Hash Based Signatures and Ascon-Sign

Presentation at Second Oxford Post-Quantum Cryptography Summit

Vikas Srivastava[1] (2020rsma011@nitjsr.ac.in) Anubhab Baksi[2] (anubhab.baksi@ntu.edu.sg)

September 7, 2023

[1]National Institute of Technology Jamshedpur, India

[2]Nanyang Technological University, Singapore

Contents

Introduction

Digital Signatures [I]

- *Digital signatures* are among the most important cryptographic tools
- Applications for digital signatures include digital certificates for e-commerce, legal signing of contracts etc.

Digital Signatures [II]

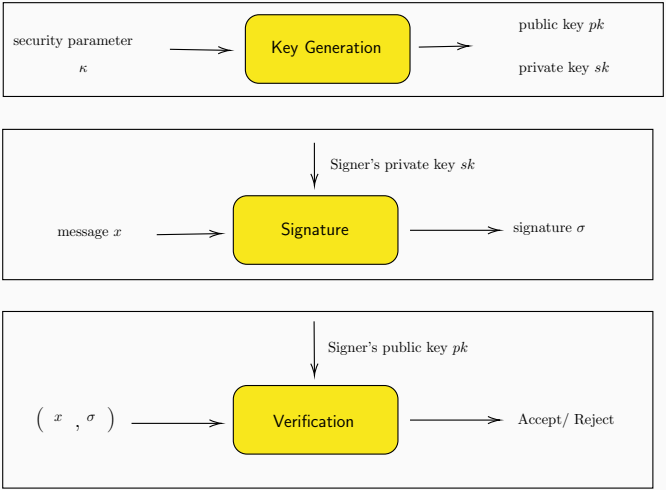

Figure 1: Overview of digital signature schemes

Principle of Digital Signatures

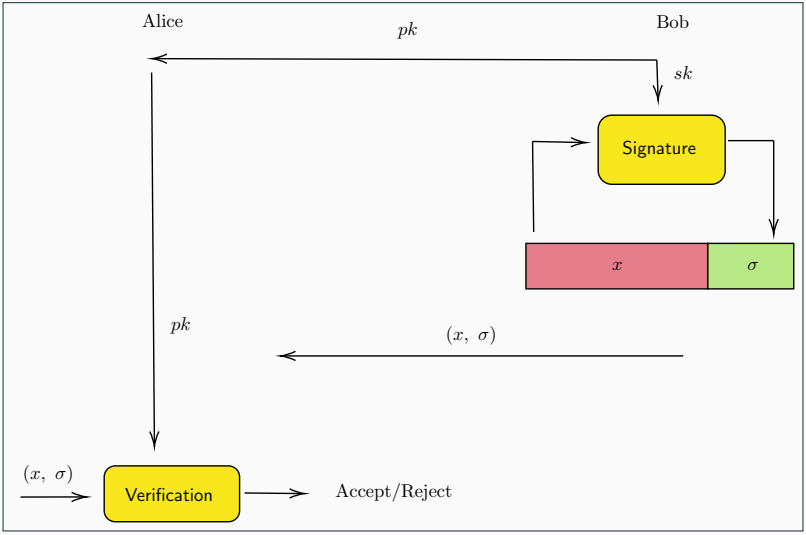

Figure 2: Usage of digital signatures

Post-Quantum Digital Signatures [I]

- Quantum computers will impact security of most (if not all) of the signature schemes used today (see Shor's algorithm[1])
- The proposed solution is the so-called *Post-Quantum Signatures*

[1] Peter W Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.

Post-Quantum Digital Signatures [II]

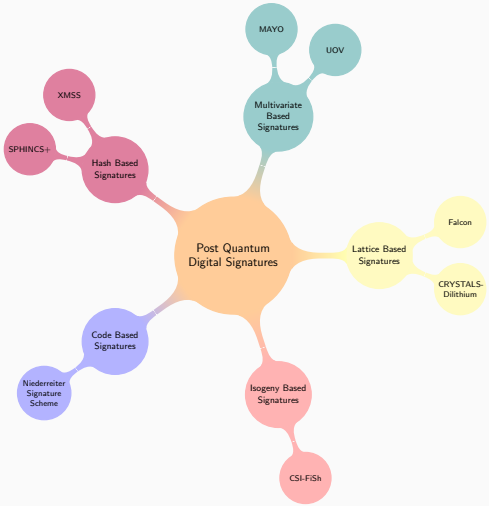

Figure 3: Classification of post-quantum digital signatures

Hash Based Signatures

- A *hash function* maps an arbitrary length message to a fixed length message:

$$H : \{0,1\}^* \rightarrow \{0,1\}^n$$

- It is easy to compute but hard to revert back (one-way property)
- The desirable properties of a hash functions are:
  - Collision resistance (hard to find $x' \neq x$ with $H(x') = H(x)$ given x)
  - Pre-image resistance (hard to find x given $H(x)$)
  - Second pre-image resistance (given x, hard to find $x' \neq x$ with $H(x') = H(x)$)
- These are considered quite fast and simple to implement/analyze
- Hash functions belong to the symmetric key cryptography and are generally well-understood

Lamport OTS (LOTS)

Lamport OTS [I]

Overview

- The first hash-based signature scheme[2] by Lamport back in 1979
- It is based on the observation that, given only a (secure) hash function (collision resistance is not needed) one can build a (secure) signature

[2] Leslie Lamport. "Constructing digital signatures from a one way function". In: *SRI International (CSL-98)* (1979).

Key Generation

- Hash $H : \{0,1\}^{256} \to \{0,1\}^{256}$
- Message $m \in \{0,1\}^{256}$
- Private key (sk_0, sk_1):

$$sk_0 = sk_1^0, sk_2^0, \ldots, sk_{256}^0$$

$$sk_1 = sk_1^1, sk_2^1, \ldots, sk_{256}^1$$

- Public key (pk_0, pk_1):

$$pk_0 = H(sk_1^0), H(sk_2^0), \ldots, H(sk_{256}^0)$$

$$pk_1 = H(sk_1^1), H(sk_2^1), \ldots, H(sk_{256}^1)$$

Signature

- Represent m as a sequence of 256 individual bits:

$$m = m_1, \ldots, m_{256}, \quad m_i \in \{0, 1\}$$

- For $i = 1$ to 256;
  if the i^{th} message bit $M_i = 0$, take the i^{th} private string (sk_i^0) from the sk_0;
  output that string as part of our signature
- If the message bit $M_i = 1$, we take the appropriate string (sk_i^1) from the sk_1 list
- Concatenate all the strings together to output the signature (σ)

Verification

Given a message signature pair (m, σ) and the public key $pk = (pk_0, pk_1)$, a verifier proceeds in the following way:

- Let σ_i denotes the i^{th} component of σ
- For each $i \in \{1, 256\}$, the verifier considers the message-bit m_i, and calculate $H(\sigma_i)$
- If $M_i = 0$, the $H(\sigma_i)$ should be equal to the corresponding element from pk_0
- If $M_i = 1$, $H(\sigma_i)$ should be equal to the corresponding element in pk_1
- Signature is declared valid if every component of the signature when hashed, matches the correct portion of the pk

