CRYPTO4A

WHITE PAPER

Sectorization

Addressing HBS' Shortcomings

Introduction

This whitepaper introduces the basic ideas behind hash-based signatures (HBS), discusses their suitability to use in a firmware- or code-signing application, and describes some of their limitations in a real-world setting in regards to disaster recovery and survivability. All of this is in preparation for the introduction of the proposed use of sectorization as a means of addressing some of HBS' shortcomings.

We've tried to keep things as simple as possible by simplifying our examples, deferring to other excellent articles for additional details related to HBS, and choosing to limit descriptions to a particular choice of HBS, but all of the concepts described herein are generic enough that they can be applied to other, potentially more complex HBS instantiations. We've also added a number of asides that attempt to provide a bit more detail regarding some aspects of the discussion which will (hopefully!) assist the curious reader. In addition, we're happy to address any questions the reader may have via e-mail using the author's address: *jimg@crypto4a.com*.

Hash Based Signature (HBS) 101

In 1979 Ralph Merkle submitted a Ph.D. dissertation [1] in which he described a means to efficiently generate certified signatures using one-way hash functions as the underlying mechanism. The basic idea had already been invented by Leslie Lamport, but that original solution required a very large private AND public key be generated and published for EVERY message that was to be signed. Merkle's principal contribution to HBS was the Merkle tree which provided an efficient means to bind together a large number of one-time signatures under a single public key, which greatly improved the usability of HBS.

Merkle's approach utilizes 2^n one-time signature (OTS) instances whose public keys are hashed together via a binary tree to generate a single public key that binds all of the OTS instances together, thereby allowing you to generate 2^n signatures from a single public key. This is then combined with a pseudo-random method for generating the private keys of each of the OTS signatures from a single reasonable-length secret seed value (e.g., 32 bytes), to yield a reasonably compact public/private keypair, and a many-time signature (MTS) scheme. Signatures consist of the OTS signature on the message, combined with the information necessary to re-compute the public key which is the sequence of values required to climb the binary tree from the OTS instance at the leaf of the tree, all the way up to the root of the tree (i.e., the public key). This is shown in a greatly simplified fashion in Figure 1.

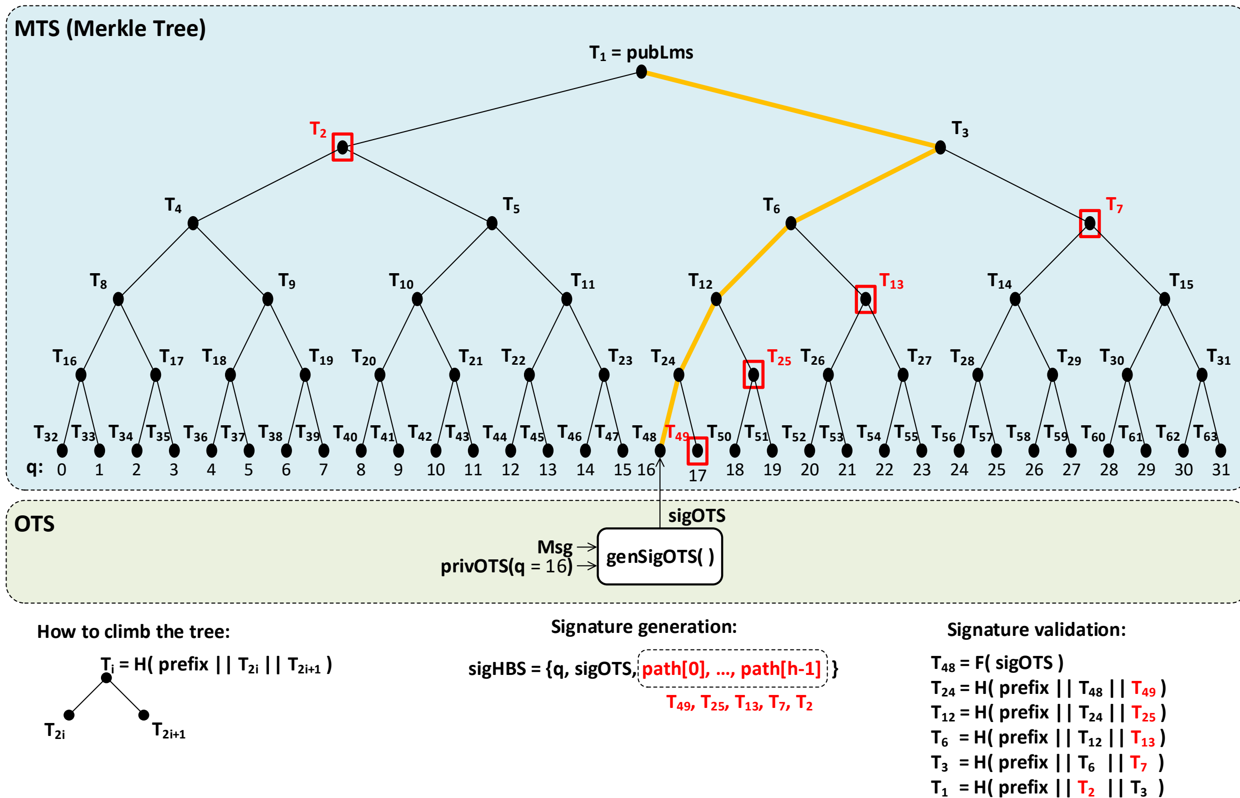


Figure 1. Greatly simplified HBS signing operation

Unfortunately, this introduces some constraints on practical implementations. One of these constraints is related to performance. Computing the public key requires you to compute 2^n OTS public keys, and then combine those results via a n-level binary tree which requires you to compute $2^n - 1$ node values. This can quickly prove prohibitively expensive as n increases.

