2], & \text{T}[2*2+1], & \text{ustr32}(2), \quad \text{0x04)} \\
\text{H(I,} & \text{T}[2*3], & \text{T}[2*3+1], & \text{ustr32}(3), \quad \text{0x04)} \\
 & \multicolumn{3}{c}{\ldots} \\
\multicolumn{4}{l}{\text{H(I, T}[2*(2^h-2)], \text{T}[2*(2^h-2)+1], \text{ustr32}(2^h-2), \text{0x04)}} \\
\multicolumn{4}{l}{\text{H(I, T}[2*(2^h-1)], \text{T}[2*(2^h-1)+1], \text{ustr32}(2^h-1), \text{0x04)}}
\end{array}
$$

where `I` is the identifier of the LMS public key used for the Merkle tree, and `T[2*i], T[2*i+1]` are the child nodes to interior node `i`.

Using oracle access to the hash compression function, the attacker then computes the intermediate state after the partial hash computation of `I || T[2*r] || T[2*r+1]` (indicated by the dashed line in the table) for every r within the tree. Note that this partial hash computation is on a SHA-256 block boundary, and hence the intermediate state consists solely of the internal 256 bits of internal state variables. Then, the attacker selects q strings z for length 64 (distinct from `T[2*i] || T[2*i+1]` for any i), and performs q queries of the intermediate state of `H( I || z || ...)`; if the attacker finds an intermediate state that matches any of the Merkle tree intermediate state in the above list (say, for node r), then he attacker has found $\texttt{Coll}_r$, as

```
H(I || T[2*r] || T[2*r+1] || u32str(r) || 0x04) =
          H(I || z || u32str(r) || 0x04)
```

For a height h Merkle tree, this succeeds with probability $(2^h - 1)q \cdot 2^{-8n}$,which for $h > 2$ is higher than the probability $3q \cdot 2^{-8n}$ proven assuming only hash function oracle access. This is not a violation of the original proof, as that proof assumed that the attacker had only oracle access to the hash function, and hence could not view intermediate hash states; it does indicate that, with SHA-256 at least, the attack model assumed by the proof might not the most appropriate.

Because of this observation, McGrew et al has revised their draft to version 7 [4]; this paper seeks to prove that this revised version is secure in this stronger attack model.

1.1 Assumptions of the Merkle-Damgård hash function

For the purposes of this proof, we will assume that the hash function H used is a Merkle-Damgård hash function, based on a hash compression function $\mathfrak{C}$ that we will treat as a random oracle.

As a review of the construction of a Merkle-Damgård hash function, it is based on a collision resistant compression function $\mathfrak{C} : \{0,1\}^s \times \{0,1\}^b \rightarrow \{0,1\}^s$ (where b is the block size of the input message, and s is the size of the internal state), and a padding function $\text{Pad}(M)$, such that $M||\text{Pad}(M)$ is always a multiple of b in length. To hash an n bit message M, we append $\text{Pad}(M)$ and then divide the padded message $M||\text{Pad}(M)$ into successive b bit blocks $M_0, M_1, ..., M_k$. Then, we start with a fixed s-bit state $S_0 = IV$, and then successively compute $S_{i+1} = \mathfrak{C}(S_i, M_i)$, and the final state S_{k+1} is the hash result[1]

We assume that the padding function we use has the property that $M||\text{Pad}(M) \neq M'||\text{Pad}(M')$ if $M \neq M'$. For this paper, we further assume that if M, M' have the same length, then the length of $\text{Pad}(M), \text{Pad}(M')$ will also be the same.

The padding function used in common Merkle-Damgård hash functions (such as SHA-256) meet these criteria.

[1] It is common for Merkle-Damgård hash functions to truncate the output state when generating the final hash output. For the purposes of this paper, we will assume that the hash function does not truncate the output; the hash function used in the draft (SHA-256) does not.