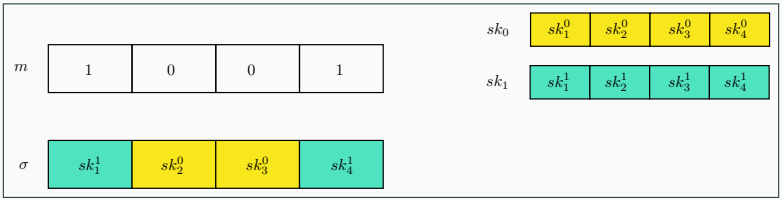

Figure 4: Signature generation of LOTS (example with message $m = 1001$ with secret key $sk = (sk_0, sk_1)$)

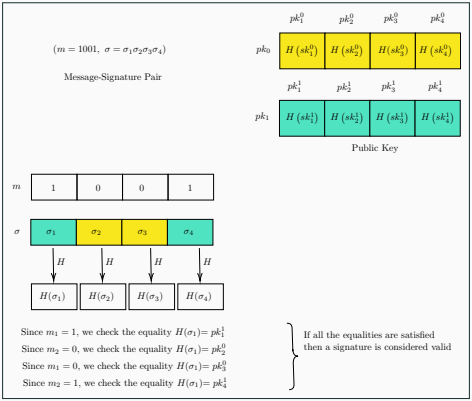

Figure 5: Signature generation of LOTS (example with message $m = 1001$ with secret key $sk = (sk_0, sk_1)$)

Winternitz One Time Signature (WOTS+)

Overview

- The WOTS+[3] approach aims to reduce the size of signatures and key pairs, albeit at the cost of additional hash evaluations
- WOTS+ first converts the message m into a new form using a base w representation, and then breaks it down into blocks of length $\log w$
- For each block, it applies a function up to a maximum of $w - 1$ times, and the output of the function becomes the signature for that block
- The resulting signatures for each block are concatenated in sequence to form the entire signature for m

[3] Andreas Hülsing. "W-OTS+–shorter signatures for hash-based signature schemes". In: *International Conference on Cryptology in Africa*. Springer. 2013, pp. 173–188.

WOTS+ [II]

Parameters and Functions

- Cryptographic hash, $F : \{0,1\}^n \rightarrow \{0,1\}^n$
- Pseudo-random generators $G_\lambda : \{0,1\}^n \rightarrow \{0,1\}^{\lambda n}$ for different values of λ
- Base w
- Message length n
- $l_1 = \lceil \frac{n}{log(w)} \rceil$
- $l_2 = \lfloor \frac{\log(l_1(w-1))}{\log w} \rfloor + 1$
- $l = l_1 + l_2$

Chain Function

- Given a input value $x \in \{0,1\}^n$, a iterative counter $i \in \mathbb{N}$, and bitmask $r = (r_1, \ldots, r_j) \in \{0,1\}^{n \times j}$ ($j \geq i$), the chain function works as follows:
  1. If $i = 0$, $c^0(x, r) = x$
  2. If $i \geq 0$, $c^i(x, r) = F(c^{i-1}(x, r) \oplus r_i)$

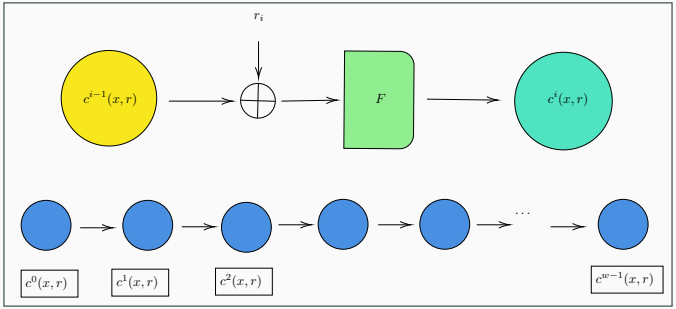

Figure 6: Chain function used in WOTS+

Key Generation

- Inputs:
  - Seed $S \in \{0,1\}^n$
  - Bitmasks $r = (r_1, \ldots, r_{w-1}) \in \{0,1\}^{n \times (w-1)}$
- Secret key:

$$sk = (sk_1, \ldots, sk_l) \leftarrow G_l(S)$$

- Public key:

$$pk = (pk_1, \ldots, pk_l) = (c^{w-1}(sk_1, r), \ldots, c^{w-1}(sk_l, r))$$

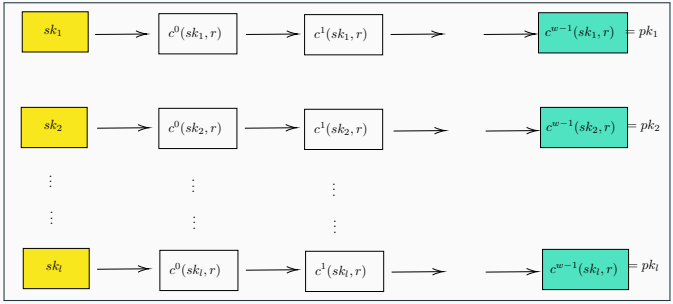

Figure 7: Key generation in WOTS+

Signature

- Inputs:
  - Message m (n-bit long)
  - Seed S
  - Bitmasks r
- Compute a base-w representation of m; $m = (m_1, \ldots, m_{l_1})$, $m_i \in \{0, \ldots, w-1\}$
- $C = \sum_{i=1}^{l-1}(w - 1 - M_i)$ is represented in base w representation, $C = (C_1, \ldots, C_{l_2})$
- Append m and C to get $b = m||C$, i.e., $b = (b_1, \ldots, b_l)$
- Signature:

$$(\sigma_1, \ldots, \sigma_l) = (c^{b_1}(sk_1, r), \ldots, c^{b_l}(sk_l, r))$$

Verification

- Construct $(b_1, \ldots, b_l)$
- Check

$$(pk'_1, \ldots, pk'_l) \stackrel{?}{=} (c^{w-1-b_1}(\sigma_1, r_{b_1+1,w-1}), \ldots, c^{w-1-b_l}(\sigma_1, r_{b_l+1,w-1}))$$

