Hash-based TPM Signatures for the Quantum World

Megumi Ando, Joshua D. Guttman, Alberto R. Papaleo, and John Scire

The MITRE Corporation, Bedford, U.S.A.
{mando, guttman, apapaleo}@mitre.org, jscire@stevens.edu

Abstract. Trusted Platform Modules (TPMs) provide trust and attestation services to the platforms they reside on, using public key encryption and digital signatures among other cryptography operations. However, the current standards mandate primitives that will be insecure in the presence of quantum computers. In this paper, we study how to eliminate these insecure primitives. We replace RSA-based digital signatures with a hash-based scheme. We show that this scheme can be implemented using reasonable amounts of space on the TPM. We also show how to protect the TPM from rollback attacks against these state-sensitive signature operations.

Keywords: Post-quantum · Trusted Platform Module · TPM · Attestation Identity Key · AIK · Merkle trees

1 Introduction

A Trusted Platform Module (TPM) is a low-cost cryptographic microprocessor for enabling trusted computing functionalities. TPMs follow a set of global industry standards laid out by the Trusted Computing Group (TCG) also called the TCG standards (versions 1.2 and 2.0). As of this writing, there are a number of vendors which supply hardware TPMs: AMD, Atmel, Broadcom, IBM, Infineon, Intel, Lenovo, National Semi, Nationz, Qualcomm, STMicroelectronics, Samsung, Sinosun, Texas Instruments, Nuvoton Technology, and Fuzhou Rockchip [1]; and most personal computers and a few mobile devices (with the notable exception of Apple computers) are shipped with a TPM [2]. It is assumed that a TPM is implicitly trustworthy and tamper-resistant. This trust is bootstrapped to enable two trusted computing capabilities, secure encrypted storage and remote attestation, explained below.

1.1 Secure Encrypted Storage

Costing a couple of dollars each, a TPM is an inexpensive solution for securing sensitive data even under the threat that the hosting platform may become corrupted [2]. This is achieved by storing sensitive information outside the TPM only in encrypted form, with the corresponding encrypting TPM key either in the TPM's hardware-protected area or also encrypted by another TPM key. In

this way, a TPM serves as a tamper-resistant hardware Root-of-Trust for Storage (RTS) [2–4].[1]

Because the TPM has a limited amount of non-volatile RAM (NVRAM), only a few special TPM keys are kept in protected space within the TPM. Other keys are stored in encrypted form in unprotected areas on the platform. They exist in unencrypted form only transiently as needed and only within the TPM. The storage keys thus form a key hierarchy (a tree), where each non-root key is encrypted under its parent key.

In TPM version 2.0, the encryption of child nodes can be done either symmetrically or asymmetrically [2, 3]. Thus, we can trivially avoid a quantum-insecure storage mechanism by using only symmetric storage keys. A single caveat to the solution is duplication, or a mechanism for transferring or backing up TPM keys. Currently, duplication is handled by decrypting a specific key (or even an entire subtree) and then re-encrypting it with the public portion of another TPM's RSA key [2–4]. Symmetric encryption cannot replace RSA encryption without preplacing shared TPM keys across platforms.

1.2 Attestation

A platform state measurement indicates whether a platform is in an expected state and therefore still trustworthy. Through a separate mechanism, the hash of this measurement can be written into special TPM registers called Platform Configuration Registers (PCRs). This is done using a trusted hardware component: a Root-of-Trust for Measurement (RTM; e.g., Intel TXT for boot measurements) [5].

A TPM can be used for securely reporting the measurement stored in the PCRs [2–7]; and so it serves also as a hardware Root-of-Trust for Reporting (RTR). This is done using two types of TPM keys: an Attestation Identity Key (AIK) and an Endorsement Key (EK). An AIK is an asymmetric signature key corresponding to a user and or application on the platform and is used for enabling anonymity. There may be many AIKs per platform. To be useful, an AIK must satisfy the property that it is infeasible for an adversary to forge a valid signature (under the key) using any information stored on the platform outside of the TPM. An EK is an asymmetric encryption key with the unforgeability property that it is infeasible for an adversary to create a valid encryption under the key. It is unique to the platform, created randomly, and properly certified by the manufacturer. The EK is kept inside the TPM, never leaves the TPM, and is used for certifying AIKs as genuine.

The AIK is used to sign a quote, which includes the measurements stored in the PCRs as well as a verifier-provided nonce for freshness. For the signed quote to be meaningful, the AIK itself is certified in order to prove that the AIK

[1] Note that while a TPM can guarantee confidentiality and detect modification of securely stored data, it cannot retrieve secured data in the event that data is damaged; some other mechanism should be implemented to mitigate data loss.

belongs to a genuine TPM. The signed quote, along with the AIK's certificate, is sent to a verifying party.

Each TPM is shipped with a vendor-certified EK (version 1.2) or Primary Seed (version 2.0) [2–4]. All AIKs are securely derived from this key or seed as needed and certified by a trusted Privacy-CA (PCA). The Direct Anonymous Attestation (DAA) protocol also ensures anonymity [8]. This alternative does not require a trusted third party but incurs a large cost in cryptographic complexity instead.

1.3 Our Contributions

Current TPM solutions were designed to be secure against classical adversaries, as opposed to quantum adversaries. This is problematic since some of the trust arguments rely on the intractability of computational problems with known efficient quantum algorithms. For example, TPM keys were originally (in version 1.2) all RSA keys, which are insecure given Shor's quantum integer factorization scheme [9], and both PCA and DAA protocols use RSA encryption.

This paper presents our initial investigation into architecting a quantum-secure TPM. We have identified the following set of current TPM mechanisms to be those which rely on RSA and, thus, known to be vulnerable to quantum-attacks:

1. Secure (encrypted) storage
2. Duplication
3. TPM signing keys
4. PCA protocol
5. DAA protocol
6. Encrypted transport session

(By encrypted transport session, we mean transporting information, e.g., a TPM command, securely to a TPM.)

Although RSA is insecure in a post-quantum era, there are some notable alternatives without any known attacks. One such alternative is hash-based digital signature schemes, such as Merkle's tree authentication using a one-time signature (OTS) scheme. We claim that Merkle tree signatures are a practical alternative to RSA authentication, solving issues 3 and 4 above. (**Disclaimer:** The signing-based PCA protocol described in this paper is weaker than the original; we lose repudiability of a claim that two AIKs are linked.)

In this paper:

- We present our hash-based TPM signing key construction: QUAntum Secure Hash (QUASH). Given the space limitation of the TPM, our solution offloads most of the storage to the untrusted platform in a way that preserves security.
- We also show that our solution prevents replay attacks; it prevents a particular OTS key from being used to sign multiple messages.
- Lastly, we provide recommendations for system parameters.

1.4 Related Work

Efficient quantum algorithms [9, 10] break RSA, Digital Signature Algorithm (DSA), and Elliptic Curve Digital Signature Algorithm (ECDSA) in a quantum-world setting [11]. Fortunately, a number of cryptographic techniques are believed to be quantum-secure despite these known algorithms, including: hash-based, code-based, lattice-based, and multivariate-quadratic-equations cryptography [11, 12]. In this paper, we propose a hash-based TPM signing solution, whose security relies solely on the existence of collision-resistant hash functions and pseudorandom number generators and assess its practicality as a system. Our approach for storing data outside the TPM is similar to the technique presented in [13] for creating virtual monotonic counters for TPMs.