2 Version 07 of the McGrew-Fluhrer-Curcio draft

In analyzing the most recent version of the McGrew-Fluhrer-Curcio proposal, we begin by showing the security of a hash compression function in an abstract attack model. We will then show that the LMS signature scheme can be proven to be secure, assuming the security results of the abstract attack model.

2.1 Abstract hash model

In this model, we give the attacker a series of strings with prefixes and hash targets; the attacker's goal is to find another string that has both the same prefix and hash target as any of the strings.

We will denote pre^i as the values used as the string prefixes, and H^i as the hash targets. We assume that, for any value pre, there are at most k prefixes $pre^i = pre$ (where k is a security parameter). In addition, we assume that no prefix is a proper prefix of any other, that is $pre^i \neq pre^j||S$, for any i, j, and any nonempty string S. We assume that these prefixes has been chosen as a part of the model set up procedure[2].

We are interested in bounding the attacker's success probability in the following expirement.

1. We create a random oracle $\mathfrak{C} : \{0,1\}^s \times \{0,1\}^b \rightarrow \{0,1\}^s$, a value IV and a padding function Pad that meets the requirements listed above. We create a Merkle-Damgård hash function H (with s bits of internal state and a block size of b) based on $\mathfrak{C}$, Pad and IV.
2. For $i = 1, ..., N$, we generate the prefixes pre^i arbitrarily, with the constraints that no prefix value is used more than k times, no prefix is a proper prefix of another, no prefix is used for two different groups, and that each prefix is no more than b bits in length[3]. We give these prefixes to the attacker.
3. We divide the prefixes into three groups (and we assume that no two prefixes from different groups share a common value):
   (a) For a Group 1 prefix pre^i, we choose uniform $S^i \in \{0,1\}^s$, and uniform $H^i \in \{0,1\}^s$, add the tuple $(IV, pre^i||S^i||\text{Pad}(pre^i||S^i), H^i)$ to $\mathfrak{C}$, and give H^i (but not S^i) to the attacker. For these prefixes, we assume that the length of $pre^i||S^i||\text{Pad}(pre^i||S^i)$ is precisely b bits.
   (b) For a Group 2 prefix pre^i, we select an arbitrary string S^i, evaluate $H^i = H(pre^i||S^i)$, and give S^i and H^i to the attacker
   (c) For a Group 3 prefix pre^i, the attacker gives us a string S^i; we choose a uniform $C^i \in \{0,1\}^s$, evaluate $H^i = H(pre^i||C^i||S^i)$, and give C^i and H^i to the attacker. For these prefixes, we require that the length of pre^i be at most $b - s$[4].

[2] These indentifier could be chosen adaptively by the attacker (subject to the requirements listed above) without any significant change ot the proof in the following section, for simplicity, we treat them as fixed in advance. When we use this proof for LMS, the identifiers will be fixed in advance.

[3] The prefixes used as Group 1 and Group 3 will have stricter length requirements

[4] So that $pre^i||C^i$ fit in the initial hash compression block

4. The attacker is given the padding function Pad, the value IV and oracle access to $\mathfrak{C}$, bounded to m invocations; note that this allows the attacker to compute $H(S)$ for any string S (by making the appropriate queries on $\mathfrak{C}$).
5. We consider the attacker to be successful if any of these conditions hold:
   (a) For a Group 1 prefix pre^i, the attacker is able to produce a string S' of length s with $H^i = H(pre^i||S')$
   (b) For a Group 2 prefix pre^i, the attacker is able to produce a string $S' \neq S^i$ with $H^i = H(pre^i||S')$
   (c) For a Group 3 prefix pre^i, the attacker is able to produce a string $S' \neq C^i||S^i$ with $H^i = H(pre^i||S')$

where, in the above procedure, where we evaluate $H(m)$, we mean:

1. Append pad(m) to m, and parse the padded message into b-sized blocks $m_1, m_2, ..., m_n$
2. Set s to IV
3. For $i = 1, ..., n$, if an entry (s, m_i, t) (for any t) is in $\mathfrak{C}$, then set $s = t$, otherwise choose uniform $t \in \{0,1\}^s$, and add the tuple (s, m_i, t) to $\mathfrak{C}$, and set $s = t$
4. Return the final value of s

In the below lemmas, we will introduce value λ it is an arbitrary integer; these lemmas will hold for all values of λ.