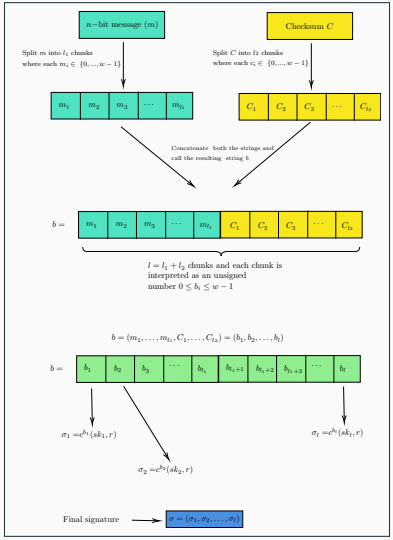

Figure 8: Signature generation (schematic) in WOTS+

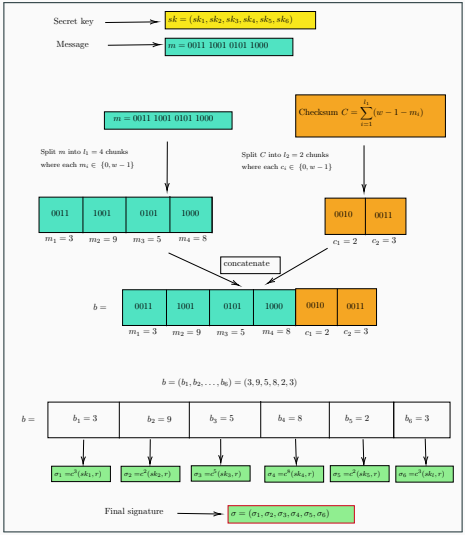

Figure 9: Toy example of WOTS+ signature generation ($n = 16, w = 16$)

Merkle Signature Scheme (MSS)

Overview

- The Merkle Tree Signature Scheme[4] to manage OTS keys
- The idea here is to use Merkle tree leaves to store OTS keys
- Merkle trees are binary trees (a Merkle tree of height h has 2^h leaves)
- The signature consists of the index of the leaf, the OTS public key, the digest of the OTS public key (the leaf), and the authentication path of that leaf

[4] Ralph C Merkle. "A certified digital signature". In: *Advances in cryptology—CRYPTO'89 proceedings*. Springer. 2001, pp. 218–238.

Key Generation

- Generate $N = 2^n$ public key-private key pairs

$$(OTS_{PK_0}, OTS_{SK_0}), \ldots (OTS_{PK_{N-1}}, OTS_{SK_{N-1}})$$

  of some OTS scheme
- For each $i \in \{0, 2^n - 1\}$, compute $h_i = H(OTS_{PK_i})$
- The hash values h_i are placed as leaves and hashed recursively to form a binary tree
- The private key of the Merkle signature scheme is the entire set of (OTS_{PK_i}, OTS_{SK_i}) pairs
- The public key pub is the root of the tree $a_{n,0}$

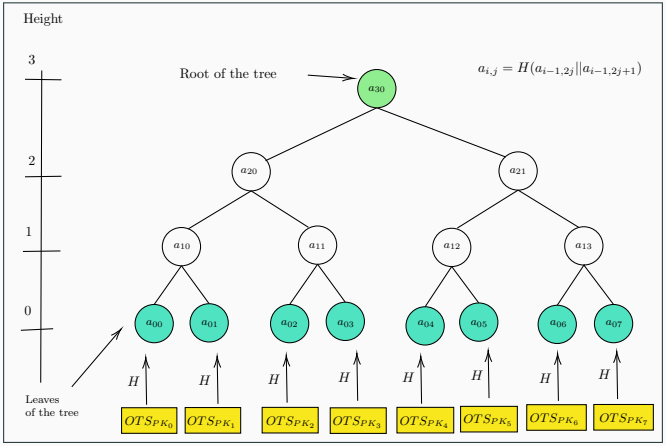

Figure 10: Key generation of Merkle signature (with Merkle tree height $h = 3$ and $2^h = 8$ leaves)

Signature

- Select the i^{th} public key OTS_{PK_i} from the tree
- Sign the message using the corresponding OTS secret key OTS_{SK_i} resulting in a signature σ_{OTS_i}
- Signature $\sigma = (i, OTS_{PK_i}, \sigma_{OTS_i}, Auth_i)$

Verification

- Verify σ_{OTS_i} using the OTS public key OTS_{PK_i}
- Computes $a_{0,i} = H(OTS_{PK_i})$
- Using $Auth_i$ to compute the root of the Merkle tree pub'
- If pub'=pub, then the verifier declares the signature as valid otherwise rejects

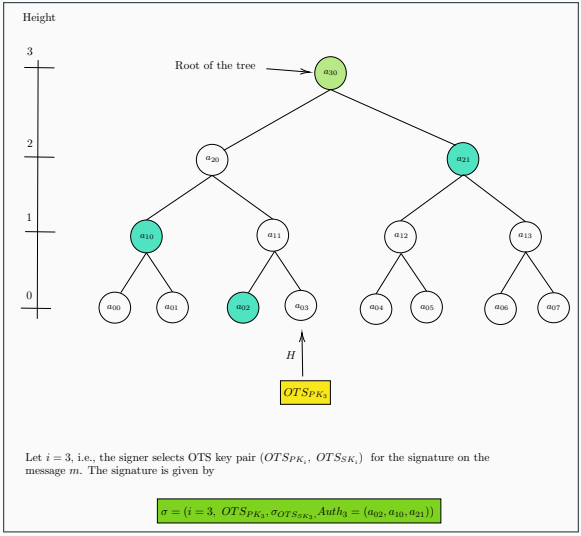

Figure 11: Signature generation of MSS (toy example)

Few Time Signature (FTS)

HORS [I]

Overview

- *Hash to Obtain Random Subset* (HORS)[5] is an FTS algorithm
- Unlike OTS, an FTS algorithm can be used to sign messages for a few times
- Each time it is used, some information is exposed, thereby reducing the key's security

[5] Leonid Reyzin and Natan Reyzin. "Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying". In: *Information Security and Privacy*. Ed. by Lynn Batten and Jennifer Seberry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 144–153. ISBN: 978-3-540-45450-2.

Key Generation

- Generates t random n-bit strings to produce the secret key: $SK = (s_1, \ldots, s_t)$
- Public key is computed as $PK = (k, v_1, \ldots, v_t)$ where $v_i = F(sk_i)$

Signature

- Compute $a = Hash(m)$
- Split a into k substrings $a_1, \ldots a_k$, of length $\log_2 t$ bits each
- Interpret each a_j as an integer i_j for $1 \leq j \leq k$
- Output signature $\sigma = (sk_{i_1}, \ldots, sk_{i_k})$

Verification

- Compute $a = Hash(m)$
- Split a into k substrings $a_1, \ldots a_k$, of length $\log_2 t$ bits each
- Interpret each a_j as an integer i_j for $1 \leq j \leq k$
- If for each j (where $1 \leq j \leq k$), $v_{i_j} = F(sk_j)$; accept the signature; otherwise reject

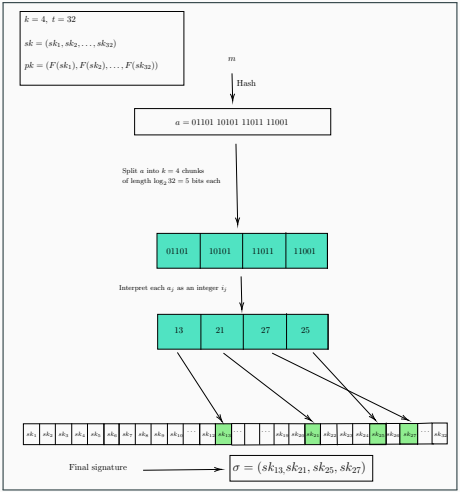

Figure 12: HORS signature (toy example)

Hash to Obtain Random Subsets with Trees (HORST)

HORST [I]

Overview

- HORST[6] is proposed by Bernstein et al. as an improvement over HORS
- Uses a binary hash-tree structure to reduce the size of both the public key and signature
- HORST replaces the t-value public key with a single value that represents the root of the Merkle tree, the leaves of this tree are the pk_i's
- A HORST signature includes k sk_i's and their respective authentication paths

[6] Daniel J Bernstein et al. "SPHINCS: practical stateless hash-based signatures". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2015, pp. 368–397.