Lattice-based cryptography provides a strong alternative candidate for enabling quantum-secure TPM signatures. A recent paper [12] presents an efficient signature scheme Tightly-secure Efficient Signatures from standard LAttices (TESLA) that relies on the intractability of standard lattices as opposed to ideal lattices. The paper presents *fixed* parameters for quantum-security. Our hash-based approach has the benefit that its security is based solely on the existence of collision-resistant hash functions and pseudorandom number generators, whereas the lattice-based approach relies on the intractability of lattice problems. However, the hash-based approach has a key management problem that the lattice-based approach avoids. A more in-depth comparison between our approach and one that uses this lattice-based signature scheme is outside the scope of this paper.

Road Map: Section 2 contains our problem statement (the definitions and system model we adopt for our construction) and the technical background necessary for understanding our post-quantum attestation solution. In Sects. 3 and 4, we provide our main results: our hash-based TPM signing key construction, QUASH, and a practicality assessment of this solution. In Sect. 5, we conclude with a summary of our work.

2 Problem Statement and Preliminaries

Our goal is to architect hash-based TPM signing keys, defined below.

Unlike the RSA signature scheme, hash-based signature schemes require maintaining some key state information to work. At a minimum, the leaf number is needed in order to avoid reusing an OTS key pair; and some auxiliary information is required for efficient signing. We also introduce the notion of an Endorsement *Signing* Key (ESK), which is a vendor-certified TPM *signing* key for certifying AIKs.

Definition 1 *A tamper-evident hash-based AIK is a TPM signing key with the additional property that it is infeasible for an adversary to change an AIK state value without the change being detected.*

Definition 2 *The hash-based ESK is an unforgeable TPM signing key derived from a Primary Seed, with the property that it is infeasible for an adversary to*

change the ESK state value. (A Primary Seed is a large random number which is created by the vendor and protected by the TPM.)

If the number of AIKs is limited and relatively small, we can simply store all required key state information within the protected NVRAM space of the TPM. Our goal is to extend this solution to the case where an *unbounded* number of AIKs can be created, despite the hard space limit within the NVRAM. We do this by storing the key state information outside of the TPM in balanced binary (e.g., red-black trees or treaps) hash trees and by keeping only digests of the state information within the NVRAM for integrity protection.

Our construction, QUASH, has the following set of desirable properties.

- **Tamper-evidence: prevention of OTS key reuse.** The key state information is integrity-protected by storing hash digests in the TPM NVRAM. Thus, our tamper-evident signing solution prevents an OTS key from being used more than once, a concern if we use hash-based signatures.
- **Availability: localization of data loss.** While a TPM can guarantee confidentiality and data modification detection of securely stored data, it cannot retrieve secured data in the event that encrypted data is lost. Thus, a modification to any of the AIKs' state information renders *multiple* AIKs unusable. Given "registers" for holding multiple integrity check values within the TPM's NVRAM, we minimize the number of AIKs that are lost when the state information of a single AIK is corrupted.
- **Efficiency: TPM space requirements.** We show that our construction can work with a space-limited TPM, despite hash-based signatures schemes requiring key state storage and producing large signatures.
- **Efficiency: integrity checks.** Looking up and updating key state information on the platform is efficient. Our solution uses a balanced binary tree structure that requires only $\mathcal{O}(\log N)$ steps, where N denotes the number of AIKs. More critically, the integrity checks executed by the TPM are efficient, also requiring only $\mathcal{O}(\log N)$ steps.
- **Efficiency: AIK re-generation.** Hash-based signatures require generating fresh OTS keys post-provisioning since only a finite number of OTS keys can be created at set-up time. We amortize the key re-generation time by creating fresh OTS keys for a subsequent Merkle tree during signing.

2.1 System Model

We assume that the TPM is trusted and tamper-resistant, but the hosting platform is untrusted. The TPM's functionalities are augmented to include hash-based capabilities. Specifically, it can create a Merkle signature scheme key, integrity check a current AIK state, and sign a message using such an integrity-checked current key state. Additionally, its NVRAM is equipped with a small number of special registers, which we call Integrity Registers.

In today's implementation, the TPM's NVRAM is used for storing root keys for certificate chains, the EK (an RSA encryption key), the expected measurement of the machine launch state, and decryption keys used before the disk is

made available [14]. The minimum NVRAM size required by the TPM (version 1.2) spec is 1280 bytes [15]. We assume that the TPM RAM size is also fairly limited.

2.2 Merkle Tree Authentication

Merkle tree authentication is specified by two algorithms: a one-time signature (OTS) scheme and a Merkle signature scheme. Merkle signature schemes differ from each other in their traversal algorithms for constructing the "next authentication path" efficiently. We describe the Merkle signature scheme generically, intentionally hiding the details for the traversal algorithm for readability, and also because it is well-understood how the traversal time and storage size vary depending on the algorithm used, see [11].[2]

Moreover, real experimental results reveal that the overall performance of a tree authentication scheme (using a state-of-the-art traversal algorithm) is dominated by the performance of the underlying OTS scheme [21]. Given that our choice in OTS scheme greatly affects the practicality of our construction, we provide descriptions of several OTS schemes in the appendix. While the original Lamport-Diffie scheme (LD-C in this paper) was shown to be optimal in number of hash computations [22], LDWM is the usual go-to OTS scheme, because it decreases both the signature and storage sizes.

Within the context of this paper, $h(\cdot)$ is a cryptographic one-way hash function, and $r(\cdot)$ is a cryptographically secure pseudorandom number generator (PRNG) as defined below.

Definition 3 *A cryptographic hash function $h : \{0,1\}^* \to \{0,1\}^n$ maps arbitrary length strings to strings of length n, such that the following properties are satisfied:*

- *(Easy to compute) Given any $x \in \{0,1\}^*$, it is easy to compute its hash $y = h(x)$.*
- *(Pre-image resistance) Given any y in the image, it is computationally infeasible to find any x such that $h(x) = y$.*
- *(Second pre-image resistance) Given a $x_1 \in \{0,1\}^*$, it is computationally infeasible to find any $x_2 \neq x_1$ such that $h(x_2) = h(x_1)$.*
- *(Collision resistance) It is computationally infeasible to find* any pair *$x_1, x_2 \in \{0,1\}^*$ such that $h(x_2) = h(x_1)$.*

Definition 4 *A cryptographically secure pseudorandom number generator*

$$r : \{0,1\}^m \to \{0,1\}^n$$

is a function that generates an n-bit output from a truly random m-bit seed and satisfies the next-bit test: Given the first polynomial number of output bits, it is computationally infeasible to to predict the next bit of the output with probability non-negligibly larger than $\frac{1}{2}$.

[2] Merkle's original construction requires $\mathcal{O}(H^2)$ space and $\mathcal{O}(H)$ time [16], but recent constructions provide more efficient results [17–20].