Lemma 1. *If the total number of hash compression operations done as a part of the setup process and the attacker Oracle queries is bounded by m, then the probability that we will have an entry $(*, *, IV) \in \mathfrak{C}$ is bounded by $m2^{-s}$.*

Proof. This follows from the fact that each entry in $\mathfrak{C}$ has a value which is selected uniformly, and so the probablity of a single trial being IV is 2^{-s}, hence the probability bound listed after m trials follows immediately.

Lemma 2. *If the total number of hash compression operations done as a part of the setup process and the attacker Oracle queries is bounded by m, then the probability that we will have at least λ pairs of entries in $\mathfrak{C}$ that collide[5] is bounded by $(m^2 2^{-s-1})^\lambda / \lambda!$.*

Proof. This follows from the fact that each entry in $\mathfrak{C}$ has a value which is selected uniformly, and the probability that there are λ distinct repeats out of m random values from of a range of 2^s is bounded by the probability given.

Lemma 3. *The attacker will get a* Group 1 *success with probability at most* $2km2^{-s}$

Proof. Let us denote the prefixes that share the prefix value pre as $pre^{a_1}, pre^{a_2}, ..., pre^{a_n}$ for $n \leq k$. Any $\mathfrak{C}$ query $(IV, pre||S||\text{Pad}(pre||S), -)$ will return a uniform value in $\{0,1\}^s$, unless the attacker happens one of the value $S \in \{S^{a_1}, S^{a_2}, ..., S^{a_n}\}$. Hence, any query of $\mathfrak{C}(IV, pre^i||S'||pad)$ will return one of the $H^{a_1}, H^{a_2}, ..., H^{a_n}$ values with probability bounded by the probability that

[5] that is, both $(x, y, c), (z, w, c) \in \mathfrak{C}$, with $(x, y) \neq (z, w)$

$S' = S^{a_j}$ (probability at most $k2^{-s}$), plus the probability that the uniform value selected will happen to be H^{a_i} (probability bounded by $k2^{-s}$). Any other query will yield no information about a Group 1 success; hence the probability after m queries is bounded by $2km2^{-s}$.

Lemma 4. *Assuming that the conditions in Lemma 1, 2 and 3 are not met, the attacker will get a* Group 2 *success with probability bounded by* $\lambda km2^{-s}$

Proof. If the attacker does produce a string S' with $H^i = H(pre^i||S')$; we will assume that the attacker has issued all the $\mathfrak{C}$ oracle queries involved with computing this hash.

Let us consider the padded message blocks

$$(M_0, M_1, ..., M_n) = (pre^i||S^i||\text{Pad}(pre^i||S^i))$$

$$(M'_0, M'_1, ..., M'_n) = (pre^i||S'^i||\text{Pad}(pre^i||S'^i))$$

, and the sequence of states

$$S_0 = IV, S_1 = \mathfrak{C}(S_0, M_0), ..., S_n = \mathfrak{C}(S_{n-1}, M_{n'-1})$$

$$S'_0 = IV, S'_1 = \mathfrak{C}(S'_0, M'_0), ..., S'_{n'} = \mathfrak{C}(S'_{n'-1}, M'_{n'-1})$$

and the smallest j such that $(S_{n-j}, M_{n-j}) \neq (S'_{n'-j}, M'_{n'-j})$. We know such a $j \leq n, n'$ just exist, because:

- We know that we cannot have either $S_x = IV$ or $S'_x = IV$ for $x > 0$ (by the assumption of Lemma 1), and hence one sequence cannot be a proper prefix of the other
- We know that the sequences cannot be identical, as if they were, then we have $S^i = S'$, contrary to the assumption.

