Merkle Signatures for Real-World Use

ABSTRACT
Hash-based signatures have seen little use in practice. We explore the possibility of using hash-based signatures as the signature mechanism used in certificates on the Web. After optimizing the parameters for these settings, in both a stateful and stateless setup, we experiment with an implementation of these schemes and report on their performance and resulting signature size. We also explore an approach to further improve these signatures.

1. INTRODUCTION
Digital signatures are widely used on the Internet: from public-key certificates to key exchange protocols and remote attestation. All signature schemes used in practice (e.g., RSA and ECDSA) are based on concrete algebraic assumptions. If one of these assumptions is violated, say due to the development of a quantum computer, then the corresponding signature scheme is irreparably damaged.

Another approach to digital signatures is based on one-way and collision-resistant hash functions [11, 13]. These schemes make no use of algebraic assumptions: the only primitive needed is a collision resistant hash function such as SHA-256. If at some point it is discovered the SHA-256 is not collision resistant one can repair the signature scheme by simply replacing SHA-256 by another collision resistant candidate. Moreover, since there is no generic fast quantum collision finding algorithm\(^1\) these signature schemes will not be crippled by the development of a quantum computer.

Hash-based signatures have seen little use in practice. The main reason is signature size. While the public-key is always short, simply one output of SHA-256, signature size is larger than signatures generated by algebraic schemes. We give a detailed comparison in Section 4.

In this paper we explore the use of hash-based signatures in certificates. To do so we adapt hash-based signatures for use by a large certificate authority. Since every certificate contains both a public-key and a signature our goal is to minimize the sum of the lengths of the public-key and a signature.

We do so while requiring that signature generation time is no more than what is required for RSA signatures of comparable security.

The most efficient hash-based signatures are stateful: the signer must keep a counter that is incremented every time a signature is generated and must never repeat. Under these constraints the total length of the public key and signature is about 5KB for a system supporting \(2^{64}\) signatures. In comparison, RSA signatures are only 0.25KB.

Implementation. We built a complete implementation of this optimized hash-based signature scheme and explain how to use it. One difficulty with stateful signatures is a settings where a farm of servers is used to generate signatures, as in the case of a large CA. It is crucial that servers never reuse each other’s state. Our system provides simple ways to ensure that, as discussed in Section 4.

In Section 5 we tune the algorithms to the case of stateless signatures where there is no risk of state reuse. In this case signature size grows to about 20KB.

The hash-based signature scheme we use is built from one-time Winternitz signatures [13] combined with Merkle trees and builds upon earlier work in this space [6, 5, 4, 9]. We survey earlier work in Section 6 and compare to our results.

Generalizing Winternitz. In Section 3 we explore the question of generalizing the Winternitz one-time signature scheme to a \(k\)-time signature for small \(k\). Trivially, this can be done by setting up \(k\) independent keys. The question is whether there is a better solution that leads to shorter parameters. Similar generalizations for Lamport’s one-time signature [15, 16, 1] show that indeed Lamport’s scheme can be generalized to \(k\)-time signatures more efficiently than setting up \(k\) independent systems. Unfortunately, using a standard theorem from Ramsey theory we prove that a natural approach to generalizing the Winternitz scheme cannot give better results than trivially setting up \(k\) independent systems.

2. STATEFUL MERKLE SIGNATURES
This stateful scheme takes a one-way function (OWF) based one time signature scheme, Winternitz, and adapts it to an efficient many-time scheme using Merkle hash trees.

\(^1\)The best generic quantum collision finding algorithm runs in time \(2^{n/3}\) for an \(n\)-bit hash function [2].
2.1 Winternitz one-time signatures

Winternitz OTSS [13] is a generalization of the Lamport scheme [11]. Both schemes are based on an OWF. As with many OWF-based signature schemes, Lamport signature size makes it impractical. In order to be adapted for a 256-bit hash of a message, one needs a 1KB signature (256-128 bits), and this is only for a single use. The Winternitz Scheme can trade signature size with setup time. We will use the construction proposed by Buchmann et al, with security based on target-collision resistant hash functions [3].

Let $N$ be the message size, $p : \{0,1\}^* \rightarrow \{0,1\}^{128}$ a PRF, $f : \{0,1\}^{128} \rightarrow \{0,1\}^{128}$ be an OWF, and $H : \{0,1\}^* \rightarrow \{0,1\}^M$ a collision-resistant hash to $M$ bits. Given the parameter, $\ell$, generally a power of 2 in the range $2^2$ to $2^{10}$, we can setup an instance of the Winternitz OTSS, with a public key of only 256 bits.

Setup ($\ell \in \mathbb{N}, SK \in \{0,1\}^{256} \rightarrow (x_1, \ldots, x_n), PK \leftarrow$).

We compute

$$t \leftarrow \left\lceil \frac{N}{\log_2(\ell)} \right\rceil, \quad t' \leftarrow \left\lceil \frac{\log_2(\ell t)}{\log_2(\ell)} \right\rceil, \quad \text{and} \quad n \leftarrow t + t',$$

where $n \cdot 128$ will be the signature length in bits. Then we use a PRF seeded with $SK$ to generate $n$ secret keys $x_1, \ldots, x_n \in \{0,1\}^{128}$. Then, we compute chains of length $\ell$ using $f$. To get the $i^{th}$ chain, we iteratively compute $f$ on $x_i$. The $j^{th}$ link in the chain is $f^j(x_i)$. The last link is $f^{\ell}(x_i) = y_i$, which will be used to compute the public key.