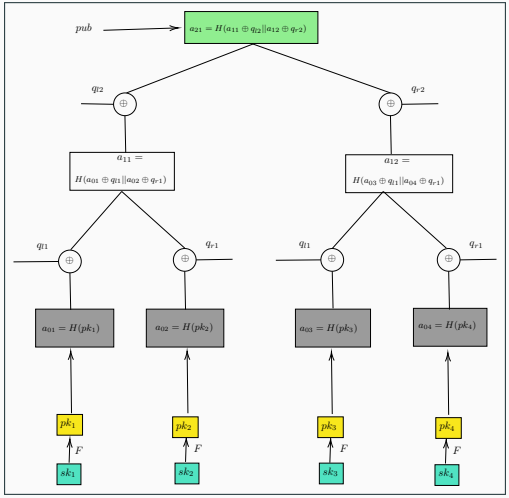

Figure 13: HORST key generation

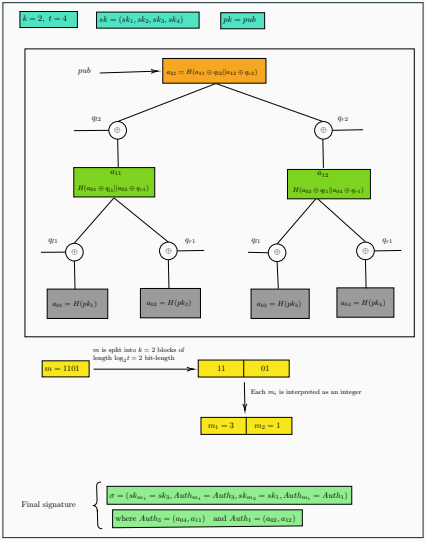

Figure 14: HORST signature generation (toy example)

Extended Merkle Signature Scheme (XMSS)

Overview

- The XMSS-XOR[7] tree is an improved variant of the Merkle tree
- Level j, $0 < j \leq h$, is constructed using a bit-mask $(q_{l,j}||q_{r,j}) \in_R \{0,1\}^{2n}$
- The nodes are computed as

$$a_{i,j} = Hash((a_{2i,j-1} \oplus q_{l,j})||(a_{2i+1,j-1} \oplus q_{r,j}))$$

[7] Bernstein et al., "SPHINCS: practical stateless hash-based signatures"; Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS-a practical forward secure signature scheme based on minimal security assumptions". In: *International Workshop on Post-Quantum Cryptography*. Springer. 2011, pp. 117–129.

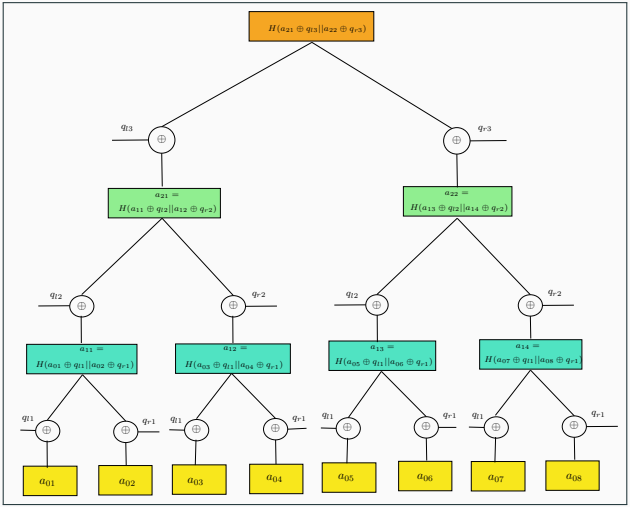

Figure 15: Tree structure of XMSS (height = 3)

L-Tree in XMSS

- Used to compress the public keys of each WOTS+
- The first l leaves of an L-tree are the l bit strings $(pk_1, \ldots, pk_l)$ from the corresponding public key of WOTS+
- If l is not a power of 2, a node with no right sibling is pushed to a higher level of the L-tree until it becomes the right sibling of another node

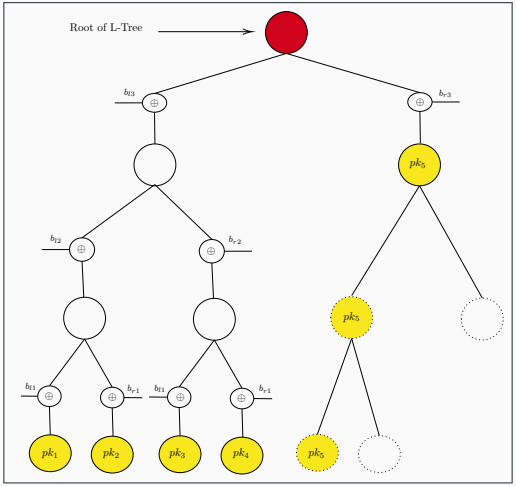

Figure 16: Toy example illustrating the L-tree construction of a WOTS+ public key $pk = (pk_1, \ldots, pk_5)$.

Signature

- To sign the i^{th} message, the i^{th} W-OTS key pair is used
- The signature $SIG = (i, \sigma, Auth)$

Verification

- To verify $SIG = (i, \sigma, Auth)$, the string $(b_1, \ldots, b_l)$ is computed
- The i^{th} verification key $(pk_1, \ldots, pk_l)$ is computed similar to verification algorithm of WOTS+
- The corresponding leaf of the XMSS tree is constructed using the L-tree.
- This leaf and the authentication path are used to compute the root

XMSS [VII]

MSS vs XMSS

- Leaves of XMSS-tree is not simply a hash of OTS public key
- Root of another tree (also known as L-tree) is used as the leaves of the XMSS tree

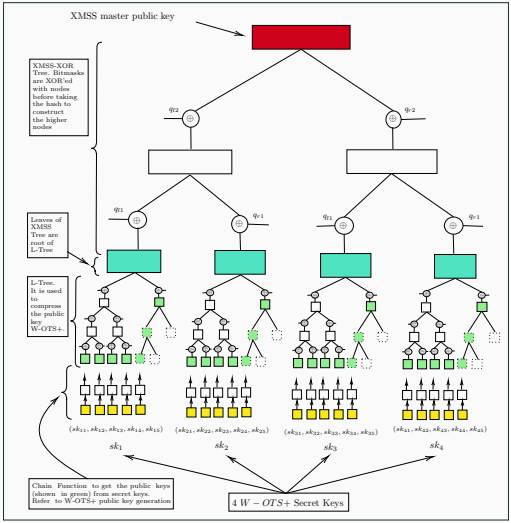

Figure 17: Representation of XMSS with L-Tree

XMSS Multitree (XMSS^{MT})

XMSS^{MT} [I]

Overview

- Hypertree variant of XMSS[8] which enables an unlimited number of messages to be signed cryptographically
- Uses XMSS to build the interior authentication path in a subtree
- Utilizes WOTS+ to sign the root of the subtree by the signature key corresponding to the leaf node on the one layer higher

[8] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. "Optimal parameters for XMSS MT". In: *Security Engineering and Intelligence Informatics: CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Regensburg, Germany, September 2-6, 2013. Proceedings 8*. Springer. 2013, pp. 194–208.