Let us consider the set of sequences in $\mathfrak{C}$ such that $S"_0 = IV, S"_1 = \mathfrak{C}(S"_0, M"_0), ..., S"_{n"} = \mathfrak{C}(S"_{n"-1}, M"_{n"-1}), S"_{n"} = S'_{n'-j}$; namely $S'_k = S"_k, j = n"$.

Because there are at most $\lambda - 1$ collisions, that is, places where two chains potentially merge, that mean that there are at most λ starting places (with the initial state being IV) that such a sequence can start. In addition, each such sequence can start with a specific prefix value, for each such sequence, there are at most k such prefixes with that value.

Hence, there are at most λk prefixes that have a resulting S_i value that can make this chain work. So, the probability that the attacker will find a (s, b) value that makes this work, after m queries, is bounded by $\lambda km2^{-s}$

Lemma 5. *Assuming the conditions in Lemma 1 and Lemma 2 are not met, the attacker will get a* Group 3 *success with probability at most* $\lambda(k+1)m2^{-s}$

Proof. Before the attacker submits the value S^i, he has a probability bounded by $km2^{-s}$ of querying the value $\mathfrak{C}(IV, \text{Initial}(pre^i||C^i||S^i))$, where Initial$(M)$ is the first b block of the string $M||\text{Pad}(M)$. If he has not queries such a value, then the value selected for the state $\mathfrak{C}(IV, \text{Initial}(pre^i||C^i||S^i))$ when we evaluate H^i

will be random. Hence, in this case, the success probability is bounded by the sum of the two probabilities, namely $\lambda(k+1)m2^{-s}$.

Theorem 1. *The probability that the attacker would succeed in this game is bounded by the probability* $m(2k+1)(\lambda+1)2^{-s} + (m^2 2^{-s-1})^{\lambda}/\lambda!$.

Proof. The attacker wins if he succeeds against either a Group 1, Group 2 or Group 3 prefix. The conditional probabilities of the attacker succeeding are bounded by by expressions given in lemmas 3, 4, 5, conditional that the assumptions of lemma 1 and lemma 2 hold. The probabilities that the assumptions given by lemma 1 or llemma 2 do not hold are bounded by the expression given in those lemmas. Hence, the probabilility that the attacker will succeed is bounded by the sum of the probabilities given in the five lemmas, which is the probability given.

2.2 Description of the LMS scheme

LMS is a two level signature scheme, where a one time signature (LM-OTS) is used to sign the message, while a Merkle tree signs the LM-OTS public key.

2.2.1 Description of LM-OTS We begin with a detailed description of the LM-OTS scheme, following [4]. Let $H : \{0,1\}^* \to \{0,1\}^s$. Fix $w \in \{1,2,4,8\}$ as a parameter of the scheme, and set $e \stackrel{\text{def}}{=} 2^w - 1$. Set $u \stackrel{\text{def}}{=} s/w$; note that the output of H can be a sequence of u integers, each w bits long. Set $v \stackrel{\text{def}}{=} \lceil \lfloor \log u \cdot e + 1 \rfloor / w \rceil$ and $p \stackrel{\text{def}}{=} u + v$. Define a function $\texttt{checksum} : (\{0,1\}^w)^u \to \{0,1\}^{wv}$ as follows:

$$\texttt{checksum}(d_0, ..., d_{u-1}) \stackrel{\text{def}}{=} \sum_{i=0}^{u-1} (e - d_i)$$

where each $d_i \in \{0,1\}^w$ is viewed as an integer in the range $\{0, ..., 2^w - 1\}$ and the result is expressed as an integer using exactly wv bits.[6] For positive integers i, b with $i < 2^{8b}$, we let $[i]_b$ denote the b-byte representation of i in bigendian order. For a string s and postive integer j, set $H^0_{I,q}(x; j) \stackrel{\text{def}}{=} x$. For positive integers $i \geq 1$ and j, define

$$H^i_{I,q,d}(x; j) \stackrel{\text{def}}{=} H(I, [q]_4, [d]_2, [i + j - 1]_1, H^{i-1}_{I,q,d}(x; j))$$

Define the LM-OTS scheme as follows:

Key-generation algorithm `GenOTS`
Key Generation takes as input I, q, where I is a 16 byte identifier, and q is a 4 byte *diversification factor*. The algorithm proceeds as follows[7]:

[6] In [4] the result is expressed as a 16-bit integer, but only the top vw bits are used.
[7] In the below descriptions, the values `8080` through `8383` listed are in hexadecimal.