Setup. A Merkle signature scheme (MSS) public key is equivalent to the result of the following computation. First we initialize a tree of constant branching factor D and height H and generate D^H OTS key pairs. The OTS key pairs are derived from the hash of the AIK name id, the re-generation number i, the leaf index ℓ, and the private-value index j: The AIK seed for AIK id is

$$s_{\mathrm{AIK}}(id) = h(id);$$

the seed for the i-th Merkle tree for id is

$$s_{\mathrm{MSS}}(id, i) = s_{\mathrm{AIK}}(id)|i = h(id)|i;$$

and the seed for the ℓ-th leaf of the i-th Merkle tree for id is

$$s_{\mathrm{LEAF}}(id, i, \ell) = s_{\mathrm{MSS}}(id, i)|\ell = h(id)|i|\ell.$$

The concatenation of a leaf seed and a private-value index is inputted into the pseudorandom number generator $r(\cdot)$ to create an OTS private key value, see appendix.[3] We store the hash of the ℓ-th OTS public key in the ℓ-th leaf of the tree, and the value of each non-leaf node is computed as the hash of the concatenation of its children's stored values. The MSS public key corresponds to the root value of this tree.

The approach above–computing values for all D^H leaves and subsequently hashing up the Merkle tree–is unnecessarily space consuming in practice. The convention is to keep a much smaller number of hashes in a stack, see [11].

Signing. Given a leaf in the Merkle tree, its authentication path is the sequence of sibling nodes along the path from it to the root. For example, the authenticating path for leaf 12 in Fig. 1(a) is $(\nu(11), \nu(6), \nu(1))$, where $\nu(i)$ denotes the hash value at node i.

An MSS signature is valid iff two conditions hold: the OTS signature $\boldsymbol{\sigma}$ verifies, and the OTS public key value $\boldsymbol{y}$ is consistent with the MSS public key Y and an authentication branch B. Thus, an MSS signature Σ is the quadruple:

$$\Sigma = (\ell, \boldsymbol{\sigma}, \boldsymbol{y}, B), \tag{1}$$

where ℓ denotes the leaf index number. The OTS public key $\boldsymbol{y}$ and signature $\boldsymbol{\sigma}$ can be computed from the OTS setup and signing algorithms, respectively. A traversal algorithm is implemented to compute the next authentication path B, and some key state information is saved to do this efficiently.

Because MSS setup is time-consuming, we recommend amortizing the cost of generating the next Merkle tree during signing. A single call to the signing algorithm should produce a signature, while doing a little bit of computation for incorporating *one* additional leaf in setting up the next tree, see Fig. 1(b). We

[3] In general, the OTS keys do not have to be derived from the same parent seed. We do so here to save storage space.

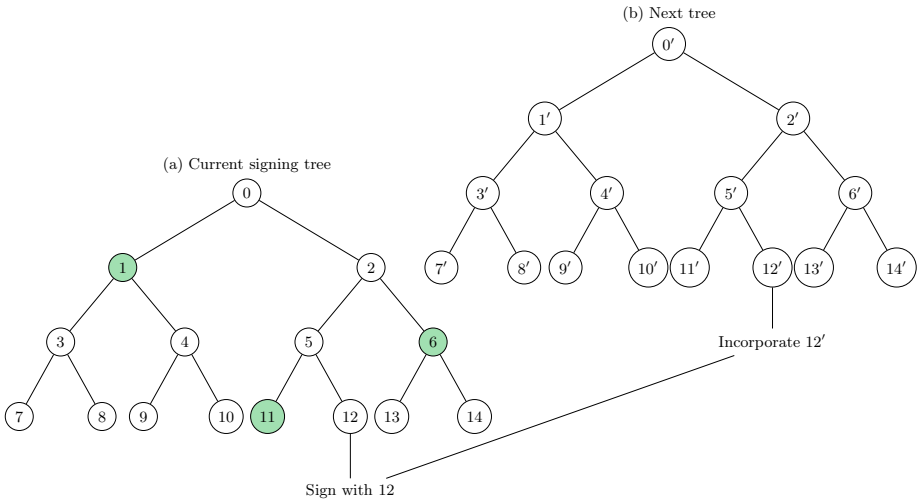

Fig. 1. (a) The nodes for node 12's authentication path are in green. (b) Computation for the next tree during is amortized during signing. During signing using leaf i, leaf i' for the next tree is created and incorporated into the next tree's stack.

could increase or decrease the key re-generation rate according to how fast the OTS keys are being used. This would require some other mechanism for determining the key usage rate and slight modifications to the `key_state` structure below.

The `key_state` object has the following fields:

- `ID` $\triangleq$ TPM signing key id
- `parent` $\triangleq$ parent TPM signing key id
- `iter` $\triangleq$ tree iteration or re-generation number
- `height` $\triangleq$ height of Merkle tree
- `bfactor` $\triangleq$ branching factor of Merkle tree
- `ots` $\triangleq$ OTS scheme used
- `lparam` $\triangleq$ loop parameter for the OTS scheme, see appendix
- `leaf` $\triangleq$ leaf number (for signing)
- `stack` $\triangleq$ stack for next tree
- `state` $\triangleq$ initial state for next tree
- `cur_state` $\triangleq$ saved state for updating the current tree's authentication path

`ID` stores the MSS key's unique id. `parent` stores the id of the key's parent. `iter` stores the re-generation number. `height` and `bfactor` store the tree's height and constant branching factor, respectively. `ots` and `lparam` store the OTS scheme and its corresponding loop parameter. `leaf` stores the leaf index.

`stack` stores the stack for setting up the next tree. `state` stores the initial saved state for the next tree, which includes the first authentication path. `cur_state` stores auxiliary state structures for tree traversal algorithms that are more optimal and may include the authentication path for the next signature. An instance of a `key_state` object is initialized with `iter` and `leaf` set to 1; `lparam` set to $\perp$; and with `state` and `cur_state` empty.

3 QUAntum Secure Hash (QUASH)

The MSS setup algorithm must be executed within the TPM. Otherwise malicious software may swap out the real public key with any value, including a MSS public key created from its choice for the seed. The MSS signing algorithm must also be executed within the TPM, since the purpose of having a TPM signing key is to prove (to some verifier) that some function (e.g., construction of a quote) was carried out by a trusted component.

However, hash-based digital signature schemes are space-intensive, requiring key state and producing large signatures. Thus, the design challenge is to offload the key state storage to the untrusted platform in a way that preserves security. We outline the main design challenges here:

- **Key state must be stored outside TPM:** A single `key_state` object for an AIK of branching factor 2 and height 20 takes 4 KB of space, see Table 1 in Sect. 4. To support even twenty such AIKs, either the TPM's NVRAM size must increase by an order of magnitude, or we must store the required key state information outside of the TPM in such a way that the overall solution remains secure.
- **Preventing rollbacks:** Rollbacks must be carefully managed, since the AIK key states are stored on the untrusted platform, and an OTS key is secure only if it is used once.[4]
- **Efficient integrity checks:** To prevent rollback attacks and issues from unintended data corruptions, we keep hash chains of the AIKs' key states in the protected space. These hashes are kept in special registers in the TPM's NVRAM and are used for integrity checks prior to key creation and signing.

  Since the integrity checks must be computed by the resource-limited TPM, the naive approach of computing the hash chain directly from the AIKs' key states is impractical; a more efficient method is needed.
- **Resiliency from data corruption:** Each TPM Integrity Register holds a compressed representation of a group of AIKs' key state information. Thus, damage to a single key state in the group could render all AIKs in the group useless. Our solution allows for recovery of AIKs and TPM registers where possible and minimizes the number of AIKs that are unrecoverable.