Another important constraint is related to security. If you were ever to re-use an OTS instance then you greatly increase the risk of having an adversary generate undetectable forgeries from previously-generated signatures. Each instance corresponds to one of the 2^n leaves of the Merkle tree so we need to track which instances have been used already. We track this using a monotonically increasing counter that counts up from 0 to $2^n - 1$, incrementing after each OTS is generated. This count value (i.e., leaf number) is referred to as the state of the HBS, and its management is absolutely crucial to ensuring the security of the HBS scheme.

Fast forward almost 40 years and HBS has evolved into a mature cryptographic mechanism that utilizes optimized OTS schemes and innovative hierarchical hyper-tree structures to address the computational complexity of computing signatures and HBS public keys in a timely fashion. As an added bonus, HBS are considered to be safe from the threat posed by quantum computers, which will completely undermine the security of traditional RSA/ECDSA-based certified signature schemes. Hence, there is a lot of interest in HBS at this time for certain use cases that it is well

suited for (e.g., firmware signing). In addition, work has been done to create stateless HBS schemes (e.g., SPHINCS [2] and its successor SPHINCS+ [3]) that employ a few-time signature (FTS) scheme to provide resiliency against the state reuse problem, effectively rendering it a non-issue, though at the cost of additional complexity and signature size. This whitepaper however is focused only on the stateful variants.

For those readers who want to know more regarding HBS then I definitely recommend you check out Adam Langley's explanation of HBS from his online blog [4] as he does an excellent job filling in the details in regards to what HBS is and how it is used. More curious (and brave!) readers can even go so far as to review the two primary stateful HBS schemes' informational RFCs ([5], [6]) which serve as the definitive references for these variants.

In the interest of simplicity, the remainder of this whitepaper will focus on the LMS-HBS method defined in RFC 8554 [5], though everything discussed should be realizable using XMSS [6] as well.

The Winternitz One Time Signature (WOTS) Scheme

All HBS methods rely on the use of OTS schemes, and one particularly popular OTS is the Winternitz OTS (WOTS) which is used in one form or another for both LMS-HBS [5] and XMSS [6]. At its heart, the WOTS scheme utilizes hash chains to compute a digital signature by chunking a hash of a message (plus some additional information that we're ignoring here for simplicity's sake) that is to be signed into $\boldsymbol{u}$ digits, each of length $\boldsymbol{w}$ bits ($\boldsymbol{w}$ = Winternitz parameter which is essentially the radix of the encoding). Each of those digits ($digit_i$) is then used to compute the length of a hash chain using the formula $len_i = digit_i$, where each len_i determines how many times element x_i of the WOTS private key is iteratively hashed to yield the WOTS signature element y_i.

Note that the WOTS public key elements (z_i) are computed using the same iterative hashing approach with a fixed chain length of $2_w - 1$ (i.e., $z_i = H^{2^w-1}(x_i)$) for all elements of the WOTS private key. These z_i are then concatenated and hashed together to yield the WOTS public key value that is used in the WOTS signature verification. A simplified version of the calculation of the WOTS public key is shown in Figure *2*.