1. Choose p uniform values $x_0, ..., x_{p-1} \in \{0,1\}^s$.
2. For $i = 0$ to $p-1$, compute $y_i = H^e_{I,q,i}(x_i; 0)$.
3. Compute $pk := H(I, [q]_4, [\texttt{8080}]_2, y_0, ..., y_{p-1})$

The public key is pk, and the private key is $sk = (x_0, ..., x_{p-1})$.

Signing algorithm `SignOTS`
Signing takes as input a private key $sk = (x_0, ..., x_{p-1})$ and a message $M \in \{0,1\}^*$ as usual, as well as I, q as above. It does:

1. Choose uniform $C \in \{0,1\}^s$.
2. Compute $Q := H(I, [q]_4, [\texttt{8181}]_2, C, M)$ and $c := \texttt{Checksum}(Q)$. Set $V := Q||c$, and parse V as a sequence of w-bit integers $V_0, ..., V_{p-1}$
3. For $i = 0, ..., p-1$, compute $\sigma_i := H^{V_i}_{I,q,i}(x_i; 0)$
4. Return the signature $\sigma = (C, q, \sigma_0, ..., \sigma_{p-1})$

Verification algorithm `VrfyOTS`
Verification takes as input a message $M \in \{0,1\}^*$ and a signature $(C, q, \sigma_0, ..., \sigma_{p-1})$, as well as I and q as above. It does:

1. Compute $Q := H(I, [q]_4, [\texttt{8181}]_2, C, M)$ and $c := \texttt{Checksum}(Q)$ Set $V := Q||c$, and parse V as a sequence of w-bit integers $V_0, ..., V_{p-1}$
2. For $i = 0, ..., p-1$, compute $\sigma_i := H^{V_i}_{I,q,i}(\sigma_i; V_i)$
3. Output $H(I, [q]_4, [(8080)]_2, y_0, ..., y_{p-1})$

We note that, in contrast to the usual convention, `VrfyOTS` returns a string rather than a bit and does not take a public key as input. A signature σ on some message $M$is valid relative to some fixed public key pk if the output of `VrfyOTS` is equal to pk.

One can verify that correctness holds in the following sense: for any I, q, and (sk, pk) output by $\texttt{GenOTS}(I, q)$, and any message M, we have:

$$\texttt{VrfyOTS}(\texttt{SignOTS}(sk, M, I, q), I) = pk$$

.

2.2.2 Description of LMS An instance of the LMS scheme is defined by computing a Merkle tree of height h using 2^h LM-OTS public keys at the leaves. We give a formal definition now.

Let $H : \{0,1\}^* \to \{0,1\}^s$ as previously. We fix $w \in \{1, 2, 4, 8\}$ for use in the LM-OTS system, and we also select an integer h.

Key-generation algorithm `Gen`
Key Generation takes as input a parameter h and a value $I \in \{0,1\}^{128}$. The algorithm proceeds as follows:

1. For $q = 0, ..., 2^h - 1$, compute $(pk^q, sk^q) \leftarrow \texttt{GenOTS}(I, q)$.
2. For $r = 2^h, ..., 2^{h+1} - 1$, set $T[r] := H(I, [r]_4, [8282]_2, pk^{r-2^h})$
3. For $r = 2^h - 1, ..., 1$, set $T[r] := H(I, [r]_4, [8383]_2, T[2r], T[2r+1])$

The public key is $pk = (h, I, T[1])$, and the private key is $sk = (sk^0, ..., sk^{2^h-1}, T[1..2^{h+1} - 1])$.