[4] Encrypting the AIKs' state information does not prevent rollbacks; an adversary could restore an AIK to a previous state by writing over its current encrypted state with a saved previous encrypted state.

3.1 Data Structures

State Storage Trees. The AIKs' state information is organized in balanced binary trees outside of the TPM. Each node in a State Storage Tree corresponds to an AIK and contains the information necessary for signing under this key. Specifically, each node stores a pair: a `key_state` object k (introduced in Sect. 2.2) and a subtree hash. The subtree hash is the hash of the concatenation of k and the subtree hashes of the node's children.[5]

Integrity Registers. Digests of last good key states are stored in the TPM's NVRAM in registers, which we call Integrity Registers.

Group Membership. The key states of a group of AIKs are hash-chained together to create a single hash digest. Thus, damage to any key state in the group can render all of the keys in this group useless. To mitigate this kind of data loss, an AIK belongs to multiple groups, where the hash chain of each group is stored in a separate Integrity Register. Given $(d \cdot I)$ Integrity Registers, each key state object belongs to I groups. Group membership is decided by the hash value of the key id. A key id is a member of group G_i^b if the i-th d-ary bit of its hash is b:

$$G_i^b = \{id : i\text{-th } d\text{-ary bit of } h(id) \text{ is } b\}. \tag{2}$$

For every group G_i^b, there is a corresponding State Storage Tree T_i^b for maintaining the state information of the AIKs in G_i^b and a corresponding Integrity Register R_i^b for storing the tree hash of T_i^b. We denote by $\texttt{trees}(id)$ a list of State Storage Trees to which id belongs, ordered lexiconically:

$$\texttt{trees}(id) = \{T_i^b : id \in G_i^b\}. \tag{3}$$

For example, if $d = 2$, $i = 2$, and the hash of id ends in 01, id belongs to groups G_0^1 and G_1^0. The State Storage Trees $\texttt{trees}(id) = (T_0^1, T_1^0)$ and corresponding Integrity Registers R_0^1 and R_1^0 are depicted in green in Fig. 2 below.

3.2 AIK Methods

We describe our methods for creating and certifying AIKs and signing under AIKs using the data structures described in the previous subsection. In Sect. 3.3, we show that our solution is tamper-evident and thwarts forgeries. Our practicality assessment of our solution is given in Sect. 4.

[5] The AIK seeds are never stored anywhere. The Primary Seed s_0, i.e., the "unsalted ESK seed," is the only seed stored in the TPM's NVRAM. A TPM signing key seed is generated from s_0 as needed and only in the protected space in the TPM.)

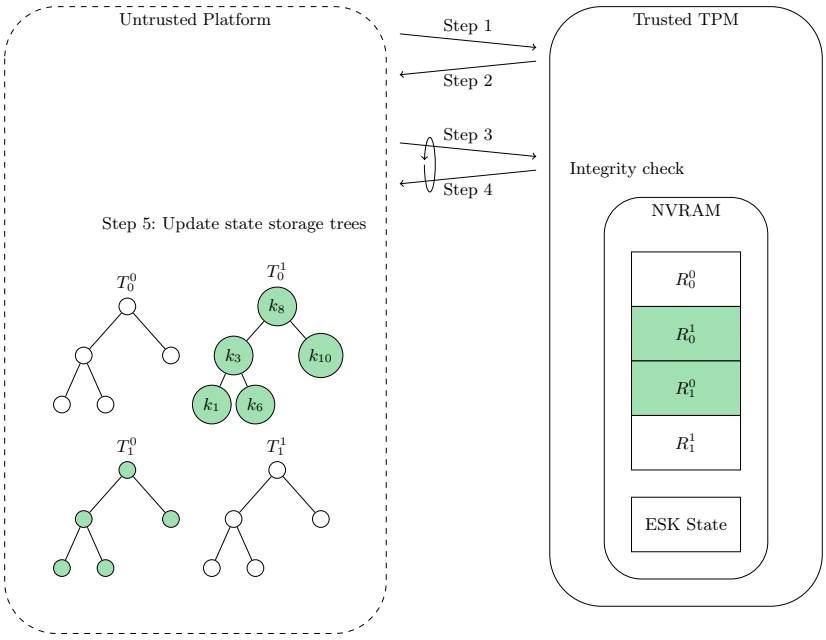

Fig. 2. State Storage Trees track AIK key state information. Integrity Registers in the TPM's NVRAM store integrity check values.

AIK Creation. To create a new identity key id,

1. The platform determines the list $\texttt{trees}(id)$ and sends a request to create an AIK to the TPM via a TPM interface, such as the TCG Software Stack (TSS). The request contains id, the parent id, the OTS scheme, and the Merkle tree height and branching factor.
2. Upon receiving the request, the TPM computes the list $\texttt{trees}(id)$ and sends an acknowledgement that the request was received to the platform.

Steps 3 and 4 are repeated for every $T_i^b \in \texttt{trees}(id)$:

3. The platform computes what the authentication path for node $h(id) \in T_i^b$ will be after the node is inserted into the tree. It sends this authentication path to the TPM.
4. To check the integrity of the authentication path, the TPM ensures that (i) the hashes along the authentication path are consistent with each other, and (ii) the hash at the root is equal to the value stored in the Integrity Register R_i^b corresponding to T_i^b.
   a. For the first authentication path that passes both integrity checks, the TPM instantiates a new `key_state` object k using the MSS setup method. The AIK seed s is determined from the parent signing key seed s_P and

id, e.g., $s = s_P||id$.[6] The TPM computes the hash of the key state k and returns k and the public key Y to the the platform.

   b. If both integrity checks pass, the TPM updates the Integrity Register R_i^b using the hash of k; otherwise, if either integrity checks fail, the Integrity Register is marked invalid.
   c. If T_i^b is the final tree in $\texttt{trees}(id)$: if a new key was successfully created, the public portion Y of the MSS key is signed by the ESK within the TPM. The TPM returns the ESK endorsement to the platform; otherwise, it returns an error.
   d. Send an acknowledgment to inform the platform that the round has ended, if nothing has been sent yet this round.
5. The platform updates all the trees in $\texttt{trees}(id)$ with the returned key state k.

Signing. The protocol for signing a message M using an identity key id is similar:

1. The platform determines the list $\texttt{trees}(id)$ and sends a sign-message request to the TPM. The request contains id and the message M.
2. Upon receiving the request, the TPM computes the list $\texttt{trees}(id)$ and sends an acknowledgement that the request was received to the platform.

Steps 3 and 4 are repeated for every $T_i^b \in \texttt{trees}(id)$:

3. The platform computes the authentication path for node $h(id) \in T_i^b$. It sends this authentication path and the key state k stored at node $h(id) \in T_i^b$ to the TPM.
4. To check the integrity of the authentication path, the TPM ensures that (i) the hash of k and the hashes along the authentication path are consistent with each other, and (ii) the hash at the root is equal to the value stored in the Integrity Register R_i^b corresponding to T_i^b.
   a. For the first authentication path that passes both integrity checks, the TPM signs the message M using the MSS signing method and k.
   b. If both integrity checks pass, the TPM updates the Integrity Register R_i^b using the updated MSS object k; otherwise, if either integrity checks fail, the Integrity Register is marked invalid.
   c. If T_i^b is the final tree in $\texttt{trees}(id)$: if M was successfully signed, the TPM returns the signature and the updated key state k' to the platform; otherwise, it returns an error.
   d. If the signature used the “last leaf" in the current Merkle tree, the public portion Y of the next MSS key is signed by the ESK and this newly created ESK endorsement is also sent to the platform.
   e. Send an acknowledgment to inform the platform that the round has ended, if nothing has been sent yet this round
5. The platform updates all the trees in $\texttt{trees}(id)$ with the returned key state k.