Overview

- Trees at the lowest level are utilized for message signing
- Trees at higher levels are used for signing the roots of the trees located on the layer below
- To create a signature, all these WOTS+ signatures along the way to the highest tree are combined
- Signature $\sigma = (i, \sigma_0, Auth_0, \sigma_1, Auth_1, \ldots, \sigma_d, Auth_d)$

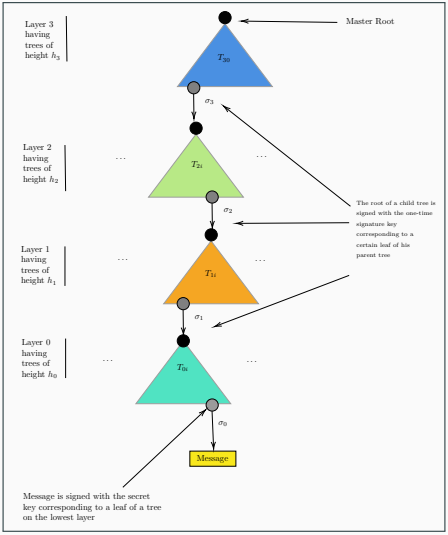

Figure 18: Pictorial description of XMSS^{MT} with 4 layers.

SPHINCS

SPHINCS [I]

Stateless vs. Stateful

- Stateful schemes have a Merkle tree (or tree of trees) with a number of one-time signatures at the bottom
- Each one-time signature can be used once; hence the signer needs to keep track of which ones have been used
- When signer uses a one-time signature to sign a message, the state needs to be updated

SPHINCS [II]

Stateless vs. Stateful

- Stateless schemes (such as SPHINCS[9]) has a large tree-of-trees; but at the bottom, they have a number of few time signatures (SPHINCS uses HORST)
- Each such few time signature can sign several messages
- The signer picks a random few-time-signature, uses that to sign the message, and then authenticates that through the Merkle trees up to the root (which is the public key)
- Since we are using a few-time signature, we do not mind if we pick the same few-time signature multiple times on occasion
- The few time signature scheme can handle it
- Since we do not need to update any state when generating a signature (the so-called *stateless* property)

[9] Bernstein et al., "SPHINCS: practical stateless hash-based signatures".

SPHINCS [III]

Overview

- Composed of WOTS+, XMSS, and HORST as the building blocks
- Stateless
- SPHINCS can manage a much larger quantity of keys without the need to pre-compute all the leaves by utilizing two methods:
  - Hyper-tree
  - Random key path addressing scheme

Overview

- The hyper-tree structure in SPHINCS is a tree of trees, where the height of the hyper-tree is denoted by h. This hyper-tree is composed of trees with a height of h/d
- At the bottom level of the SPHINCS hyper-tree, there is a level of HORS trees that contain private keys used for signing messages
- When a message needs to be signed, SPHINCS selects a HORS tree to sign the message and generates a signature σ_H
- Above the HORS level, which is level 0, there are L-trees consisting of WOTS+ key pairs. Each leaf of these trees contains the public key strings of WOTS+, and their corresponding private keys are used for signing the root of the trees on the level below

Overview

- There is only one tree on level $d-1$ which is the top tree
- There are $2^{(d-i-1)*(h/d)}$ trees on level $i, i \in [0, d-2]$, and the root of the tree in level i will be signed by the WOTS+ private key of the tree on level $i+1$
- SPHINCS only identifies specific paths in the hyper-tree when signing a message (by employing an addressing scheme to locate the WOTS+ public keys in the hyper-tree)
- The addressing scheme consists of the level of the hyper-tree, the tree on that level, and the leaf within that tree
- We can uniquely identify the location of each WOTS+ public key at every level of the hyper-tree

Key Generation

- An n-bit key SK_1 generated using a PRG. It is used to generate random seeds for HORST and WOTS+ private key generation
- An n-bit key SK_2 is also generated using a PRG. This key is used for generating an unpredictable index and message hash
- Bitmasks $Q = (Q_0, Q_1, \ldots, Q_{p-1})$: Bitmasks are used in HORST, WOTS+, L-tree, and hyper-tree. WOTS+ needs $w - 1$ bitmasks, HORST needs $2\log(t)$ bitmasks, and L-tree needs $2\lceil\log(l)\rceil$ bitmasks. In total, the complete SPHINCS structure needs p bitmasks where $p = \max(w - 1, 2(h + \lceil\log(l)\rceil), 2\log(t))$

Key Generation

- The address of the leaves of the trees at the highest layer, i.e., layer $d-1$ is given by

$$A = (d-1||0||i)(i \in [2^{\frac{h}{d}} - 1)]$$

- Generate the seed $S_A \leftarrow F(A, SK_1)$ using the n-bit secret key SK_1
- Use S_A as the seed for the generation of the private keys of WOTS+
- Compute the root of this top level tree (let us denote the root by PK_{root})
- Final private key and public key of SPHINCS is given by

$$SK = (SK_1, SK_2, Q)$$
$$PK = (PK_{\text{root}}, Q)$$

Signature Generation

- Generate two random n-bit numbers R_1 and R_2 by $F(M, SK_2)$
- Compute the message digest $D \leftarrow H(R_1, M)$
- Compute HORST address $i \leftarrow Chop(R_2, h)$ and $\text{Address}_H = (d||i(0,(d-1)h/d)||i((d-1)h/d, h/d))$
- Generate HORST key pair and HOTST signature
  - Generate HORST key pair seed by $\text{Seed}_H \leftarrow F(\text{Address}_H, SK_1)$
  - Generate HORST signature and public key by (σ_H, pk_H) by executing the signature generation algorithm of HORST with (D, Seed_H, Q_H) as inputs
- Generate all WOTS+ signatures along the SPHINCS path
  - Compute all addresses of WOTS+ in the path $\text{Address}_{w,j} = (j||i(0,(d-1-j)h/d||i((d-1-j)h/d), h/d))$ where j is the level and $j \in [0, d-1]$.
  - Compute all the seeds $\text{Seed}_{w,j} = F(\text{Address}_{w,j}, SK_1)$
  - Generate WOTS+ signature $\sigma_{w,j}$ by running the signature generation of WOTS+ with $(pk_{w,j-1}, \text{Seed}_{w,j}, Q_{WOTS+})$ as inputs (here, $pk_{w,j-1}$ is the root of the tree of $j-1$ level)
  - We need the authentication path auth_{A_j} of corresponding WOTS+ public key
- $\sigma_{\text{SPHINCS}} = (i, R_1, \sigma_H, \sigma_{w,0}, \text{auth}_{A_0}, \sigma_{w,1}, \text{auth}_{A_1}, \ldots, \sigma_{w,d-1}, \text{auth}_{A_{d-1}})$