The public key is the collision-resistant hash of $y_1, \ldots, y_n$. $PK = H(y_1, \ldots, y_n)$. This way, the signer only needs to publish $M$ bits of data as a public key.

On a high level, the setup should look like figure 1 below.

![Figure 1: On the left, what Winternitz scheme abstractly looks like after being set up. On the right, which parts of the scheme are used in the signature.](image)

**Signing** ($m, SK \rightarrow \sigma$). To sign an $N$-bit message, we compute $m \pmod{\ell}$, resulting in splitting the bits of $m$ into $t$ segments, $b_1, \ldots, b_t$ of $\log\ell$ bits each, $0 \leq b_i < \ell$. Then we compute a checksum, $C = \sum_{i=1}^{t} \ell - b_i$, and write $C$ in base $\ell$. We split $C$ up into $t'$ bits $b'_1, \ldots, b'_{t'}$, again $0 \leq b'_j < \ell$. The signature for $m$ is then $(f^{b'_1}(x_1), f^{b'_2}(x_2), \ldots, f^{b'_{t'}}(x_n))$; so we take one value from each chain into our signature, shown in figure 1. The checksum ensures that no signature will be computable from any other signature without the secret key.

**Verify** ($PK, \sigma = (s_1, \ldots, s_n), m$). To verify the $N$-bit message, we turn $m$ into the same values $b_1, \ldots, b_n$ using the same transformation as in signing; the value of $m$ is calculated base $\ell$ for $b_1, \ldots, b_n$, and for the rest, we calculate the same checksum base $\ell$ to get $b_{t+1}, \ldots, b_n$. Then we compute $y'_1, \ldots, y'_n$ so that

$$y'_i = f^{\ell - b_i}(s_i), \quad \text{and} \quad H(y'_1, \ldots, y'_n) = PK'. \quad \text{Finally the verifier accepts if} \quad PK' = PK.$$

Correctness. If the signature is valid, then each $y'_i = f^{\ell - b_i}(s_i) = f^{\ell - b_i}(f^{b_i}(x_i)) = f^{\ell}(x_i) = y_i$.

Security. This scheme is existentially unforgeable, as proved by Buchmann et al in [3].

2.2 An efficient many-time signature system

2.2.1 Merkle Trees

Let the capacity of a signing scheme instance be the number of messages it can securely sign. Winternitz one-time signature scheme had a capacity of 1. In order to have a capacity of $k > 1$ messages, we can generate $k$ independent instances of the Winternitz one time scheme and use the $i^{th}$ instance to sign the $i^{th}$ message. However, this process makes the public key larger by a factor of $k$, since we need to use a public key for each instance. Instead, we will use a full binary tree structure to hash the $k$ public keys in pairs to one string which will serve as the public key for all $k$ instances.

Formally, assume $k$ is a power of 2, and $Y_1, \ldots, Y_k$ are public keys from $k$ instances of the Winternitz scheme generated from single secret key $SK$, a PRF, an OWF $f$ and collision-resistant hash function $H$, as in the previous section. We can create a full binary tree with $k$ leaves $l_1, \ldots, l_k$ such that every node $m$ in the tree has an $M$ bit string $\phi(u)$ associated with it. We define $\phi(u)$ recursively. For each leaf $\phi(l_i) = Y_i$, and then for each node $u$ with children $u_L$ and $u_R$, $\phi(u) = H(u_L, u_R)$, the hash of the two children. The value at the root, $r$, of this tree, $\phi(r)$ will be the public key of the system, in addition to a description of $f$ and $H$. See figure 2 below for how the tree should look. This structure of hashing leaves to get their parents produces a Merkle hash tree [13].

![Figure 2: Structure of the Merkle tree with Winternitz OTSS. Here $k = 4$, and we have a full binary tree as described above.](image)

Setup ($h, \ell, SK$). We let $k = 2^h$, and run the setup for each of the $k$ Winternitz instances using the PRF on $SK$. 
We extract the public key from each instance and build the tree from the public keys $Y_1, \ldots, Y_\ell$ so that we can compute the value of the root of the tree $\phi(r)$.

**Signing**. To sign the $i^{th}$ message, we use the $i^{th}$ instance of the Winternitz OTSS. We can use the PRG on SK to get the secret keys for the scheme, generate the Winternitz signature $\sigma_W$ with public key $Y_i$. Then we send a proof that $Y_i$ is indeed the $i^{th}$ public key in the form of an authentication path in the hash tree.

The authentication path is defined as follows. Consider the path from the public key $Y_i$ up to the root node $r$, $u_1, \ldots, u_{h-1}, r$. Each $u_j$ has a sibling $v_j$, we let the authentication path be $v_1, \ldots, v_{h-1}$. See figure 3 for an example when signing the third message.