[6] If the parent key is not the ESK, this is determined recursively.

Signing a Key Handle. Each AIK must be certified as having been created by a legitimate TPM. This is done by signing each AIK public value using the ESK. The ESK itself is certified by the vendor, perhaps using a message authentication code (MAC) derived from the Primary Seed s_0.[7]

The ESK is a TPM hash-based signing key whose seed is derived from s_0 and some user-inputted salt. Both s_0 and the ESK state information is kept in the TPM's NVRAM and never leaves the protected space.

Key and Register Recovery. The TPM itself does not provide a mechanism for recovering data lost during storage, transmission, or processing. Even a low bit error rate can render all AIKs useless without some means of recovering AIKs (when possible) and reverting Integrity Registers to useable states. Here, we describe how this can be accomplished.

First, for each State Storage Tree, the platform computes the hash chain value from scratch (i.e., from the key states, as opposed to from the subtree hashes) and sends an AIK recovery request to the TPM along with the tree hashes. The TPM responds with the set B of State Storage Trees which did not pass the integrity checks and resets the corresponding Integrity Registers. (By resetting, we mean that the registers reflect the valid integrity value corresponding to an empty State Storage Tree.) For every tree $t \in B$, for every node $v \in t$, the platform checks if there exists a State Storage Tree $t' \notin B$, and $t' \in \texttt{trees}(id)$, where id is the id of v. Note that this can be computed efficiently from the hash of id. If t' exists, the platform marks the AIK as recoverable.

For every tree $t \in B$ and recoverable node $v \in t$, the platform sends to the TPM a request to add node v to t, along with an authenticating path for v in t and a proof of correctness for v (i.e., a tree $t' \notin B$, and $t' \in \texttt{trees}(id)$ and an authenticating path for v on t'). If the proof verifies, the TPM updates the corresponding Integrity Register accordingly; otherwise, the register is marked as invalid.

From above, we see that AIKs in damaged State Storage Trees can be recovered from undamaged State Storage Trees. An AIK id is recoverable iff there exists a State Storage Tree in $\texttt{trees}(id)$ that passes all integrity checks. In other words, an AIK is *unrecoverable* iff

$$t \in \texttt{trees}(id) \implies t \in B.$$

For a fixed number N of AIKs, increasing d and I decreases the conditional probability of an AIK residing in a damaged tree and the probability of an AIK being unrecoverable. However, this obviously increases the number of Integrity Registers in the TPM's NVRAM.

3.3 Correctness and Security Proofs

We provide sketches of correctness and security proofs for our construction here.

[7] Note that ESK may periodically require a fresh certificate.

Lemma 1 *If an authenticating path passes both integrity checks (step 4 for both AIK setup and signing), then the path does not contain any unauthorized modifications.*

Proof. If the hash value in the Integrity Register holds the intended value, then in order for an adversary to modify the path, it must do so in such a way that the final tree hash at the root retains this intended value. This is infeasible by the (second pre-image resistance) property of the hash function, see Def. 3.

Moreover, if an Integrity Register holds a valid value, it must be the intended value; since the Integrity Register is protected by the TPM, and the TPM updates the register with a valid value iff all integrity checks on the corresponding authenticating branch pass. (**Remark:** Thus when an integrity checks fails for a State Storage Tree T_i^b, the corresponding Integrity Register R_i^b must be marked invalid in order for Lemma 1 and Corollaries 1 and 2 to hold, see step 4b in AIK creation and signing.) □

Corollary 1 (Tamper-resistance) *Any unauthorized modification of an AIK's key state information (stored outside the TPM) is detectable.*

Corollary 2 (Unforgeability) *It is infeasible for an adversary to forge a signature under a properly certified AIK.*

Theorem 1 (Correctness) *Let id be any hash-based AIK with at least one authentication path which passes all three checks (for signing). Given any message M of any arbitrarily length, a signature on M under AIK id verifies.*

Proof. Let p be the first authenticating path that passes all three integrity checks. From Lemma 1, p does not not contain any unauthorized modifications. Thus, the AIK's key state is the intended key state, and so any signature under it verifies. □

4 Practicality Assessment

4.1 Space Analysis

We first determine how much space is needed for storing a key state object. The hash output size is assumed to be (256 bits or) 32 bytes. Eight of the `key_state` object fields—`ID`, `parent`, `iter`, `height`, `bfactor`, `ots`, `lparam`, and `leaf`—take up very little space. For concreteness, we have allotted 4 bytes for each of these fields; so $8 \times 4 = 32$ bytes are needed for storing all of these fields.

The maximum total storage size is dominated by the storage requirements for the remaining fields—`stack`, `state`, and `cur_state`—which depend on the traversal algorithm being used and the dimensions of the Merkle tree. We use the state-of-the-art traversal scheme from [20]; and our Merkle tree is binary with height $H =$ `height`.

During Merkle tree setup, an initial `state` consisting of H "authentication" nodes, $H-2$ "treehash" nodes, and a single "retain" node are stored [20], hence, $2H-1$ nodes. The `stack` for this computation requires a maximum of H nodes.

Additionally, the traversal algorithm requires storing at most $3.5H-4$ nodes in `cur_state` at any given step [20]. Thus, the total maximum space (in bytes) required for storing a key state object is bounded by $\Gamma(H) =$

$$32 + (32 \times (6.5H - 5)) = 208H - 128. \tag{4}$$

Table 1 shows the key state storage size for binary Merkle trees of various sizes.

		Targeted Range		
		$H = 15$	$H = 20$	$H = 25$
# OTS keys	2^H	$3 \cdot 10^4$	$1.05 \cdot 10^6$	$33.55 \cdot 10^6$
Storage size $\Gamma(H)$	$208H - 128$	3.0 KB	4.0 KB	5.1 KB

Table 1. Key state storage size in KB.

TPM Space for AIK Creation. Here, we provide an estimate for the TPM space needed to create an AIK. This estimation is meant to approximate the maximum space used by a space-efficient implementation.

In the analysis below, a unit is 32 bytes. For simplicity, we assume that the Merkle trees are binary of height H; we use the traversal algorithm described in [20] and the space-efficient [23] SHA-256 for the hash function; the space required for running the PRNG likewise is minimal; and the length of each OTS private x-value is 1 unit.