Verification

- The first step involves checking the HORST signature. The verification algorithm computes the digest D by computing $H(R_1, M)$.
- The verifier runs verification of HORST with (D, Q_{HORST}, σ_H) as inputs to check the validity of the HORST signature σ_H
- The second step involves checking all WOTS+ signatures. The verifier first verifies $\sigma_{w,0}$ by executing verification algorithm of WOTS+ with $(pk_H, \sigma_{w,0}, Q_{HORST})$ as inputs.
- In the following, the verifier verifies $\sigma_{w,i}$ by running the verification algorithm of WOTS+ with $(pk_{w,i}, \sigma_{w,i}, Q_{HORST+})$ as inputs. Here $i \in [1, d-1]$
- Reject if any one of the WOTS+ signatures cannot be validated
- On hyper-tree level $d-1$, the verifier gets the root of the hyper-tree. If the $root == PK_{\text{root}}$, the $\sigma_{SPHINCS}$ is validated, otherwise reject

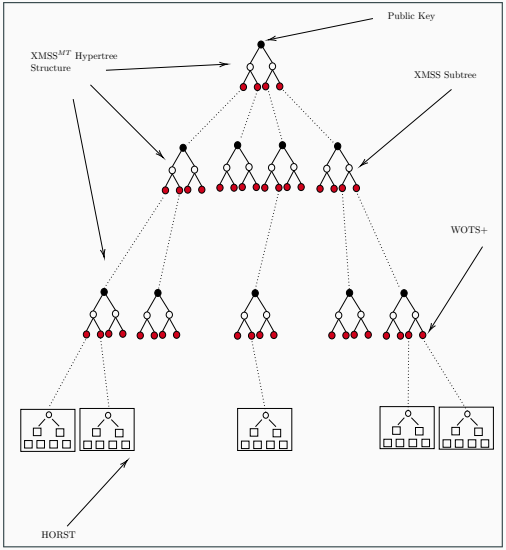

Figure 19: Construction of SPHINCS (schematic)

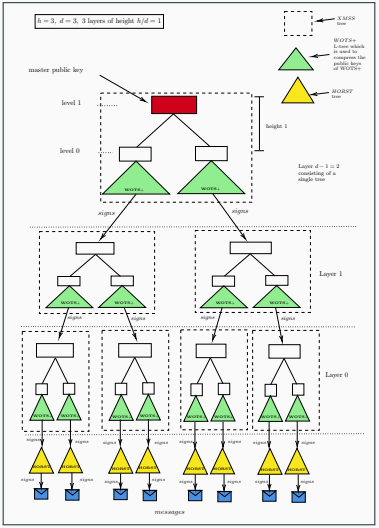

Figure 20: Tree structure of SPHINCS (schematic)

SPHINCS+

SPHINCS+: An Improvement over SPHINCS

- SPHINCS+[10] is proposed by Bernstein et al.
- Multi-target attack protection
- Tree-less WOTS+ public key compression
- FORS
- Verifiable index selection

[10] Daniel J Bernstein et al. "The SPHINCS+ signature framework". In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 2129–2146.

SPHINCS+ [II]

Variants

- Korean-SPHINCS[11]
- SPHINCS-Simpira[12]
- SPHINCS-Streebog[13]
- SPHINCS-α[14]
- SPHINCS-Gravity[15]

[11] Minjoo Sim et al. "K-XMSS and K-SPHINCS+ : Hash based Signatures with Korean Cryptography Algorithms". In: *Cryptology ePrint Archive* (2022).

[12] Shay Gueron and Nicky Mouha. "Simpira v2: A family of efficient permutations using the AES round function". In: *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*. Springer. 2016, pp. 95–125; Shay Gueron and Nicky Mouha. "Sphincs-simpira: Fast stateless hash-based signatures with post-quantum security". In: *Cryptology ePrint Archive* (2017).

[13] EO Kiktenko et al. "SPHINCS+ post-quantum digital signature scheme with Streebog hash function". In: *AIP Conference Proceedings*. Vol. 2241. 1. AIP Publishing LLC. 2020, p. 020014.

[14] Kaiyi Zhang, Hongrui Cui, and Yu Yu. "SPHINCS-α: A Compact Stateless Hash-Based Signature Scheme". In: *Cryptology ePrint Archive* (2022).

[15] Jean-Philippe Aumasson and Guillaume Endignoux. "Improving stateless hash-based signatures". In: *Topics in Cryptology–CT-RSA 2018: The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*. Springer. 2018, pp. 219–242.

SPHINCS-α

- The design rationale behind SPHINCS-α follows the original SPHINCS+ construction and apply the optimized CS-WOTS+ one time signature scheme
- CS-WOTS+ allows a larger message space/signature size ratio as compared to the original WOTS+ scheme (allowing a smaller one-time signature size)
- For instance, in both running time (in terms of the expected number of hash function calls) and size:
  - The SPHINCS+-256s parameter set suggests $w = 16$ and $l = 67$
  - For SPHINCS-α, we require $l = 66$ for $w = 16$ (which reduces both running time and size by 1.5%)

Ascon-Sign

Ascon-Sign [I]

Overview

- We propose *Ascon-Sign*[16], which is a variant of the SPHINCS signature scheme with Ascon-Hash and Ascon-XOF[17] as building blocks
- The ASCON cipher suite offers both authenticated encryption with associated data (AEAD) and hashing capabilities
- The primary goal of Ascon-Sign is to offer efficient and secure cryptographic operations for immediate use in a resource-constrained environment

[16] Vikas Srivastava et al. *Ascon-Sign*. NIST PQC Additional Round 1 Candidates. https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/Ascon-sign-spec-web.pdf. 2023.
[17] Christoph Dobraunig et al. "Ascon v1. 2: Lightweight authenticated encryption and hashing". In: *Journal of Cryptology* 34 (2021), pp. 1–42.