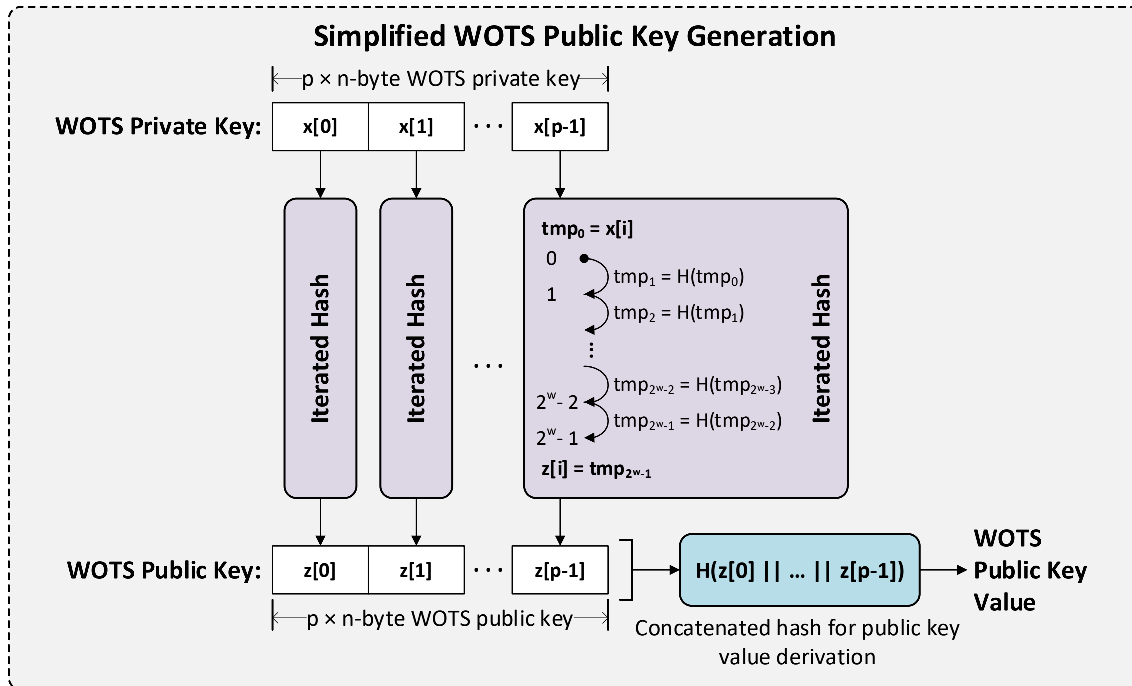


Figure 2. Simplified WOTS Public Key Generation

Unfortunately, this approach is prone to forgery in that given some y_i, an attacker can easily perform additional hash iterations to yield $y_i' \neq y_i$ which appears valid, but maps to a different len_i, and hence $digit_i$. WOTS mitigates this attack vector by computing a checksum across the $\boldsymbol{u}$ digits using the formula:

$$checksum = \sum_{i=0}^{u-1} 2^w - 1 - digit_i$$

The checksum is then encoded as $\boldsymbol{v}$ additional radix-w digits that get signed as well, giving you $\boldsymbol{p} = \boldsymbol{u} + \boldsymbol{v}$ digits in all that need to be encoded/signed. Hence, an attacker that performed additional hash iterations would effectively be increasing the value of some $digit_i$, which in turn would reduce one or more checksum digits, thereby requiring them to generate a result from a hash chain with less iterations. This is equivalent to them being able to find a pre-image of a cryptographically secure hash function (e.g., SHA256) which is thought to be computationally infeasible at this time. A simplified version of the calculation of the WOTS signature is shown in Figure *3*.