In creating a TPM signing key, the maximum resident TPM space is needed when the public key is being signed. We estimate the space required as follows:

- 1 unit for storing the public key (input);
- $\Gamma(H) = 6.5H - 4$ units for storing the ESK key state (input), see (4);
- 1 unit for storing the leaf index ℓ (output);
- $N(ots)$ units for the maximum OTS signature $\boldsymbol{\sigma}$ size (output);
- $N(ots)$ units for the maximum OTS public-key portion $\boldsymbol{y}$ size (output); and
- H units for the maximum authentication path size (output);

where the loop parameter $N(ots)$ depends on the OTS scheme, see appendix. For example, for LD-C, $N_C(ots) = 512$. At this point, we need not retain the input authentication path, the newly created key state, its hash, nor the updated register value. (**Remark:** In order to save TPM space, the newly created AIK state is returned to the platform before signing the retained public key value using the ESK state, see **AIK Creation** in Sect. 3.2.)

We need only negligible scratch space for running the MSS signing method: Only negligible additional space (beyond the space allotted for storing the output) is required for computing the OTS signature and public-key. Likewise, only negligible additional space (beyond space for storing the inputted ESK state) is required for updating `cur_state`, `state`, and `stack`.

Table 2 shows the maximum TPM space needed for creating an AIK of various sizes. Note that the total space required is not impacted by the number N of AIKs; (unless N is impractically large).

	LD-C	LD-Z	LDWM $w = 2$	LDWM $w = 4$	LDWM $w = 8$
$H = 15$	36.4	20.5	12.1	7.9	5.8
$H = 20$	37.6	21.7	13.3	9.1	7.0
$H = 25$	38.8	22.9	14.5	10.3	8.2

Table 2. TPM space required for AIK creation in KB.

TPM Space for AIK Signing. The maximum space in the TPM needed for signing a message occurs at the point when the message is being signed using the MSS signing method. The space requirement is identical to that of setup: The TPM retains the AIK state, but does not need the ESK state. It also needs to store the hash of M, but does not need to store a public key value.

4.2 Time Analysis

Since the State Storage Trees are balanced binary hash trees,

- Determining (the key state and) the authentication path takes $i \cdot \log(N)$ time, where N is the number of AIKs, and an id belongs to i groups.
- Determining the root value from (the key state and) the authentication path also takes $i \cdot \log(N)$ time.
- To update a State Storage Tree, only the nodes along a path to the root need to be reevaluated. Moreover, reevaluating a subtree hash requires only one hash operation. So, updating the State Storage Trees takes $i \cdot \log(N)$ time.

There are traversal algorithms that require $\mathcal{O}(H)$ space and $\mathcal{O}(H)$ time with low constant factors [18–20]. Therefore, by using an optimal traversal algorithm, the cost of MSS signing is dominated by the cost of executing the underlying OTS scheme once. Amortizing the re-generation cost only increases the signing time by roughly a factor of two.

The only time-inefficient step is MSS setup, which necessarily requires time proportional to the number of OTS keys created at setup time. For the ESK,

this can occur at provisioning time. For AIKs, we would like to set up on-the-fly, as needed. To mitigate this setup cost, we can setup a new AIK with a small Merkle tree and ramp up its size for subsequent tree(s).

5 Conclusion

We conclude that our construction for hash-based TPM signing keys is practical, if a state-of-the-art traversal algorithm, such as [20], is implemented.

QUASH can be implemented with a fairly small NVRAM: If an ESK Merkle tree includes roughly 6.5 million OTS key pairs, it can certify 100 AIKs, each requiring at most (2^{16} =) 65 thousand re-generations without needing to be re-certified by the vendor. This can be accomplished by setting a binary Merkle tree height to 23. Given these Merkle tree parameters, the maximum storage space required for storing the `key_state` object is 4.7 KB. Supporting 6 Integrity Registers would require an additional 192 bytes.

Acknowledgment

The authors would like to thank Anna Lysyanskaya for her suggestion on how to derive the ESK from a Primary Seed and Chris Eliopoulos Alicea, Joseph J. Ferraro, and John D. Ramsdell for helpful comments.

References

1. TCG, "TCG vendor ID registry," url: http://www.trustedcomputinggroup.org, Sept 2015.
2. A. Segall, "Trusted platform modules: When, why, and how to use them," version: June 21, 2015.
3. W. Arthur, D. Challener, and K. Goldman, *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security.* Apress, 2015.
4. S. Kinney, *Trusted Platform Module Basics: Using TPM in Embedded Systems.* Elsevier Inc., 2006.
5. V. Scarlata, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik, "Tpm virtualization: Building a general framework," in *Trusted Computing.* Springer, 2008, pp. 43–56.
6. G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.
7. B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *Security and Privacy (SP), 2010 IEEE Symposium on.* IEEE, 2010, pp. 414–429.
8. E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings of the 11th ACM conference on Computer and communications security.* ACM, 2004, pp. 132–145.
9. P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM journal on computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

10. L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 1996, pp. 212–219.
11. D. J. Bernstein, Ed., *Post-Quantum Cryptography*. Springer Berlin Heidelberg, 2009.
12. E. Alkim, N. Bindel, J. Buchmann, Ö. Dagdelen, and P. Schwabe, "Tesla: Tightly-secure efficient signatures from standard lattices," Cryptology ePrint Archive, Report 2015/755, 2015.
13. L. F. Sarmenta, M. Van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas, "Virtual monotonic counters and count-limited objects using a tpm without a trusted os," in *Proceedings of the first ACM workshop on Scalable trusted computing*. ACM, 2006, pp. 27–42.
14. D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn, *A practical guide to trusted computing*. Pearson Education, 2007.
15. B. Parno, J. M. McCune, and A. Perrig, *Bootstrapping Trust in Modern Computers*. Springer Science & Business Media, 2011, vol. 10.
16. R. C. Merkle, "A certified digital signature," in *Advances in Cryptology—CRYPTO'89 Proceedings*. Springer, 1990, pp. 218–238.
17. M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo, "Fractal merkle tree representation and traversal," in *Topics in Cryptology—CT-RSA 2003*. Springer, 2003, pp. 314–326.
18. M. Szydlo, "Merkle tree traversal in log space and time," in *Advances in Cryptology-EUROCRYPT 2004*. Springer, 2004, pp. 541–554.
19. P. Berman, M. Karpinski, and Y. Nekrich, *Optimal trade-off for Merkle tree traversal*. Springer, 2007.
20. J. Buchmann, E. Dahmen, and M. Schneider, "Merkle tree traversal revisited," in *Post-Quantum Cryptography*. Springer, 2008, pp. 63–78.
21. D. Naor, A. Shenhav, and A. Wool, "One-time signatures revisited: Have they become practical?" *IACR Cryptology ePrint Archive*, 2005.
22. B. Barak and M. Mahmoody-Ghidary, "Lower bounds on signatures from symmetric primitives," in *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*. IEEE, 2007, pp. 680–688.
23. K. Ideguchi, T. Owada, and H. Yoshida, "A study on ram requirements of various sha-3 candidates on low-cost 8-bit cpus," *IACR Cryptology ePrint Archive*, 2009.
24. D. McGrew and M. Curcio, "Hash-based signatures," url: https://tools.ietf.org/, 2013.

A One-Time Signature Schemes

The Lamport-Diffie Complement (LD-C), the Lamport-Diffie Zero Count (LD-Z), and the Lamport-Diffie with Winternitz improvement (LDWM) schemes were formally included in Merkle's 1979 paper introducing tree authentication [16]. All signatures are of digests, or hashes, of the original message. So to sign a message M of arbitrary length, a signing algorithm is run on the digest $d = h(M)$ of M.