Hash Function Usage

Table 1: Hash function calls in Ascon-Sign

Task	Input	Notation
Generation of pseudo-random string from the message	Secret seed SK.prf; optional random value OptRand; message M	$\mathbf{PRF}_{msg}$(SK.prf, OptRand, M)
Computation of message digest	Pseusorandom bytestring R; public seed PK.seed; public XMSS-MT root PK.root; message M	$\mathbf{H}_{msg}$(R, PK.seed, PK.root, M)
Generation of FTS secret key elements	Secret seed SK.seed; element address ADRS	**PRF**(SK.seed, ADRS)
Hash-tree construction of FTS	Public seed PK.seed; address of node to compute ADRS; hash strings of two children nodes M_1, M_2	**H**(PK.seed, ADRS, M_1, M_2)
FTS tree roots compression	Public seed PK.seed; address in XMSS^{MT} tree ADRS; k roots of FORS trees roots	$\mathbf{T}_{km}$(PK.seed, ADRS, *roots*[])
Generation of underlying OTS secret key	Secret seed SK.seed; WOTS+ key element address ADRS	**PRF**(SK.seed, ADRS)
Chain function iteration in WOTS+	Public seed PK.seed; chain address of node to compute ADRS; previous element in chain	**F**(T, PK.seed, ADRS)
Compression of public keys of underlying OTS	Public seed PK.seed; WOTS+ keypair address ADRS; WOTS+ public key elements *pub*	$\mathbf{T}_{km}$(PK.seed, ADRS, *pub*)
Computation of subtree tree on top of compressed OTS keys	Public seed PK.seed; address of node to compute ADRS; hash strings of two children nodes M_1, M_2	**H**(PK.seed, ADRS, M_1, M_2)

Ascon-Sign [III]

Variants

- Two variants of Ascon-Sign are proposed, namely the 'simple' version and the 'robust' version, similar to the approach used in SPHINCS+[18]
  - For the 'robust' instances, the process involves generating pseudo-random bitmasks, which are then XORed with the input message. These masked messages are represented as $M^{\oplus}$
  - The 'simple' instances do not include the generation of bitmasks. The 'simple' instances offer faster performance since they eliminate the need for additional calls to the PRF to generate bitmasks

[18] Bernstein et al., "The SPHINCS+ signature framework".

Hash Function Usage

- $\mathbf{H_{msg}}(\text{R}, \text{PK.seed}, \text{PK.root}, \text{M}) = \text{Ascon-XOF}(\text{R}||\text{PK.seed}||\text{PK.root}||\text{M}, 8m)$
- $\mathbf{PRF}(\text{SEED}, \text{ADRS}) = \text{Ascon-Hash}(\text{SEED}||\text{ADRS})$
- $\mathbf{PRF_{msg}}(\text{SK.prf}, \text{OptRand}, \text{M}) = \text{Ascon-Hash}(\text{SK.prf}||\text{OptRand}||\text{M})$

Hash Function Usage (Simple Variant)

$$\mathbf{F}(\text{PK.seed}, \text{ADRS}, \text{M}_1) = \text{Ascon-Hash}(\text{PK.seed}||\text{ADRS}||\text{M}_1),$$

$$\mathbf{H}(\text{PK.seed}, \text{ADRS}, \text{M}_1||\text{M}_2) = \text{Ascon-Hash}(\text{PK.seed}||\text{ADRS}||\text{M}_1||\text{M}_2)$$

$$\mathbf{T}_l(\text{PK.seed}, \text{ADRS}, \text{M}) = \text{Ascon-Hash}(\text{PK.seed}||\text{ADRS}||\text{M})$$

Hash Function Usage (Robust Variant)[19]

- $\mathbf{F}(\text{PK.seed}, \text{ADRS}, \text{M}_1) = \text{Ascon-Hash}(\text{PK.seed}||\text{ADRS}||\text{M}_1^{\oplus})$
- $\mathbf{H}(\text{PK.seed}, \text{ADRS}, \text{M}_1||\text{M}_2) = \text{Ascon-Hash}(\text{PK.seed}||\text{ADRS}||\text{M}_1^{\oplus}||\text{M}_2^{\oplus})$
- $\mathbf{T}_l(\text{PK.seed}, \text{ADRS}, \text{M}) = \text{Ascon-Hash}(\text{PK.seed}||\text{ADRS}||\text{M}^{\oplus})$

[19] For a message M with *len* bytes we compute $\text{M}^{\oplus} = \text{M} \oplus \text{Ascon-XOF}(\text{PK.seed}||\text{ADRS}, \textit{len})$.

Parameters

- n : the security parameter in bytes.
- w : the Winternitz parameter
- h : the height of the hypertree
- d : the number of layers in the hypertree
- k : the number of trees in FORS
- t : the number of leaves of a FORS tree
- m: the message digest length in bytes: $m = \lfloor (k \log t + 7)/8 \rfloor + \lfloor (h - h/d + 7)/8 \rfloor + \lfloor (h/d + 7)/8 \rfloor$
- len: the number of n-byte string elements in a WOTS + private key, public key, and signature. It is computed as $len = l_1 + l_2$, with $l_1 = \lceil 8n/\log w \rceil and l_2 = \lceil \log(len_1(w-1))/\log(w) \rceil$

Hash Calls

Table 2: Hash calls in Ascon-Sign

	F	**H**	**PRF**	$\mathbf{T}_{\text{len}}$
Key Generation	$2^{h/d} w$ len	$2^{h/d} - 1$	$2^{h/d}$len	$2^{h/d}$
Signing	$kt + d(2^{h/d})w$ len	$k(t-1) + d(2^{h/d} - 1)$	$kt + d(2^{h/d})$len	$d2^{h/d}$
Verification	$k + dw$ len	$k \log t + h$	–	d

Key and Signature Sizes

Table 3: Key and signature sizes for Ascon-Sign

	Secret key	Public key	Signature
Size	$4n$	$2n$	$(h + k(\log t + 1) + d \cdot len + 1)n$

Parameters for Ascon-Hash and Ascon-XOF

	Size in bits of			Rounds	
	Hash output	Rate	Capacity	p^a	p^b
Ascon-Hash	256	64	256	12	12
Ascon-XOF	arbitrary	64	256	12	12

Parameters Sets

Table 4: Parameter sets for Ascon-Sign

	n	h	d	$\log(t)$	k	w	Security level	Signature size (bytes)
Ascon-Sign-128s	16	63	7	12	14	16	1	7856
Ascon-Sign-128f	16	66	22	6	33	16	1	17088
Ascon-Sign-192s	24	63	7	14	17	16	3	16224
Ascon-Sign-192f	24	66	22	8	33	16	3	35664

Security

- Ascon-Sign is based on the SPHINCS+[20] signature framework with Ascon-Hash and Ascon-XOF[21] as the internal hash function
- Ascon-Sign is expected to have the same security strength as SPHINCS+

[20] Bernstein et al., "The SPHINCS+ signature framework".
[21] Dobraunig et al., "Ascon v1. 2: Lightweight authenticated encryption and hashing".