![Figure 3: An example of signing the third message $m$ using a Merkle Tree structure. The path from the public key $Y_3$ to the root is bolded. The authentication path nodes are shaded. The resulting signature is $\sigma = (\sigma_W, v_1, \ldots, v_{h-1})$.](image)

**Verify**. A verifier first computes $Y_i'$ from $\sigma_W$ using the verification for Winternitz signatures to get the public key corresponding to that signature. Then, he computes the root node using the given authentication path. This is done recursively. The verifier knows at $u_1$ should be $Y_i'$, so for each $1 < j < h$, $u'_j = H(u'_{j-1}, v_{j-1})$. The root node is computed as $r' = H(u'_h, v_{h-1})$. Finally, the verifier accepts if $r' = r$.

**Correctness.** Given a valid signature, we know that the calculated public key for the Winternitz instance $Y' = Y_i$. So, we have successfully computed $l_i = u_1$, the $i^{th}$ leaf of the tree. Since we have the sibling of $u_1$, we can get $u_2 = H(u_1, v_1)$. For $j > 1$, assuming that $u_{j-1}$ was computed correctly, $u_j = H(u_{j-1}, v_{j-1}).$ This implies that the root is also computed so that $r' = r$.

**Security.** This Merkle tree adaptation is generic for different one-time signature schemes. The adaptation has been proved existentially unforgeable by García in [7]. Since Winternitz is existentially unforgeable, so is the adaptation.

### 2.2.2 Adaptive Merkle Trees

When $k$ is large, a single Merkle tree structure becomes impractical: it takes too long to setup, and it is too large to store in memory. So, instead of storing the entire tree in memory, we can make the Merkle scheme adaptive, and only store sections of the tree at a time.

The idea is to make a tree of Merkle trees from the previous section. We will use the construction and algorithms proposed by Naor et al [14]. There is a root tree, which signs the public keys of trees below it. These trees in turn, sign the public keys of trees below them, etc. Each of these trees can be generated independently of each other, so we only need to store enough trees to sign a subset of the messages at a time. Let there be $H$ levels of trees. If we think of what is required to sign a single message, we need a bottom tree, at level 1, to sign the message itself, the tree above that to sign the public key of the bottom tree at level 2, etc. . . until the $H$-level (top-level) tree. So, we only need $H$ trees at a time. See figure 4 to see the structure.

However, we eventually run out of leaves in the bottommost tree signing messages. We need to have constructed another tree to replace it at this point. We do so in an amortized fashion, proposed by Naor et al [14]. For each leaf we use in the bottom-most tree when we sign (one leaf per message), we construct part of the next tree. We do the same process for the trees above for each leaf that is used. So, we are constantly generating the next trees to use, but in amortized constant time. This tree generation is covered by an algorithm called update.

**Notation.** We want to sign $k = 2^K$ messages. We have $H$ levels of subtrees of height $h_i$ for $1 \leq i \leq H$, so that $\sum_{i=1}^{H} h_i = K$. Then, we have a set of exist subtrees, that are ready to sign messages. Finally, we have a set of desire subtrees, which are being constructed to replace the exist trees once they run out of leaves to sign with. See figure 4 for the structure.

![Figure 4: An example of an adaptive Merkle tree with 3 levels. Each level $i$ has an exist and desire tree of height $h_i$, except for the root level, which only has an exist tree.](image)

At each of the $H$ levels, we can define an independent height $h_i$ and Winternitz parameter $l_i$. So, in setup, we need $H$ heights and $H$ $l$s to define the scheme.
Setup \((H, h_1, \ldots, h_H, \ell_1, \ldots, \ell_H, SK) \rightarrow PK\). These parameters yield an Adaptive scheme with \(H\) levels of trees, where the height of the top-most tree (the root tree) \(h_H\), and the height of the bottommost trees \(h_1\). Each \(\ell_i\) is the Winternitz security parameter for the Winternitz signatures at the leaves of tree \(i\). \(SK\) is a secret key used to generate the secrets of all of the Winternitz OTSS.

The signer initializes \(H\) Merkle trees the \(i\)th tree of height \(h_i\), using \(SK\) and a PRF, to generate the other secret keys for each of the trees. \(tree_H\) is the top tree, and its root is the Public Key. \(tree_H\) signs the public key of \(tree_{H-1}\), and so on so \(\ldots\) \(tree_i\) signs the public key of \(tree_{i-1}\). \(tree_1\), has its leaves available to sign actual messages. Most of the signature, except the part of the signature generated by leaves, can be stored in memory and updated with each new message it signs; most of the signature does not change from message to message.

Signing \((m_i) \rightarrow (\sigma_1, \ldots, \sigma_H)\). Given \(m_i\), message \(i\), to sign, the server first performs an update on its stored signature and its tree structure.

The algorithm for updating the tree structure involves creating Merkle trees. Each operation creates a single node in a Merkle tree: either it sets up a Winternitz OTSS to get the public key for a new leaf node or it hashes two nodes together to get a new parent node. See figure 5 for an example of how update should work.