Lamport-Diffie Complement. The Lamport-Diffie Complement (LD-C) scheme requires a private and public value pair (x, y) for each bit of the message digest d as well as each bit of the bit-wise complement $\bar{d}$ of the message digest. We assume that the message M is also hashed using the hash function $h(\cdot)$, and therefore the hash output length n is known and fixed. The input to the signing algorithm is the pre-computed loop-parameter $N_C = 2n$.

An LD-C key is a vector of pairs: Each pair consists of a pseudorandomly derived value (a private value x) and its hash (a public value y), see Alg. 1 below. $||$ denotes concatenation.

Algorithm 1: LD-C set-up algorithm

Data: A leaf seed s, a loop parameter $N_C = 2n$
Result: An LD-C public-key $\boldsymbol{y} = [y_1, y_2, \ldots y_{N_C}]$

```
for i = 1 : N_C do
    x_i = r(s||i);
    y_i = h(x_i);
return y;
```

Let d_i denote the i-th bit of a message digest d.

The signature of a message consists of a vector of N_C binary strings each of length n. For each $i \in [n]$, either the i-th string $\sigma_i = x_i$, and σ_{n+i} is a string of zeros (if $d_i = 1$); or σ_i is a string of zeros and $\sigma_{n+i} = x_{n+1}$ (if $d_i = 0$), see Alg. 2.

A verifier can check a signature by computing the hash of each non-zero block in the signature and comparing it against the corresponding public value in $\boldsymbol{y}$. The signature is valid iff all hashes equal their corresponding public value.

The signature scheme is secure since forging a valid signature requires finding an inverse of a hashed value (a public value). Note that including the complement of the message digest prevents a man-in-the-middle attack.

Lamport-Diffie Zero Count. In LD-C, the bit-wise complement of the message digest is used to determine the signature. Instead of the complement, Lamport-Diffie Zero Count (LD-Z) includes a zero count of the message digest bits to thwart forgery. Thus, the set-up algorithm is identical to Alg. 2, with the exception that the inputted loop parameter is $N_Z = n + \log n$, where $\log(\cdot)$ is the logarithm base 2.

Algorithm 2: LD-C signing algorithm

Data: A message digest d, a leaf seed s, and a set-up loop parameter N_C
Result: An LD-C signature $\boldsymbol{\sigma}$
`/* allocate signature structure with all zeros */`
Set $\boldsymbol{\sigma}$ as a length-N_C vector of length-n zero-blocks;
for $i = 1 : N_C/2$ **do**
    if $d_i = 1$ **then**
        $\sigma_i = r(s||i)$;
    else
        $\sigma_{n+i} = r(s||n+i)$;
return $\boldsymbol{\sigma}$;

To sign a message digest d, we count the number of zeros in d and concatenate the zero count to d, so that

$$d' = d||z, \tag{5}$$

where z is the $\log(n)$-bit binary representation of the zero count. Then, the signature $\boldsymbol{\sigma}$ is a length-N_Z vector of blocks, each of length n, where the i-th block is either x_i (if $d' = 1$) or a string of zeros (if $d' = 0$), see Alg. 3.

Algorithm 3: LD-Z signing algorithm

Data: A message digest d, a leaf seed s, and a loop parameter N_Z
Result: An LD-Z signature $\boldsymbol{\sigma}$
Compute the zero count z of d, and determine $d' = d||z$;
`/* allocate signature structure with all zeros */`
Set $\boldsymbol{\sigma}$ as a length-N_C vector of length-n zero-blocks;
for $i = 1 : N_Z$ **do**
    if $d_i = 1$ **then**
        $\sigma_i = r(s||i)$;
return $\boldsymbol{\sigma}$;

As before, a verifier can check that the signature hashes are the expected y-values. In this scheme, the zero count prevents man-in-the-middle attacks.

Winternitz Improvement. Whereas an LD-C key is of length N_C blocks, an LD-Z key takes up only N_Z blocks of size equal to the LD-C key blocks. The Winternitz improvement (LDWM) further reduces the key length by signing w bits at a time, where w is the Winternitz parameter. Typical values of the Winternitz parameter are either 2, 4, or 8 [24].

An LDWM key pair consists of a length-N_W vector of value pairs, where

$$N_W = \left\lceil \frac{n}{w} \right\rceil + \left\lceil \frac{\lfloor \log \left(\left\lceil \frac{n}{w} \right\rceil \right) \rfloor + w + 1}{w} \right\rceil, \tag{6}$$

and where each value pair (x_i, y_i) consists of a pseudorandom value x_i and a hash value y_i from applying $h^{2^w-1}(\cdot)$ to x_i, see Alg. 4 below. Here the exponent $(2^w - 1)$ in the hash function means iterating the hash function $(2^w - 1)$ times. As before, we assume that N_W has been computed by a parent function and is passed to the LDWM set-up algorithm.

Algorithm 4: LDWM set-up algorithm

Data: A leaf seed s, a loop paramter N_W and the Winternitz parameter w
Result: An LDWM public-key $\boldsymbol{y} = [y_1, y_2, \ldots y_{N_W}]$

```
for i = 1 : N_W do
    x_i = r(s||i);
    y_i = h^{2^w - 1}(x_i);
return y;
```

To sign a message digest d: we first pad d with zeros so that its length is an even multiple of w and compute a checksum c (in binary) as follows

$$c = \sum_{i=1}^{N_W} (2^w - d'), \tag{7}$$

where d' denotes the padded d. Then we pad c with zeros so that its length is also a multiple of w. We denote by d'', the concatenation of the padded d with the padded c; so

$$d'' = d'||c', \tag{8}$$

where c' denotes the padded c. For every $i \in [N_W]$, we compute $\sigma_i = h^{b_i}(r(s||i))$, where b_i denotes the decimal representation of the i-th length-w block of d''. The final LDWM signature $\boldsymbol{\sigma}$ is $[\sigma_1, \ldots, \sigma_{N_W}]$, see Alg. 5 below.

A signature can be verified by checking that each σ-value hashed an appropriate number of times equals the corresponding public y-value. The checksum prevents forgeries. We refer the reader to [11, 16] for the security proofs.

B Requirements for OTS Schemes

While Merkle describes LD-Z as an improvement over LD-C since it requires less space, and LDWM as a further improvement; we include our own detailed (non-asymptotic) time-space analyses of these schemes here. We derive the resource requirements for each of the OTS schemes provided in Sect. A and the overall

Algorithm 5: LDWM signing algorithm

Data: A message digest d, a leaf seed s, the loop parameter N_W, and the Winternitz parameter w
Result: An LDWM signature $\boldsymbol{\sigma}$

Determine d''' from d;
for $i = 1 : N_W$ **do**
  $\sigma_i = f^{b_i}(r(s||i))$;
return $\boldsymbol{\sigma}$;

Merkle tree authentication provided in Sect. 2.2 with respect to these primitives. Let m denote the length of an OTS scheme private value, and let n denote the length of a hash digest. The time and space requirements for (i) the PRNG function to obtain an OTS private value and (ii) hashing an OTS private value are given below:

- The PRNG $r(\cdot)$ has output size m, scratch space complexity S_r, and time complexity T_r. We assume that $S_r \geq N_C = 2n$.
- Hashing an OTS private value x to a digest $y = h(x)$ produces an output of size n and has scratch space and time complexities S_x and T_x, resp.