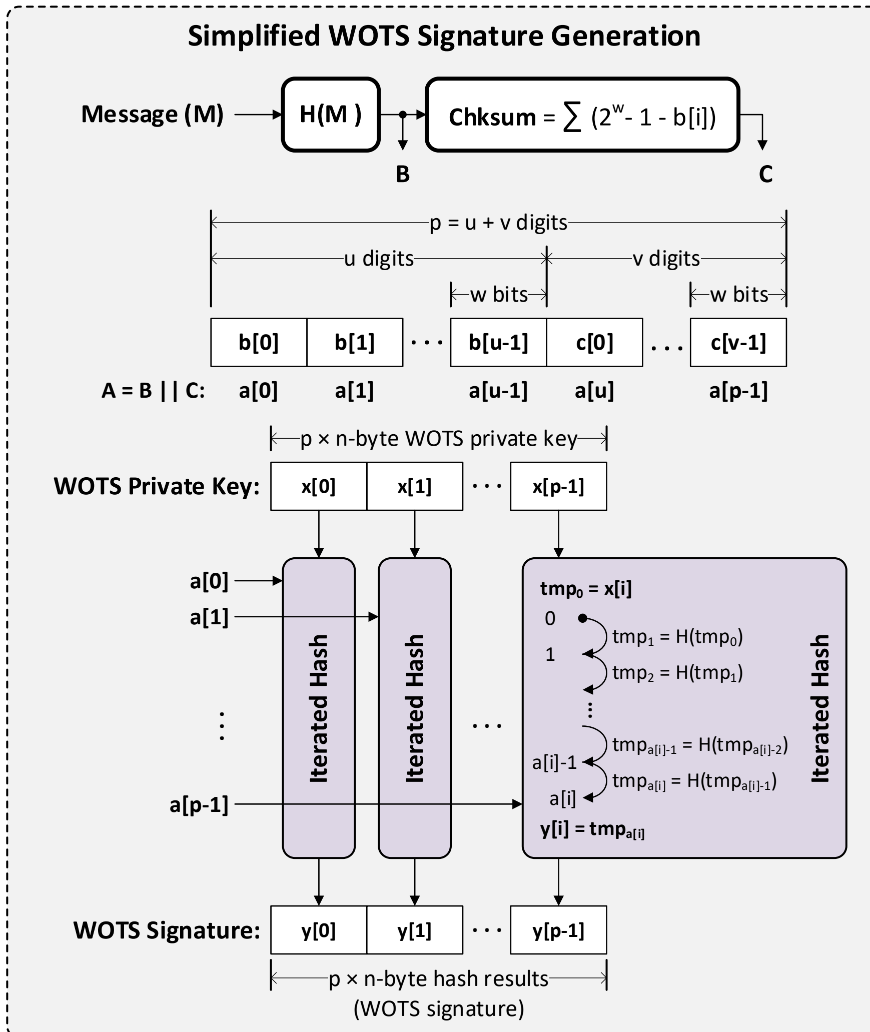


Figure 3. Simplified WOTS Signature Generation

Verification of a WOTS scheme repeats the checksum computation and encoding of the resulting concatenated message hash and checksum into $\boldsymbol{p} = \boldsymbol{u} + \boldsymbol{v}$ digits ($digit_i$). However, these digits are now used to complete the hash chains by performing $len_i = 2_w - 1 - digit_i$ iterations on the $\hat{y}_i$ values found in the WOTS signature (the hat character represents the fact the values may differ from the values computed during signature generation). If the $digit_i$ values are the same as those used during the signing process, then you are essentially re-computing the public key components z_i as:

$$H^{2^w-1-digit_i}(y_i) = H^{2^w-1-digit_i}\left(H^{digit_i}(x_i)\right) = H^{2^w-1}(x_i) = z_i$$

If $digit_i$ has been modified in some way then $\hat{y}_i \neq y_i$ and you will be computing the $\hat{z}_i \neq z_i$, leading to a different concatenated hash value, and a simple comparison to the WOTS public key

value will reveal the discrepancy so that you can reject the signature. A simplified version of the calculation of the WOTS signature is shown in Figure *4*.

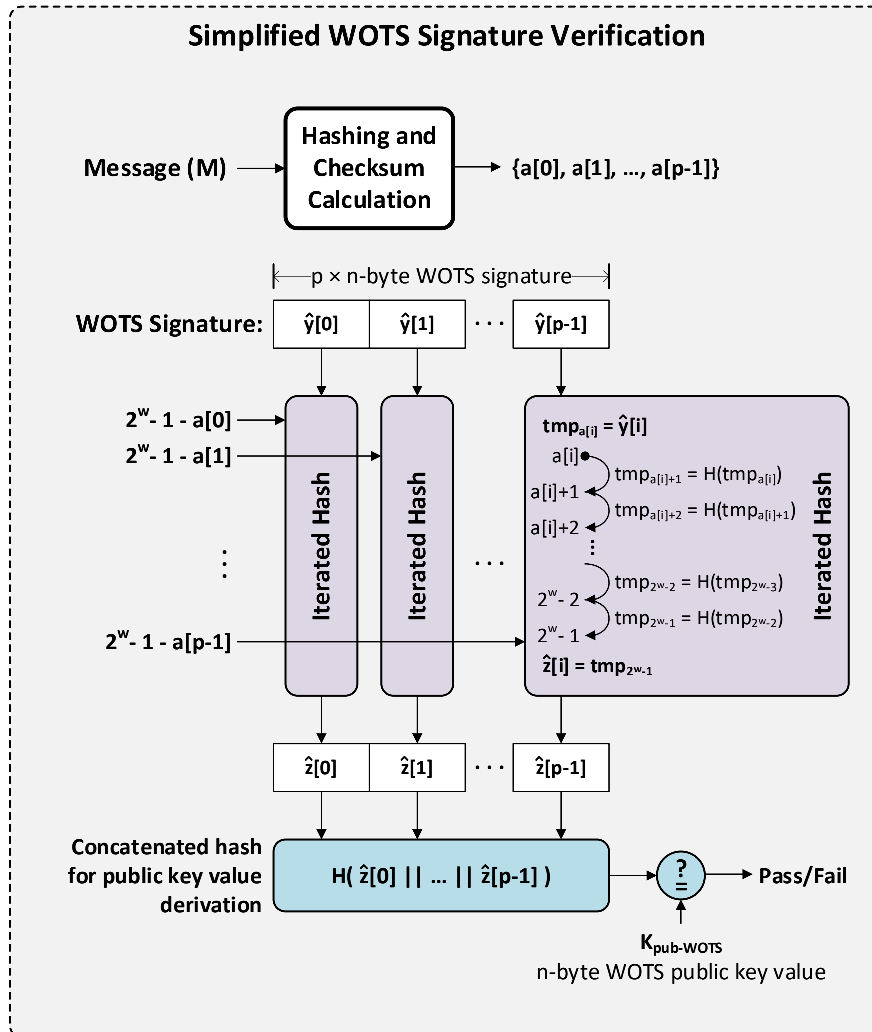


Figure 4. Simplified WOTS Signature Verification

Suitable Use Cases

HBS isn't a panacea. It has a number of limitations that make it ill-suited to use as a generic digital signature mechanism that can be a drop-in replacement for today's RSA- and ECDSA-based digital signature mechanisms. First and foremost is the stateful nature of HBS, which we have already touched on in our brief introduction to HBS, and which we will discuss further shortly when we look at the difficulties addressing real world requirements such as Disaster Recovery (DR). Another big constraint on HBS is the fact we need to know how many signatures will be required at the time of key generation, which may be impossible to do. Estimating conservatively (i.e., over-estimating to be safe) leads to increased signature sizes for a mechanism that already has relatively large signatures. Hence, care must be taken to identify appropriate use cases where HBS makes sense.