1. We check if we can sign \(m\) using the bottom tree, an existing \(tree_1\). If not, we swap it with its corresponding desire tree to get a new set of leaves to sign with.
2. Then, we work our way up the levels of trees. If \(tree_i\) needs to sign the \(tree_{i-1}\) public key, \(r_{i-1}\), then we check if \(tree_i\) has the capacity to sign another message. We have two cases. If \(tree_i\) can sign another message, then we sign it and perform two operations on the desire \(tree_i\). Otherwise, we switch \(tree_i\) with its desire tree, sign the public key with this new tree, and perform two operations on a new desire \(tree_i\). We will next need to sign the new \(tree_i\) public key with \(tree_{i+1}\). So, we repeat step 2.

After the update, the signature in memory now corresponds to \(m_i\), so it is sent to the client.

The signature is a list of Merkle tree signatures: \(\sigma = (\sigma_1, \ldots, \sigma_H)\). \(\sigma_i\) signs the message using \(tree_1\), \(\sigma_i\) signs the public key of \(tree_{i-1}\) for \(i > 1\).

Verifying \((PK, m_i, (\sigma_1, \ldots, \sigma_H))\). The signature consists of \(H\) Merkle Tree signatures, \(\sigma_i\). Verifying is an iterative algorithm. \(m\) and \(\sigma_1\) are used to compute the public key \(r_1\) of \(tree_1\), described in section 2.2.1. Iteratively, for \(i > 1\), public key \(r_{i-1}\) of \(tree_{i-1}\) is used to compute \(r_i\), the public key for \(tree_i\), using the Winternitz signature in \(\sigma_i\). After all iterations, we have \(r_H\), the public key for \(tree_H\), the root tree. The verifier accepts if \(r_H = PK\).

Correctness. This follows from the correctness of the Merkle tree scheme. If \(r_i\) is calculated correctly as the public key of \(tree_i\) in verification, then \(r_{i+1}\) is calculated correctly. Since \(r_i\) is calculated correctly if \(\sigma_i\) is an appropriate signature form \(m\) using \(tree_1\), it follows that all \(r_i\) are computed correctly. Thus, \(r_H = PK\).

Security. This follows from the security of the Merkle tree scheme. If an adversary is able to forge a signature here, he can forge at least 1 signature \(\sigma_i\) from a single Merkle tree. Therefore, this scheme is also existentially unforgeable.

3. EXTENSIONS

Notice that the construction of the many-time scheme was based on a one-time signature scheme, Winternitz. What if we used a two-time signature scheme that had comparable space and time to the one-time scheme? We would be able to sign as many messages with roughly half of the space currently used in storing the tree. Perhaps a \(k\)-time generalization of Winternitz is more efficient than the Merkle tree with \(k\) leaves construction. As an added bonus, this scheme would be stateless – no need to keep track of messages already sent.

In this section, we generalize the notion of the Winternitz one-time signature into a \(k\)-time signature. We then examine the \(2\)-time generalization and find evidence that this and other generalizations are not as efficient as the original one-time signature scheme.

One good measure of efficiency is the minimum size of the signature. The size of the signature is directly proportional to the number of chains we make in the setup phase, shown in section 2.1; the signature takes a representative node from each chain based on the contents of the message. In the one-time Winternitz scheme, we only require two chains, and so the minimum signature size is very small. We will prove that in the two-time scheme, the minimum signature size is not a constant, but is proportional with respect to the number of bits in the messages we sign.

3.1 Notation

First, we need to formalize some terms: chains, cuts, and covers. \(C_{\alpha, \ell}\) be the directed graph composed of \(n\) dis-
joint paths, chains, $C_1, \ldots, C_n$ of length $\ell$; this graph will look like figure 1, in section 2.1, where edges represent application of $f$. We denote the nodes of each chain $C_i$ by $C_i(1) \rightarrow C_i(2) \rightarrow \cdots \rightarrow C_i(\ell)$. Let a cut be a subgraph composed of exactly one node $C_i(b)$ from each chain, i.e. a signature. Let $S(C_{n, \ell})$ be the set of all cuts. For a cut $S \in S(C_{n, \ell})$, we say that $S(i) = b$ if node $C_i(b) \in S$. We say that a set of cuts $S_1, \ldots, S_k$ covers another cut $T$ if for every $C_i(b) \in T$, there is some $S_j$ and $C_i(c) \in S_j$, $c \leq b$. In terms of the Winternitz signature scheme, this means we can construct signature $T$ using $f$ and previously seen signatures $S_1, \ldots, S_k$, being able to construct a new signature from others, without knowing the secret key, renders the scheme insecure. So, we define the notion of $k$-cover-free. A set of cuts $S \subseteq S(C_{n, \ell})$ is $k$-cover-free if for every $k$ cuts, $S_1, \ldots, S_k \in S$, there does not exist another cut $T \in S$ such that $S_1, \ldots, S_k$ covers $T$.

If we have a $k$-cover-free set of $2^N$ cuts $S$, of size $2^N$, we can make a $k$-time secure signature scheme, generalizing Winternitz. We can bijectively map every $N$-bit message $m$ to a cut $S_m$, and the signature for $m$ is simply the nodes in $S_m$, where the nodes are in chains generated by a secret key and a one-way-function $f$ as in section 2.1.