Signing algorithm `Sign`

Signing takes as input a private key $(sk^0, ..., sk^{2^h-1}, T[1..2^{h+1} - 1])$, an integer $0 \leq q < 2^h$, a message $M \in \{0,1\}^*$, as well as I as above. The algorithm proceeds as follows[8]:

1. Compute $\sigma := \texttt{SignOTS}(sk^q, M, I, q)$.
2. Set $p_0, ..., p_{h-1}$ as $p_i := T[\lfloor (q + 2^h)/2^i \rfloor \oplus 1]$
3. Return the signature $\Sigma = (\sigma, p_0, ..., p_{h-1})$

Verification algorithm `Vrfy`

Verification takes as input a public key (h, I, T), a message $M \in \{0,1\}^*$, and a signature $\Sigma = (\sigma, p_0, ..., p_{h-1})$ and an integer $0 \leq q < 2^h$. The algorithm proceeds as follows:

1. Compute $pk := \texttt{VrfyOTS}(M, \sigma)$.
2. Set $r := q + 2^h$, and compute $T[r] = H(I, [r]_4, [8282]_2, pk)$
3. For $i = 1..h$, set $r := \lfloor (q + 2^h)/2^i \rfloor$, and compute $T[r]$ as follows:
   $T[r] := H(I, [r]_4, [8383]_2, T[2r], p_{i-1})$ if $\lfloor q/2^{i-1} \rfloor$ is even
   $T[r] := H(I, [r]_4, [8383]_2, p_{i-1}, T[2r+1])$ if $\lfloor q/2^{i-1} \rfloor$ is odd
4. Return 1 if and only if $T[1] = T$.

This verification procedure works because, if the signature is valid, all T elements computed during the verification procedure match the corresponding T elements of the private key.

2.3 Security of the LMS scheme

We adapt the standard notion of security for signature schemes to the multi-user setting. In this setting, multiple independent instances of the scheme are run, generating multiple public keys (which are given to the attacker) and private keys. The attacker can ask for the signatures of arbitrary messages, selecting any of the public keys, and the indexes (given the constraint that they can ask for a signature for a specific (public key, index) pair once). In addition, the

[8] The standard description of this algorithm has q managed by the secret key to ensure that no q value is reused, and has the q value embedded in the signature. However, to make the proof clearer, we make q as an explicit input; this modification has no impact on the security.

attacker is given Oracle access to the hash compression function. The attacker is considered successful if it generates a signature forgery with respect to any of those instances.

In particular, suppose we have z instances of the LMS scheme, with identifiers $I^1, I^2, ..., I^z$, with the I values and the private keys assigned randomly[9]. We will assume that no I value appears more than k times in this list (we will consider the probability of this not being true below). Then, the expirement is:

1. We choose a random function $\mathfrak{C} : \{0,1\}^s \times \{0,1\}^b \rightarrow \{0,1\}^s$, a value IV and a padding function Pad that meets the requirements. We create a Merkle-Damgård hash function h (with s bits of internal state and a block size of b) based on $\mathfrak{C}$, IV and Pad
2. For $i = 1, ..., z$, the key-generation algorithm `Gen` is run using identifier I^i to obtain (pk^i, sk^i). The attacker is given $(I^1, pk^1), ..., (I^N, pk^N)$.
3. The attacker is given the padding function Pad, IV and oracle access to $\mathfrak{C}$ (bounded to m invocations), plus a signing oracle SignOracle$(\cdot, \cdot, \cdot)$ such that SignOracle(i, j, M) returns $\texttt{Sign}(sk^i, j, M, I^i)$. For each (i, j) pair, the attacker may make at most one query. Without loss of generality, we assume the attacker makes exactly one signing query Sign(M, pk^i) for each value of i, j.
4. The attacker outputs (i, j, M, Σ) with $M \neq M^{i,j}$. The attacker succeeds if Σ is a valid signature on M for the ith public key, the jth index, i.e. if $\texttt{Vrfy}(pk^i, M, \Sigma, j)$ returns 1.