One such use case is firmware signing where we can (relatively) accurately predict the total number of signatures required based on the expected update schedule and lifetime of the firmware we're trying to protect, and for which a few kB's of signature are insignificant compared to the MB of firmware we're attaching the signature to. In addition, the quantum safe nature of HBS ensures that when you want to update your devices' firmware to include new features/functions (e.g., in response to NIST's designation of quantum-safe signature mechanisms in the 2021-2022 timeframes), it can be done so in a safe and secure way, even in the presence of adversaries in possession of a quantum computer, which addresses the bootstrapping problem present on many companies' plans to firmware update their way out of the quantum computing dilemma.

Another such use case are extremely long-lived root keys where we need to be able to generate signatures over 20+ year lifetimes, albeit on a relatively infrequent basis that can be reasonably estimated to ensure we can generate a suitable HBS tree structure.

The bootstrapping dilemma

A number of companies claim that they will just utilize their built-in firmware update procedure to introduce quantum-safe cryptographic primitives if and when they are standardized, and when they become necessary (i.e., when quantum-based attacks become a reality). Setting aside the record-and-decrypt-later aspect of things, and focusing just on the authentication aspect, this introduces some troubling issues. First, how will we know that a quantum-based attack is possible? Given the expected capabilities of a quantum-enabled adversary, and the level of resources required to develop that capability, it is very likely its existence will never be avowed until well after it is achieved, much like the Allies' development of ULTRA during the Second World War, which didn't come to light until more than 30 years later. Hence, we will never likely know the precise timing of that event, so all firmware update contingencies based on non-quantum safe authentication mechanisms are at risk, with that risk increasing the longer people choose to delay the switchover to quantum safe mechanisms. And once that capability is known to exist, its already too late. Any attempted firmware update from that moment onwards cannot be trusted as it may have been modified to introduce mechanisms to subvert any post-update authentication mechanisms since the update's integrity can't be guaranteed. So, in our opinion its best to act now to introduce proven, quantum-safe authentication mechanisms into the firmware update process as soon as possible, especially since well-suited options such as HBS already exist, and are standardized sufficiently for vendors to develop proprietary update mechanisms that don't require interoperability.

Durability Protection and Disaster Recovery

Real-world systems require the ability to deal with unexpected events that effectively remove devices from operation, rendering its resources and services unavailable. Sometimes these events lead to temporary outages (e.g., local power outage or network failure), and sometimes they lead to more permanent outages (e.g., a fire destroys the device, or the device simply permanently fails). Typically, a PKI provider will maintain a backup device, which contains the same private keys as the primary device, in a geographically distinct location to provide survivability to localized failures such as network/power outages. When the primary device goes down the backup device can be brought online to fill in until (if!) the primary device

can be restored/repaired. The goal is to do this in a manner that minimizes the downtime and impact on users.

HBS throws a wrench in the works of traditional DR mechanisms as the HBS private keys are stateful, which means they are modified during each usage so any backup device's copy of the private key needs to maintain coherency with the primary device's private key so that it NEVER re-uses a prior private key state. If we can't guarantee that behavior due to even a brief lack of synchronization, then the entire system is at risk of being compromised as attackers may have enough information to successfully forge messages! Indeed, even a single reuse can render the system compromised.

Needless to say, this is a very difficult problem to mitigate using standard HBS mechanisms and techniques, which is why we're proposing the sectorization concept as a means of effectively dealing with the limitations of stateful HBS in a DR-based scenario.

Introducing Sectorization

Sectorization is a means to partition the 2^n signatures generated by an HBS into 2^S cryptographically-isolated segments[1] (each capable of generating 2^{n-S} signatures) such that a device assigned sector i's private key cannot generate valid signatures for sector j, and vice-versa, provided i ≠ j. Note that both sector's utilize the same public key, so they are all part of the same signing authority, and in the hierarchical $XMSS^{MT}$/HSS variants, they will share the same top-level tree identifiers too. This isolation is provided by using a more elaborate key generation mechanism that knows a priori how many sectors are to be generated, and then proceeds to generate unique sector seed[2] values for each of the sectors using either an approved pseudo-random generation method from a master seed value that mixes in the unique sector numbers, or by generating unique random sector seed values. In either case, the sector seed becomes the seed of the sector's top-level tree[3] , and the same top-level tree identifier is used for all sectors. Once that is done, it's a simple matter to use each sector's seed to generate that sector's OTS private keys and public keys, and then combine those public key values together to ascend the top-level tree and generate the HBS public key.