### 3.2 Two-Time Winternitz

Now that we have formalized our extension of the Winternitz one-time scheme into a $k$-time scheme, we compare the minimum signature size of the two-time scheme with the one-time scheme.

Our exposition of the Winternitz one-time signature scheme (Section 2.1) shows that a set of cuts $S \subseteq S(C_{n, \ell})$ of size $2^N$ that is 1-cover-free whenever

$$n \geq \left[ \frac{N}{\log_2 \ell} \right] + \left[ \log_2 \ell + \log_2 \left( \frac{N}{\log_2 \ell} \right) \right].$$

Observe that by setting $\ell = 2^N$ and $n = 2$ we get a set of 1-cover-free cuts of size $2^N$ on two chains of length $2^N$. This construction achieves the optimal (minimum) signature size, requiring only $n = 2$ chains, independent of $N$.

Unfortunately, despite the fact that when $k = 1$, the answer is 2 no matter how large $N$ is, we prove that even for $k = 2$, $n$ cannot be a constant independent of $N$. This is negative evidence regarding the possibility of efficient stateless $k$-times schemes.

**Claim 1.** There does not exist a 2-cover-free set of cuts $S \subseteq S(C_{3, 5})$ of size 5.

**Proof.** Before we can prove the main theorem, we will prove this specific case. We have three chains of height 5. For a contradiction, assume there are five cuts, $a, b, c, d, e \in S(C_{3, 5})$, that are 2-cover-free. Let $a$ be the cut that appears first on the first chain. Now, without loss of generality, we can assume that $a$ appears last on the other two chains, that is $a[2] = a[3] = 5$. This is because for any two cuts, since $a$ appears first on the first chain, $a$ is still not covered by them, and if we take $a$ and any other cut $x$, if $a$ and $x$ did not cover any cut before, moving $a$ the last node of two chains means that $a$ and $x$ will still not cover any other cut.

We can define cut $b$ as the lowest cut on the second chain, and just as we did with cut $a$, can assume that $b[1] = b[3] = 5$. Then, we let cut $c$ be the lowest cut on the third chain, and can assume that $c[1] = c[2] = 5$. These chains are shown in figure 6.

Now we consider the last two cuts, $d$ and $e$. We have two symmetric cases: for at least two chains, $d[1] \leq e[1]$ or for at least two chains $e[1] \leq d[1]$. Since these cases are symmetric, assume without loss of generality that $d[1] \leq e[1]$ on at least two chains. Again, without loss of generality, we can assume that these two chains are just chains 1 and 2, since the setup of cuts $a, b$, and $c$ is symmetric on all three chains, as in figure 6. This implies a contradiction, since now $d$ and $e$ cover $c$, and $S$ is not 2-cover-free. The second case is symmetric, and again we have a contradiction.

**Theorem 1.** There does not exist a 2-cover-free set of cuts $S \subseteq S(C_{n, 2^N})$ of size $2^N$ if $n$ is a constant independent of $N$.

**Proof.** For a contradiction, assume that $n$ is a constant independent of $N$, and the set of $2^N$ cuts $S \subseteq S(C_{n, 2^N})$ is 2-cover-free. We will show that this implies the existence of a 2-cover-free set of cuts $S(C_{n, 2^N})$ of size 5, which is a contradiction given the previous claim.

We will now construct a complete 3-uniform, undirected, hyper-graph on $2^N$ nodes: one node for each cut in $S$. Next, we will color each of the edges with one of $\binom{n}{2}$ colors, one for each triple of edges. For a hyper-edge $(a, b, c)$, we look at the cuts $a, b, c$ and $S$ of this chain. Since this chain is a clique, we can choose any three of them, and they should still be 2-cover-free on these three chains. Now we have a contradiction, since it is impossible to have 5 2-cover-free cuts on three chains.

### 4. Implementation and Experiments

First, we explore the idea of a network of servers signing messages under the same public key. Even though the sys-
tem is stateful, machines do not have to share their state with each other, making it practical for a large network of signing servers.

Then, we provide an implementation of the Adaptive Merkle Scheme from section 2.2.2 in C++, using OpenSSL for AES and Crypto-Plus-Plus for the SHA-256 hash function. AES is used as the pseudorandom generator, while SHA-256 acts as a one-way-function. We use different libraries because OpenSSL seems to have the faster implementation of AES, while Crypto-Plus-Plus has the faster SHA-256. We also looked at using an HMAC instead of AES, but AES was faster.

4.1 Real-world considerations

The state of an instance of the Adaptive Merkle Scheme is the number of messages already signed. Since the scheme is adaptive, only a small set of leaves are available at a time to sign. In order to prevent a user from entering a message number that corresponds to an unavailable leaf, our implementation only allows the user to view state, but not to change it. The API of this implementation includes setup, signing, verifying, and viewing state.

This scheme lends itself well to a signing network of servers. Each server maintains its own state, so there is no state-sharing needed between servers, and it is easy for a signing server to recover from a loss of its own state.

A signing server network would involve the signing servers connected to an internal network to communicate with a main server, seen in figure 7. Recall that the capacity of a signing scheme is the number of messages it can securely sign. The main server sets up an adaptive signing scheme with a capacity proportional to the number of signing servers – if there are $2^{10}$ signing servers, the main server should be able to sign $2^{10}$ messages. The public key for the main server is the public key for the entire network.