Theorem 2. *The probability that the attacker would succeed in creating a forgery is bounded by* $m(2k+1)(\lambda+1)2^{-s} + (m^2 2^{-s-1})^\lambda / \lambda! + z^{k+1} 2^{-128k} / (k+1)!$ *(for all values* k, λ*).*

Proof. We can view this as a special case of our abstract hash model game.

– When we generate a public/private key pair:
  - We compute all the LM-OTS public/private key pairs (placing the entries used to compute the random key generate and the Winternitz values into $\mathfrak{C}$).
  - We then hash the final Winternitz values together, forming the prefix $I^i||q||\texttt{8181}$, use the final Winternitz values as the string S, and treat it as a Group 2 instance
  - We then hash the Merkle tree leaves; for leaf node r, we form the prefix $I^i||r||\texttt{8282}$, use the OTS public key as the string S, and treat it as a Group 2 instance
  - We then hash the Merkle tree nodes; for tree node r, we form the prefix $I^i||r||\texttt{8383}$, use the two child nodes as the string S, and treat is as a Group 2 instance

[9] For the purposes of this proof, we will assume that the LMS scheme uses the recommended pseudorandom key generation process given in [4, Appendix A]; it is easy to see how to modify the proof if the LM-OTS keys were generated randomly.

– When the attacker gives us a value to sign the message M with instance i, index q:
  - We form the prefix $I^i||q||\mathsf{8080}$, and treat it as a Group 3 instance along with the message M.
  - We compute the checksum of the hash, and compute the Winternitz digits.
  - When we generate the last Winternitz value for digit d, we form the prefix $I^i||q||d||v_{„}$ with v being a byte with the value $V_d - 1$ if $V_d > 0$ and a byte with the value 255 if $V_d = 0$ and treat it as a Group 1 instance[10].

We see that we meet all the requirements of the game, if we assume the SHA-256 hash function (so we have $b = 512$, $s = 256$ and use the SHA-256 padding function):

– With the exception of prefixes that use the same I value (and the rest of the prefix is the same), we do not repeat prefixes, and no prefix is a proper prefix of another
– When we do reuse the same I value, we never use it more than k times (as per the above assumption), and so any specific prefix is never used more than k times.
– The prefixes meet the length limits:
  - the Group 1 prefixes plus the length s C value come to 55 bytes in length; the SHA-256 padding function allows that to fit in one block
  - the Group 2 prefixes are less than 64 bytes in length
  - the Group 3 prefixes are less than $32 = b - s$ bytes in length

If the attacker generates a forgery, then the attacker was successful against the abstract hash model.

To see this, let us assume that we have a forgery for key pk^i, index q, message M' (using randomizer C'), which using the LMS verification procedure, yields pk^i.

We look up the message M that the attacker had submitted to be signed with key pk^i, index q (if there has been no such message, we can arbitrarily select a message distinct from M' and sign it).

Then, stepping downwards through the authentication path of both the valid message and the forgery:

– If they first differ within an internal node of the Merkle tree, that is, the hash computation of $I^i||r||\mathsf{8383}$ for the correct message and the forgery has different message texts, the attacker has won against the corresponding Group 2 instance.
– If they first differ within a leaf node hash of the Merkle tree, that is, the hash computation of $I^i||r||\mathsf{8282}$, the attacker has won against the corresponding Group 2 instance.

[10] This is the prefix for both the intermediate Winternitz computation and the suggested pseudorandom key generation process in Appendix A of the draft, hence this proof need not make a distinction

– If they first differ during the hash of the $I^i||q||\texttt{8181}$ hash of the Winternitz public key, the attacker has won against the corresponding Group 2 instance.
– If the computation $H(I^i||q||\texttt{8080}||C'||M')$ differs from $H(I^i||q||\texttt{8080}||C^i||M)$, then at least one of the digits of either of the hash or the forgery must be lesser; if that digit is d, then the attacker has won against the Group 1 instance of that digit (as that preimage is computable from the forgery signature).
– If those two computations are the same, then the attacker has won against the corresponding Group 3 instance.