Where the sectorized approach differs from a conventional HBS is that we need to augment the private key to include the off-path information that will allow a verifier to ascend the tree from the root of their sector up to the root of the tree (i.e., HBS public key). This amounts to storing S

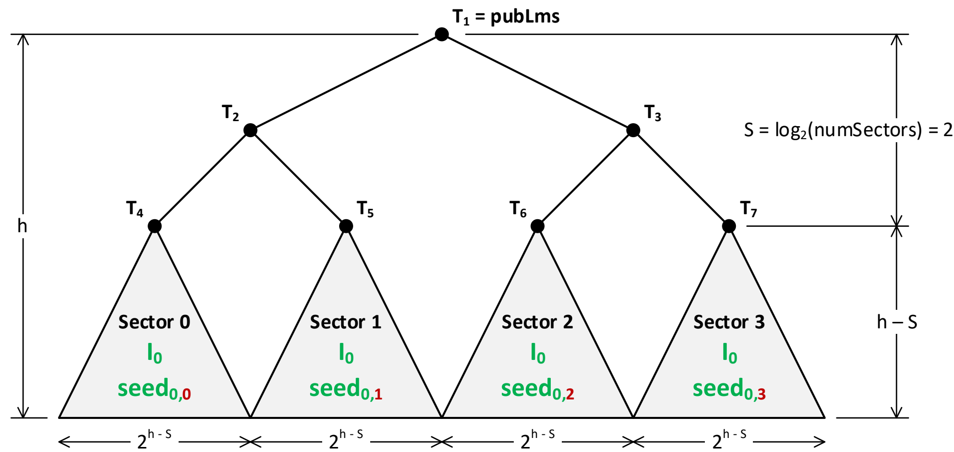
---

[1] Note that we are restricting ourselves to power-of-two sector counts that fit within the range of the top-level tree in order to simplify the explanation/discussion. There is no fundamental limitation on the total sector count, though a power of two greatly simplifies the implementation, as does ensuring the sectorization splitting all falls within the top-level tree. Examples where this is not the case are left as exercises for the reader! :->

[2] Recall that seeds are used in the pseudo-random OTS private key value generation process such as that proposed in Appendix A of [5]. However, here we are using a unique seed per sector, potentially leading to multiple seeds per LMS tree, but this shouldn't adversely affect the security of the scheme.

[3] Here we're referring to a hypertree-based hierarchical scheme made up of multiple levels of trees, it is trivial to map this onto the non-hierarchical case.

node values in our 2^S sector-based example. At this point the private key is complete and we can generate any HBS from within that sector[4]. If someone were to attempt to use a sector's private key to generate a valid signature for another sector then they would not have the information they need to compute the required path information above the sector root since that was never recorded in their private key. In addition, the cryptographically-isolated sector seed values ensure they can't feasibly guess the required seed value they'd need to be able to compute another sector's root value by recomputing its leave's OTS private key values. This is all summarized in the simple sectorization example shown in Figure *5*.



Off-path info we need to store in sector's private key ($SectorInfo_i$):

Sector 0: $\{T_5, T_3\}$ so $SectorInfo_0 = \{h, S, pubLms, I_0, seed_{0,0}, T_5, T_3\}$
Sector 1: $\{T_4, T_3\}$ so $SectorInfo_1 = \{h, S, pubLms, I_0, seed_{0,1}, T_4, T_3\}$
Sector 2: $\{T_7, T_2\}$ so $SectorInfo_2 = \{h, S, pubLms, I_0, seed_{0,2}, T_7, T_2\}$
Sector 3: $\{T_6, T_2\}$ so $SectorInfo_3 = \{h, S, pubLms, I_0, seed_{0,3}, T_6, T_2\}$

Pseudo-random seed/I generation:

$seed_{0,0}$ = H(u128(0) || u32(0) || u16(D_TOPSEED) || u8(1) || masterSeed)
$seed_{0,1}$ = H(u128(0) || u32(1) || u16(D_TOPSEED) || u8(1) || masterSeed)
$seed_{0,2}$ = H(u128(0) || u32(2) || u16(D_TOPSEED) || u8(1) || masterSeed)
$seed_{0,3}$ = H(u128(0) || u32(3) || u16(D_TOPSEED) || u8(1) || masterSeed)
I_0 = H(u128(0) || u32(0) || u16(D_TOPSEED) || u8(2) || masterSeed)

Figure 5. Simple Sectorization

What this allows us to do is confidently generate a new multi-sector HBS scheme in a suitable hardware device (e.g., HSM), and then export the resulting sector private keys using suitable levels of protection and oversight (e.g., AES key wrapping [7] combined with M-of-N secret

[4] Recall that the sector seed allows us to compute any of the sector's OTS private keys, which allows us to compute the corresponding OTS private keys and LMS leaf nodes within the sector. Given those values we can compute any of the node values within the sector, allowing us to ascend to the sector root node. Finally, we use the off-path information we saved during key generation to allow us to ascend to the root of the LMS tree, thereby completing the sectorized signature generation process. This is illustrated in Figure *5*.