While the time and space complexities of these sub-routines are really functions of m and / or n, we will write them as constants for readability. For example, it will be understood that the time requirement T_x for hashing an OTS private value to a digest is really a function $T_x(m)$ of m.

	Set-up Time			**Set-up Space**
	T_r *steps*	T_x *steps*	*add.*	$\max(S_r + m, S_x + n) +$
LD-C	N_C	N_C	$\mathcal{O}(N_C)$	$(N_C - 1)\, m$
LD-Z	N_Z	N_Z	$\mathcal{O}(N_Z)^{\dagger}$	$(N_Z - 1)\, m$
LDWM	N_W	$(2^w - 1)\, N_W$	$\mathcal{O}(N_W)$	$(N_W - 1)\, m$
	Signing Time			**Signing Space**
	T_r *steps*	T_x *steps*	*add.*	
LD-C	n	–	$\mathcal{O}(n)$	$N_C \cdot m + S_r$
LD-Z	$\frac{N_Z}{2}$	–	$\mathcal{O}(N_Z)^{\dagger}$	$N_Z \cdot m + S_r$
LDWM	N_W	$\left(2^{w-1} - \frac{1}{2}\right) N_W$	$\mathcal{O}(n)$	$(N_W - 1)\, m + \max(S_r + m, S_x + n)$

Table 3. Requirements for OTS schemes. †Add. steps required for the zero count.

B.1 LD-C and LD-Z

We first determine the resource requirements for the LD-C set-up and signing procedures provided by Algs. 1 and 2 in Sect. A.

LD-C Set-up Requirements. Each iteration of the for-loop in Alg. 1 requires T_r steps for generating a random x-value, T_x steps for hashing the x-value, and some marginal constant time for checking the loop condition. So, a total of $\mathcal{T}_C = N_C(T_r + T_x + \mathcal{O}(1))$ steps are required for the total set-up time.

We now compute the space requirement. At the i-th step of the for-loop, we need to hold the values of the previously calculated y-values; this requires $(i-1)n$ bits. For obtaining x_i, we need a scratch space of size S_r and an additional m bits allocated for the output of $r(\cdot)$. For obtaining y, we need a scratch space of S_x (which subsumes the m bits for holding the parameter x_i) and an additional n bits allocated for the output of $y_i = h(x_i)$. Thus the maximum space required for the set-up phase is given by $\mathcal{S}_C = (N_C - 1)n + \max(S_r + m, S_x + n)$, plus some negligible space for bookkeeping.

LD-C Signing Requirements. For every $i \in [n]$, either the i-th pseudorandom x-value is recomputed (if $d_i = 1$), or the $(n + i)$-th pseudorandom x-value is recomputed (if $d_i = 0$). Either way, an x-value is computed per iteration of the for-loop. Thus, the signing time requirement is $T_C = n(T_r + \mathcal{O}(1))$. The total space required is given by the sum of the space allocated for the signature ($2mn$ bits) and the scratch space for re-evaluating the x-values serially (S_r bits), plus some negligible space for bookkeeping: $S_C = 2mn + S_r$. See the first row of Table 3 for LD-C's time-space requirements.

LD-Z Requirements. Whereas an LD-C public key includes a key-value for each bit of the complement of the message digest, LD-Z includes a key-value for each bit of the zero count. Thus, by a nearly identical analysis as above, the set-up time $\mathcal{T}_Z$ and space $\mathcal{S}_Z$ requirements for LD-Z are:

$$\mathcal{T}_Z = N_Z(T_r + T_x + \mathcal{O}(1)) \tag{9}$$

$$\mathcal{S}_Z = (N_Z - 1)n + \max(S_r + m, S_x + n). \tag{10}$$

Deriving the signing requirements requires a subtly different analysis. First, we must evaluate a zero count in LD-Z (which is an unnecessary step in LD-C). Moreover, while we re-evaluate exactly one pseudorandom value (either x_i or x_{n+i}) for each $i \in [n]$ in LD-C; in LD-Z, we must check the zero count bits separately, and at every check there is some probability of not re-evaluating a pseudorandom value. Thus, signing requires $\mathcal{O}(n)$ steps for determining the zero count and an additional $N_Z(T_r + \mathcal{O}(1))$ steps for the for-loop in the worst case; and an average of

$$T_Z = N_Z\left(\frac{T_r}{2} + \mathcal{O}(1)\right) + \mathcal{O}(n) \tag{11}$$

$$= \left(\frac{N_Z}{2}\right) T_r + \mathcal{O}(N_Z) \tag{12}$$

steps assuming that the message digest is uniformly distributed over n-bit strings. The space requirement for signing is $S_Z = N_Z + \mathcal{S}_r$; the space required for computing the zero count is subsumed by the space required to compute an x-value. See the second row of Table 3 for LD-C's time-space requirements.

B.2 Winternitz Improvement

Set-up Requirements. Computing the public key using the Winternitz improvement consists of computing N_W hash digests, where

$$N_W = \left\lceil \frac{n}{w} \right\rceil + \left\lceil \frac{\lfloor \log \left(\left\lceil \frac{n}{w} \right\rceil \right) \rfloor + w + 1}{w} \right\rceil$$

and w denotes the Winternitz parameter; and where each digest is computed from generating a pseudorandom value and iterating the hash function $(2^w - 1)$ times on the pseudorandom value. Thus, the required set-up time for LDWM is

$$\mathcal{T}_W = N_W \cdot (T_r + (2^w - 1)T_x + \mathcal{O}(1))$$

Compared with LD-C or LDWM, the number of iterations reduces by roughly a factor of w; however, the dominating component is the number of hash computations, which increases considerably by an exponential factor in w. On the other hand, the set-up space requirement reduces roughly by a factor of w; the total set-up space required is

$$\mathcal{S}_W = (N_W - 1)\, n + \max(S_r + m, S_x + n)$$

plus some marginal space for bookkeeping.

Signing Requirements. To sign a message digest, we first pad the message digest, calculate the checksum, and then pad the checksum; this takes $\mathcal{O}(n)$ time.

The signature consists of N_W hash digests. Each digest is computed by iterating the hash function some b times on either a message digest block or a checksum block, where $0 \leq b \leq 2^w - 1$. Thus, the expected signing time requirement is

$$T_Z = N_W \left(T_r + \left(2^{w-1} - \frac{1}{2} \right) T_x + \mathcal{O}(1) \right) + \mathcal{O}(n) \tag{13}$$

$$= N_W T_r + N_W \left(2^{w-1} - \frac{1}{2} \right) T_x + \mathcal{O}(n) \tag{14}$$

The space required is $S_W = (N_W - 1)\, n + \max(S_r + m, S_x + n)$. See the third row of Table 3 for LD-C's time-space requirements.

We make the simplifying assumption that the checking a loop-condition, counting zeroes, and performing the Winternitz operation take marginal time and space. Comparing the resource requirements across the OTS schemes, we find that LD-Z outperforms LD-C both in setting up a key and signing a message, for all m and n. Comparing LDWM against LD-Z, the time requirements increase by an exponential factor in w, while the space requirements reduce by a linear factor in w.