Software Benchmark [I]

- CPU: Intel Core i5 10210U
- Architecture: x64
- Number of cores: 4
- Base clock speed: 1.60 GHz
- Memory (RAM): 8 GiB
- Operating System: Linux Lite 5.2
- Linux kernel version: 5.4.0-113-generic
- Compiler: GCC 9.4.0
- Compiler optimization flag: `-Wall -Wextra -Wpedantic -O3 -std=c99`
- Official benchmark reported in the submission document

Software Benchmark [I]

Table 5: Runtime results for reference and optimized implementation of Ascon-Sign ('simple' variant)

		Key Generation	Signing	Verification
Reference	Ascon-Sign-128s	315840896	2413174678	2429047
	Ascon-Sign-128f	5939611	115382780	6972950
	Ascon-Sign-192s	599392072	5458909051	4696353
	Ascon-Sign-192f	10939221	243023163	13058030
Optimized	Ascon-Sign-128s	291925878	2224377542	2137821
	Ascon-Sign-128f	5506606	107020221	6535295
	Ascon-Sign-192s	557050751	5046224790	4357430
	Ascon-Sign-192f	10117696	226197880	12333664

Software Benchmark [I]

Table 6: Runtime results for reference and optimized implementation of Ascon-Sign ('robust' variant)

		Key Generation	Signing	Verification
Reference	Ascon-Sign-128s	554679600	4225825170	5516617
	Ascon-Sign-128f	10156899	198139090	12469524
	Ascon-Sign-192s	1046162651	9916984141	10281218
	Ascon-Sign-192f	18827117	419872255	23006148
Optimized	Ascon-Sign-128s	530089300	4038032800	4232362
	Ascon-Sign-128f	10678534	182601975	11279318
	Ascon-Sign-192s	970639431	8893090510	7664451
	Ascon-Sign-192f	17174517	381735599	21408883

Software Benchmark [II]

- CPU: Intel Xeon W-2133
- Architecture: x64
- Number of cores: 6
- Base clock speed: 3.60 GHz
- TurboBoost: Enabled
- Hyper-threading: Enabled
- Memory (RAM): 16 GiB
- Operating System: Ubuntu 22.04.2
- Linux kernel version: 5.15.90
- Compiler: GCC 11.3
- Compiler optimization flag: `-Wall -Wextra -Wpedantic -O3 -std=c99`
- May not be exactly accurate as the computer was getting heated up while running the code

Software Benchmark [II]

Table 7: Runtime results for reference and optimized implementation of Ascon-Sign ('simple' variant)

		Key Generation	Signing	Verification
Reference	Ascon-Sign-128s	666346681	5267597208	4241980
	Ascon-Sign-128f	9502308	284849767	14878615
	Ascon-Sign-192s	1265809671	11659028599	11543490
	Ascon-Sign-192f	16717264	478521179	23295295
Optimized	Ascon-Sign-128s	609495351	4742432290	3956422
	Ascon-Sign-128f	8573282	201057590	11571899
	Ascon-Sign-192s	1068742913	10477062639	7687683
	Ascon-Sign-192f	16337614	500222731	23693364

Software Benchmark [II]

Table 8: Runtime results for reference and optimized implementation of Ascon-Sign ('robust' variant)

		Key Generation	Signing	Verification
Reference	Ascon-Sign-128s	1165937867	8896055893	15669154
	Ascon-Sign-128f	16119623	377882287	22074915
	Ascon-Sign-192s	2340760358	19439420305	15164053
	Ascon-Sign-192f	29026503	965865414	41771553
Optimized	Ascon-Sign-128s	1204735982	8547151265	7169614
	Ascon-Sign-128f	14574125	378004800	20351596
	Ascon-Sign-192s	2134160121	18417354765	13758299
	Ascon-Sign-192f	26695005	884829864	39895975

Benchmark Comparison [I]

Data are taken from *Signatures Zoo*[22] (less in y-axis is better)

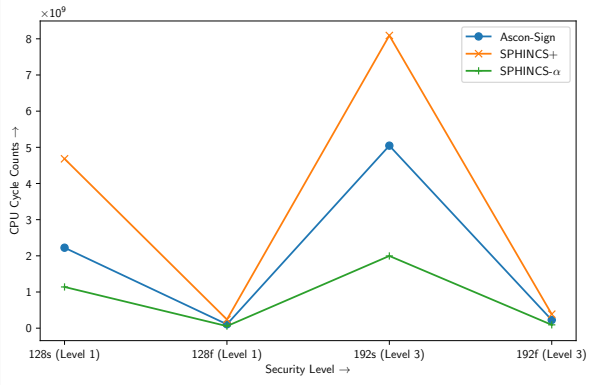

Figure 21: Signature generation time comparison

[22] PQShield. *Post-Quantum Signatures Zoo*. https://pqshield.github.io/nist-sigs-zoo/. (Visited on 09/07/2023).

Benchmark Comparison [II]

Data are taken from *Signatures Zoo*[23] (less in y-axis is better)

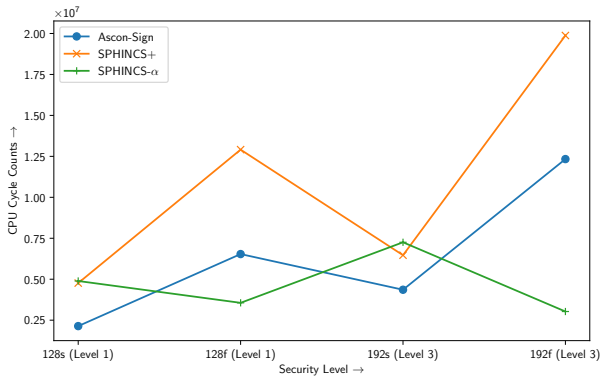

Figure 22: Verification time comparison

[23] PQShield, *Post-Quantum Signatures Zoo*.

Future Prospects

Future Prospects [I]

Ongoing Plans

- We hope to propose a quantum Level 5 secure variant
- Currently we are trying to implement/optimize in hardware (e.g., hardware implementation of Ascon-Hash[24]) and software
- We intend to eventually have our own hash function so that we can replace Ascon-Hash (which will hopefully be more lightweight)
- We are considering to adopt SPHINCS-α as the basis for signature (instead of SPHINCS+)
- Last, but not the least, we are trying to come up with our own signature (so far it seems to be more efficient than SPHINCS+/SPHINCS-α)

[24] Aneesh Kandi et al. "Hardware Implementation of ASCON". In: *Lightweight Cryptography Workshop* (2023). URL: https://csrc.nist.gov/csrc/media/Events/2023/lightweight-cryptography-workshop-2023/documents/accepted-papers/07-hardware-implementation-of-ascon.pdf.

Future Prospects [II]

Ongoing Plans

- We observe that the size of OTS/FTS scheme affects the overall performance significantly
- Therefore, we are working on reducing the size of OTS/FTS
- Our preliminary investigation suggests it may be possible to reduce the computation and verification cost of OTS/FTS.
- We hope to use different binary tree/hypertree structure used in SPHINCS+ signature framework
- Our initial estimates are as follows:
  - Constant time verification independent of parameter (as opposed to parameter dependent verification time in SPHINCS+)
  - Verification is probabilistic but system parameters can be tuned to make the failure probability as low as we want

Wrap-up

We always welcome any kind of suggestion, feedback, implementation...