To initialize, each signing server sets up its own adaptive signature scheme, and then gets its public key signed by the main server. Once it has its public key signed, it can sign incoming messages, appending the signature from the main server with its own. When a signing server has hit its capacity or has lost its state, it sets up another adaptive signature scheme with a new public key and gets it signed again by the main server.

The main server, then, spends most of the time offline. Only when a signing server needs to be initialized does it wake up, sign, and then go back to sleep. Consider models where it only needs to wake up once a day or once a week to sign. However, the main server must keep track of its state; it contains the main public key and cannot re-setup without changing that.

4.2 Experiments

Under current assumptions in cryptography, we can use SHA-256 truncated to 128 bits as a target-collision resistant OWF for Winternitz. We can use the full SHA-256 as a collision-resistant function $f$ when hashing nodes together in Merkle trees. AES in counter-mode is a valid PRF, so we use it to generate secret keys for the Winternitz leaves, based on the master secret key for the entire system.

Before running the main experiments to compare the Adaptive Merkle scheme with RSA, we first optimized for signing time using a simulation in Python. This optimization is absent in other papers, probably because there are too many parameters to optimize analytically. Our approach was to optimize via brute force over most of the practical values for parameters.

Our implementation of the Adaptive Merkle scheme is meant to be flexible enough to account for all possible parameters, not to have hard-coded pre-optimized parameters. This made it easy to test how well our implementation fared. The scheme is built in three parts: the base is Winternitz, the next are the single Merkle trees, and the final is the entire Adaptive tree. This means we need to account for the parameters of each Winternitz signature in each tree, on each level, and each of the heights of the trees. However, by applying some reasonable constraints on each of these parameters, we were able to optimize for the smallest signature size given a setup and signing time.

All experiments were run on machines with AMD Magny Cours, 24 cores each, with 96 GB of RAM. There was no parallelization in our implementation.

4.2.1 Parameter Optimization

Due to constructing trees as we go, signing is an amortized constant operation; the average time to sign Winternitz in the bottom-most leaves and the average time of an operation of constructing a new tree. To optimize for signing time, we want more Winternitz signatures that are faster to initialize; these correspond to smaller $\ell$ parameters. However, smaller $\ell$ parameters correspond to larger signature sizes. So, it is best to keep small $\ell$ parameters in the bottom-most layers of our adaptive tree. This way, we only have one or two large signatures so the other Winternitz signatures can be smaller.

When running the experiments, we wanted to optimize for the best parameters for signing time, while keeping setup time, signature size, and verification times reasonable. Since there are too many variables affecting signature size, setup and signing times, it is not practical to try to analytically optimize. So, we also constructed a simulation which gives an accurate prediction for the time and space an operation
Adding more trees greatly increases signature size, since the time linearly, while decreasing signature size by a log factor. However, it increases setup the Winternitz parameter \( \ell \) times takes off less than half of a KB. This is because changing KB from the signature size, but subsequently adding 30 minutes: at first going from 1 minute to 30 minutes takes one than 2 KB in the 2

64

case. Any more than an leaves, we tested setup long from all of the Winternitz instances. Any more than an \( \ell \) parameter of \( 2^{10} \), and Winternitz signatures take too long to setup. Our optimization limited setup time and signature size and optimized for signing time less than the signing time for running RSA on the same machine.

Table 1 has the results of the optimization. Since trees with \( 2^{32} \) leaves are much smaller than those with \( 2^{64} \) leaves, we tested setup times of 1 minute, 5 minutes, 10 minutes and 30 minutes. For the larger tree, \( 2^{64} \) leaves, we tested setup times of 1 minute, 30 minutes, one hour, and two hours.

### 4.2.2 Results

Using the simulation, we were able to optimize for which parameters would give the best signature size provided limits for setup time and signing time (bounded by the signing time for RSA). We notice that increasing the number of levels leads to larger signature sizes in table 2, so it is best to have as few layers as possible, while keeping setup time within reason.

We notice that increasing the setup time actually does very little to decrease signature size; increasing the time from 1 minute to 120 minutes changes signature size by less than 2 KB in the \( 2^{64} \) case. There are also diminishing returns: at first going from 1 minute to 30 minutes takes one KB from the signature size, but subsequently adding 30 minutes takes off less than half of a KB. This is because changing the Winternitz parameter \( \ell \) can greatly increase setup time, while decreasing signature size. However, it increases setup time linearly, while decreasing signature size by a log factor. Adding more trees greatly increases signature size, since the majority of the signature is the Winternitz signature, but decreases setup time.

Our goal is to rival RSA as a signature scheme, so we compare our implementation’s performance to 2048-bit RSA in table 3. We see that signing time is essentially equivalent to RSA signing time. Verification, however, takes much more time relative to RSA signing time. From table 2, we know that we can have Adaptive verification time similar to Adaptive signing time, since they are very similar operations. However, because they are similar operations, verification time will not be able to rival RSA verification in this setting.