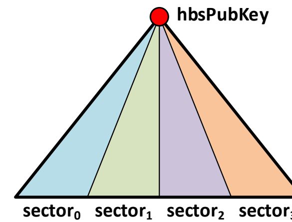
sharing to shard the wrapping key[5]), while destroying any materials within the device used to generate those sector private keys in order to ensure only a single copy exists of each. These sector keys can then be distributed as per the functional and procedural requirements of the application to ensure the appropriate levels of performance and survivability can be achieved (e.g., we may have multiple parallel HBS signing instances created to satisfy high performance requirements, or have geographically distinct instances to allow for improved survivability). Similarly, we may want to dole out the exported private keys to multiple devices over time to support very long-lived keys (longevity of the key) that would normally outlive the HSM devices into which we would be importing them.

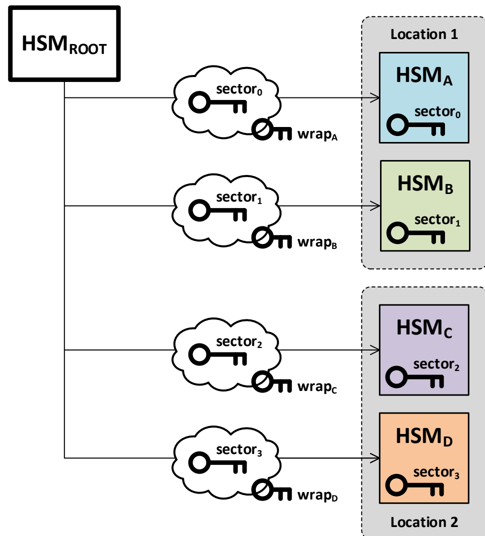
Care still needs to be taken to ensure the sector keys are only ever loaded into a single device (e.g., instituting a security policy that sees the physical media containing the exported sector key is destroyed after a single use/load), but that is far simpler than trying to ensure multiple copies of the same state are always coherent and up-to-date. Assuming there is no load replication, then devices with different sectors loaded into them are guaranteed to never re-use the same state, allowing for seamless operation during DR scenarios and recoveries. Furthermore, if a failed device is ultimately able to be brought back online then, since it was the only device loaded with its sector information, it can safely resume generating signatures without any worries about state re-use[6]. Figure 6 and Figure 7 depict simple 4-sector, and more generic N-sector, scenarios respectively to help illustrate the use of sectorization in HBS applications.

[5] Note that this is a standard private key export mechanism used extensively within today's PKI installations to provide DR, so we are simply leveraging today's best practices to bring a similar level of protection to HBS key management.

[6] Assuming of course it was well-behaved and managing its own internal state appropriately, which is an absolute necessity in all cases so we think this is a very reasonable assumption.

- Root HSM generates 4 sector HBS and wraps each sector generated with one of the wrapping keys for **HSM_A** – **HSM_D**, and exports them via approved means.
- Root HSM then destroys all materials related to sector generation (i.e., exporting of a sector is a one-time operation).





- **HSM_A** begins generating signatures from sector 0.
  - If performance demands, **HSM_B** can be brought online to double the signature generation capability by generating signatures from sector 1 in parallel with **HSM_A**.
- If at some point **HSM_A** and/or **HSM_B** go down then backup devices **HSM_C** and/or **HSM_D** can be brought online to generate signatures from sectors 2 and 3 respectively in order to shoulder the load of the missing HSM(s).
- If **HSM_A** and/or **HSM_B** recover then **HSM_C** and/or **HSM_D** could be taken offline again
  - Or **HSM_A** and **HSM_B** could remain offline and become the backup devices to **HSM_C** and **HSM_D**.
- All HSMs are generating signatures that verify back to hbsPubKey so they are all part of the same signing authority.

- Sector keys loaded up into destination HSMs and readied to generate signatures.
- Any external media used to convey the sector data (e.g., CD-ROM, USB key, etc.) is destroyed to eliminate potential re-loading and/or duplication (i.e., importation of a given sector is a one-time operation).

Figure 6. 4-sector example HBS scenario

- Root HSM generates N sector HBS, wraps each generated sector **sector**$_i$ with wrapping key **wrap**$_i$, and exports them via approved means for storage in an approved container (e.g., a safe in this example).
- Root HSM then destroys all materials related to sector generation (i.e., exporting of a sector is a one-time operation).

- Sector keys imported into destination HSMs via reconstitution of corresponding wrapping key on the destination HSM, and then readied to generate signatures.
- Any external media used to convey the sector data (e.g., CD-ROM, USB key, etc.) is destroyed to eliminate potential re-loading and/or duplication (i.e., importation of a given sector is a one-time operation).
- **HSM**$_i$ begins generating signatures from sector 0.
  - If performance demands, **HSM**$_j$ (and others) can be brought online to increase the signature generation capability by generating signatures from other sectors in parallel with **HSM**$_i$.
- If at some point **HSM**$_i$ and/or **HSM**$_j$ go down then backup device **HSM**$_m$ (and others) can be brought online to generate signatures from new sectors (e.g., sector n) in order to shoulder the load of the missing HSM(s).
- If **HSM**$_i$ and/or **HSM**$_j$ recover then **HSM**$_m$ could be taken offline again
  - Or **HSM**$_i$ and/or **HSM**$_j$ could remain offline and become the backup device(s) to **HSM**$_m$.
- Any of the HSMs can be loaded with additional sectors if they find themselves running out of signature capacity over time (or they may have been pre-loaded with these additional sectors during configuration).
- All HSMs are generating signatures that verify back to hbsPubKey so they are all part of the same signing authority.