We assumed above that we never generate a specific 128 bit I value for more than k keys; as I is selected randomly, this assumption is false with probability bounded by $z^{k+1}2^{-128k}/(k+1)!$; we add that probability to the probability that the attacker won against the abstract model.

Hence, the probability that an attacker, given z public keys, and given access a signature oracle (where no more that m hash computations were used total, including ones performed setting up the keys, and the ones responding to attacker queries), has a forgery probability no more than $m(2k+1)(\lambda+1)2^{-s} + (m^2 2^{-s-1})^\lambda/\lambda! + z^{k+1}2^{-128k}/(k+1)!$ (for any k, λ).

Corollary 1. *If we have no more than 2^{64} randomly chosen LMS private keys, allow the attacker access to a signing oracle and a SHA-256 hash compression oracle, and allow a maximum of 2^{120} hash compression computations, then the probability of an attacker being able to generate a single forgery against any of those LMS keys is less than 2^{-129}.*

Proof. Evaluating the equation of Theorem 1, with $s = 256$ (the size of the SHA-256 state), $z = 2^{64}$ (maximum number of private keys), and $m = 2^{120}$ (maximum number of hash compression evaluations), and $k = 3, \lambda = 7$ evaluates to $\approx 2^{-129.58} < 2^{-129}$.

3 Hierarchical Signature Scheme

In the Hierarchical Signature Scheme (HSS), described in [4, Section 6], a public key consists of an LMS public key plus an integer L. The signature consists of L-1 instances of a signed public key, which is an LMS public key along with the signature of that public key signed by previous public key. Finally, there is the signature of the message, signed with the last public key.

Theorem 3. *If the attacker is unable to create a forgery for the LMS public key system with nontrivial probability, he is unable to generate a forgery for the HSS system with nontrivial probability.*

Proof. To show this, we assume that the attacker can generate a forgery to the HSS system with nontrivial probability, and show how that implies a forgery to the LMS system with that same probability.

We follow the same backtracking logic; suppose someone had a forgery to an HSS public key. Then:

– For levels i := 0..L-2

    - We examine the signed message
    - If it differs from the signed public key from the valid signature at same index (with the curent public key), then the attacker has successfully generated a forgery against the current public key
    - If it is the same signed public key, we make that public key the current one and continue
- If we get to the bottom level, we examine the signed message
- If it differs from the message from the valid signature at the same index, then the attacker has successully generates a forgery
- If it is the same, then the attacker has signed precisely the same message; it's not a forgery

4 Future Work

We have analyzed the security of version 07 of the McGrew-Fluhrer-Curcio proposal when implemented using a Merkle-Damgård hash function whose output is not truncated (e.g. SHA-256). Future work may analyze the security of a truncated Merkle-Damgård hash function (e.g. SHA-512/256), or a sponge hash construction (e.g. SHA-3).

References

1. Jonathan Katz, *Analysis of a Proposed Hash-Based Signature Standard*, Security Standardization Research 2016, https://www.cs.umd.edu/j̃katz/papers/HashBasedSigs-04.pdf, Accessed 2016-11-14.
2. D. McGrew, Curcio, *Hash-Based Signatures (draft-mcgrew-hash-sigs-04)*, https://datatracker.ietf.org/archive/id/draft-mcgrew-hash-sigs-04.txt, Accessed 2017-05-30.
3. D. McGrew, Curcio, Fluhrer, *Hash-Based Signatures (draft-mcgrew-hash-sigs-06)*, https://datatracker.ietf.org/doc/draft-hash-sigs/06, Accessed 2017-05-30.
4. D. McGrew, Curcio, Fluhrer, *Hash-Based Signatures (draft-mcgrew-hash-sigs-07)*, https://datatracker.ietf.org/doc/draft-hash-sigs/07,