Signature size is also, unfortunately, not very comparable to RSA signature size. However, when considering the practicality of the signature, when sending data over the internet, a single packet is 4 KB. So, the \( 2^{32} \)-leaf Adaptive scheme easily makes it under 4 KB. Unfortunately, \( 2^{64} \) leaves are too many to have a signature size of under 4 KB efficiently. The fastest possible setup time with a signature size of 4 KB is approximately 15 hours (about 900 minutes), according to the simulation.

Notice that although in every metric but signing time, RSA outperforms the Adaptive Merkle scheme, signing time is most important for something like a certificate authority. We chose to focus on signing time because a certificate authority would want to sign many thousands of messages quickly, while space usage and setup times would be much less important. Even though signatures are bigger, they still fit within one packet.

### 5. STATELESS MERKLE SIGNATURES

The Adaptive Merkle scheme is a stateful signature scheme, where the number of messages previously signed must be kept track of. This means, if a signing server running the Adaptive Merkle scheme lost its state due to a reset or an attack, it would need to start the setup process all over again; it is not safe to use the same leaf multiple times. There is a stateless extension of the Adaptive Merkle scheme, but unfortunately, it is not efficient enough in signature size or signing time to be practical.

The idea for the stateless scheme is to use the same struc-

---

**Table 1: Table of simulated values for the stateful signature scheme.**

<table>
<thead>
<tr>
<th>Number of Leaves</th>
<th>Signature Size (KB)</th>
<th>Setup Time (min)</th>
<th>Signing Time (ms)</th>
<th>Verification Time (ms)</th>
<th>Number of Levels</th>
</tr>
</thead>
</table>
| 2

64

| 2.81 | 0.97 | 4.68 | 24.38 | 3 |
| 2.79 | 2.94 | 4.71 | 56.64 | 3 |
| 2.44 | 9.32 | 4.68 | 4.11 | 2 |
| 2.23 | 29.02 | 4.68 | 29.08 | 2 |
| 5.97 | 0.97 | 4.68 | 18.32 | 6 |
| 4.63 | 29.35 | 4.68 | 13.15 | 4 |
| 4.36 | 59.46 | 4.68 | 32.83 | 4 |
| 4.17 | 118.38 | 4.68 | 84.24 | 4 |

**Table 2: The experimental results from running the implementation.**

<table>
<thead>
<tr>
<th>Signature Scheme</th>
<th>Signature Size</th>
<th>Setup Time (sec)</th>
<th>Signing Time (ms)</th>
<th>Verification Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA-2048</td>
<td>.25</td>
<td>.02</td>
<td>6.01</td>
<td>0.20</td>
</tr>
<tr>
<td>Adaptive ( 2^{32} ) 2.81</td>
<td>62.42</td>
<td>5.18</td>
<td>30.31</td>
<td></td>
</tr>
<tr>
<td>Adaptive ( 2^{64} ) 5.97</td>
<td>63.07</td>
<td>5.03</td>
<td>21.30</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3: Comparing the performance of 2048-bit RSA to the Adaptive Merkle scheme.**

---
ture from the Adaptive Merkle scheme, except instead of using the leaves in order, we randomly choose a leaf to use to sign a message. A problem with this is that we rely on the fact that a different leaf is chosen to sign each message. So, if we want to sign $M$ messages, for a large enough set of leaves, this probability is negligible. To sign $2^{32}$ messages, this means we need on the order of $2^{128}$ leaves.

A computer cannot physically hold a single tree of height $128$ (almost $10^{22}$ petabytes), so we will use the adaptive scheme. However, since we cannot predict which leaf we are using to sign a message, we cannot prepare the tower of trees as we did in the Adaptive setup in section 2. The only tree we know will be used is the very top tree with the public key. The other trees need to be generated during signing time.

**Setup.** During setup, the signer only needs to create the top Merkle tree, so that it can publish the public key. This root tree will be necessary to sign every message, so it is best to keep this tree in memory.

**Sign.** To sign a message, we choose a random $i \in \{0, 2^K\}$, where $K$ is the total height (so $2^K$ is the number of leaves), and sign with that leaf. The signature will be the same as with stateful, so we will need to create each tree along the path from leaf $i$ to the root. Then, we compute the signature by computing the Merkle signature of the appropriate leaf at each tree and the Winternitz signatures between the trees. The signature needs to contain the verification path at each tree, the Winternitz signatures, and the leaf number.

**Verify.** Verifying the signature of a stateless signature is exactly the same as verifying in the stateful case. The verifier calculates the public keys of each tree, and verifies the Winternitz signatures between them.

Note that caching other trees is unnecessary and ultimately unhelpful, since messages should correspond to leaves independently of each other. In order to take advantage of caching, it is best to make a large root tree, since keeping smaller trees requires more space and time per each node that Winternitz public keys and signatures require.

### 5.1 Signing Capacity and Security

Security for the stateless scheme comes from assume the probability of choosing the same leaf to sign two different messages is negligible. If it is, then we have the same security as in the stateful scheme, section 2, since if we can forge a signature here we have either found a collision, or have broken the scheme.

Having a tree of size $2^{128}$ where the signer chooses a random leaf to sign a message is only effective for about $2^{32}$ messages before the signer is at risk for a collision – using the same leaf to sign two different messages. However, once we get to $2^{256}$ leaves, we can us a collision-resistant hash function like SHA-256, to choose a leaf equal to the hash of the message.