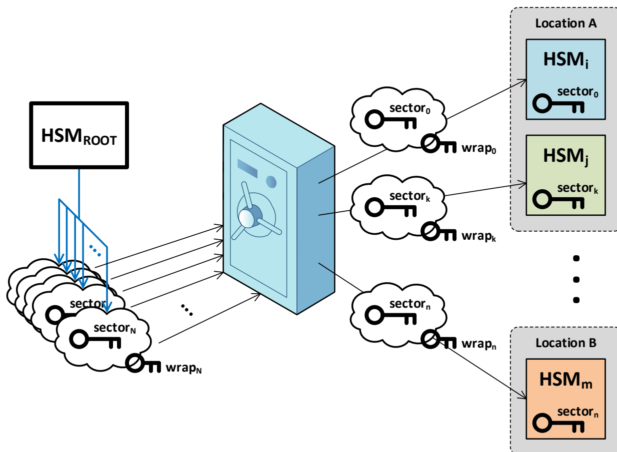


Figure 7. Generic N-sector example HBS scenario

Sectorization does introduce an additional level of complexity in the initial key generation process as we need to estimate the amount of longevity/survivability and redundancy we think is required. This will require the user to define their performance/resilience requirements, which should be well known and understood as part of their system planning and lifecycle management efforts. In addition, schemes such as LMS-HBS are able to tune each level of the HBS hypertree to deliver an optimal trade-off in terms of the number of sectors, key generation time, number of possible signatures, and signature size. So, we should be able to define an optimal configuration to meet our quantum safe signature generation needs.

It also merits stating that the sectorization concept is basically an instantiation of the reservation concept described in Section 5 of [8], where we use cryptographic isolation between the reservations such that each reservation's private key information cannot be derived from another reservation's private key information, thereby preventing it from generating a HBS from a different reservation.

Lastly, it should be noted that sectorization is essentially a vertical slicing of the tree made up by an HBS scheme, be it a single level LMS/XMSS or a multi-level HSS/XMSS^{MT} structure. In either case we are taking the bottom-most leaf nodes and grouping them together into contiguous regions (a.k.a., sectors), and distributing each group to an individual HSM such that this HSM can generate all HBS within a single sector. This is illustrated by the image of the tree shown in the

top-right of Figure 6 where each coloured portion of the tree is allocated to a different HSM. Note as well that the sectorization is only done at the top-most subtree in the case of $XMSS^{MT}$/HSS[7].

Speeding up key generation

Earlier on we had identified the key generation time as being a potential drawback of HBS. The introduction of hypertree-based variants such as HSS/$XMSS^{MT}$ help address this issue by only requiring us to compute the public-keys and node values for subtrees along the path from the current signature state (i.e., bottom-level leaf node) up to the hypertree's root. In a sectorized implementation we can simplify this further to have the initial sector generation only compute the top-level public key, sector seed, and additional off-path information (which is computed from the sector seed information). This information together constitutes all of the information required for a device to compute the rest of the HBS information, which can be done as part of a readying operation when the sector generation information is imported into its destination device. Hence, the device generating all of the initial sector information (i.e., $SectorInfo_i$ in Figure *5*) should be able to do so quite quickly assuming reasonable parameter choices. As an added bonus, the amount of exported information is minimized by this approach, making it easier to store and transport as well.

References

1. **Merkle, Ralph.** *Secrecy, Authentication, and Public Key Systems.* Stanford : Stanford University, 1979.

2. *SPHINCS: practical stateless hash-based signatures.* **Bernstein, Daniel, et al.** 2015.

3. *The SPHINCS+ Signature Framework.* **Berstein, Daniel, et al.** 2019.

4. **Langley, Adam.** Hash based signatures. *ImperialViolet.* [Online] July 18, 2013. https://www.imperialviolet.org/2013/07/18/hashsig.html.

5. *Leighton-Micali Hash-Based Signatures.* **McGrew, David, Curcio, Michael and Fluhrer, Scott.** s.l. : IETF, 2019, RFC8554.

6. *XMSS: eXtended Merkle Signature Scheme.* **Hulsing, Andreas, et al.** s.l. : IETF, 2018, RFC8391.

7. *Recommendation for Block Cipher Modes of Operation: Methods of Key Wrapping.* **Dworkin, Morris.** s.l. : NIST, 2012. SP800-38F.

8. *State Management for Hash-Based Signatures.* **McGrew, David, et al.** s.l. : Springer, 2016, Lecture Notes in Computer Science (Security Standardisation Research), Vol. 10074, pp. 224-260.

[7] Nothing precludes you from sectorizing additional levels of the tree, but we think the vast majority of interesting use cases can be satisfied by a simple single-layer sectorization of the top-most subtree. Hence, we chose to exercise the time-honoured KISS principle for the purposes of this introductory whitepaper.

CRYPTO4A