### 5.2 Simulated Estimation of Stateless Signatures

Using the same program, we estimate the signature size and signing/verifying times as we did for stateful signatures. The program gives a good estimation for how long certain procedures should take, as can be seen by comparing previous simulations to the empirical results.

<table>
<thead>
<tr>
<th>Number of Leaves</th>
<th>Signature Size (KB)</th>
<th>Setup Time (min)</th>
<th>Signing Time (ms)</th>
<th>Verification Time (ms)</th>
<th>Number of Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{256}$</td>
<td>16.39</td>
<td>521.00</td>
<td>6.22</td>
<td>61.48</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>9.42</td>
<td>521.00</td>
<td>611.10</td>
<td>88.91</td>
<td>10</td>
</tr>
<tr>
<td>$2^{128}$</td>
<td>20.59</td>
<td>2084.0</td>
<td>13.14</td>
<td>61.61</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>17.00</td>
<td>130.25</td>
<td>138.39</td>
<td>267.39</td>
<td>18</td>
</tr>
</tbody>
</table>

**Table 4: The results obtained from simulating a stateless adaptive Merkle signature scheme.**

Since signing with the stateless scheme is much more symmetric, we used a simpler optimization. We let the top tree be largest and have an independent $\ell$ Winternitz parameter, and each of the other trees be of the same height with the same $\ell$ parameter, since they will all need to be generated and used for signing each message. Then, we ran our simulation on reasonable parameters to get the following data for trees of both heights 128 and 256.

From table 4 we see the results of running the simulation program for stateless signatures. It’s easy to see from the table, just as in stateful, fewer levels of trees means a smaller signature size. However, more levels corresponds to a faster signing time. Notice that there are no signing times less than a second. This is because as long as there are multiple levels in the tree, it is impossible to sign in less than a second.

Signing time is the crutch of the stateless signature. We need to be able to generate Merkle trees on the fly, but setting up these trees takes time. Setting up Winternitz signatures accounts for most of this time, since Winternitz require many hashes. If signing time is not important, then this signature is a robust, multi-use, stateless, signature.

### 6. RELATED WORK

The Winternitz one-time scheme and the Merkle trees are presented by Merkle in [13]. The full explanation of the Merkle tree, many-time scheme in section 2.2, was given by Dodis et al [6] where they also prove its security for certain hash functions and provide experimental results. They only present results for Merkle trees of height at most 14 ($2^{14}$ leaves) and where the $\ell$ parameter in Winternitz is at most 4. Naor et al [14] implement this scheme, but with only fractal trees of height at most 20. This project explores larger trees, a greater range of $\ell$ parameters, and optimizes for signature size and time.

The adaptive tree generation algorithm was first suggested by Buchmann et al [5] and a general such tree is given in [4]. These follow the offline/online paradigm of Goldwasser et al [8]. The Buchmann et al scheme in [4] is essentially the same as the one given here, but there are some differences.

First, the implementation does not perform any offline operations; it works like the Naor et al implementation in [14]. Second, they use SHA-160 as both a OWF and collision resistant hash. We need SHA-256 for a collision resistant hash, and we settle for the first 128-bits of SHA-256 for a OWF.

Third, they use the Szyldo tree traversal algorithm in [17] instead of storing the trees in memory, which makes their signing times larger. Finally, they use fewer layers in the adaptive trees, making their setup times much longer; the shortest setup time they report is more than 300 minutes. Here, we generally have setup times less than 60 minutes.

An alternative tree-traversal/tree-generation algorithm has recently been proposed by Knecht et al [10]. They described
three improvements to the traversal algorithm we use here. Two of them trade space efficiently with time, but the most time-efficient, is the algorithm we use; we only calculate each node on the tree exactly once during setup and keep the entire tree in memory. We focused on time efficiency. The third improvement made worst-case signing times closer to average-case signing times. We chose to focus on average-case signing times, not on improving worst-case, so that we could compare average signing times to average RSA signing times.

The stateless version of this scheme has been discussed by Hopwood in [9] and Langley in [12]. They both confirm the need for many leaves, and discuss how the stateless construction is more secure than the stateful version (by virtue of being stateless). Hopwood also discusses security against rollback attacks [9]. Langley, like we do, provides a simulation for a stateless scheme. However, he assumes that all of the trees are the same height when it is beneficial to have a larger root tree as a method of caching, as shown in Section 5.

7. CONCLUSIONS

Building on previous work on Adaptive (or fractal) Merkle trees, we provide an implementation with experimental results, along with simulations for a stateless modification, and applications for the stateful version. By making the Merkle tree adaptive we were able to cut back on the storage required to sign messages and the time required to set up the scheme.

Signing with this hash-based scheme is faster than signing using RSA, but signature verification is slower and signatures are longer, about 4KB. Since the total size of RSA certificates is about 1KB this would mean adding 3KB of traffic to the TLS server_hello message.

To conclude we note that there are no known lower bounds on the length of hash-based signatures and it possible that further improvements, along the lines of Section 3, will further reduce their size.

8. REFERENCES