Ascon-Sign

Submission to the NIST Post-quantum Project

Vikas Srivastava[1], Naina Gupta[2], Arpan Jati[2], Anubhab Baksi[2], Jakub Breier[3], Anupam Chattopadhyay[2], Sumit Kumar Debnath[1], and Xiaolu Hou[4]

[1]National Institute of Technology Jamshedpur, India
[2]Nanyang Technological University, Singapore
[3]Silicon Austria Labs, Graz, Austria
[4]Slovak University of Technology, Bratislava
[1]2020rsma011@nitjsr.ac.in, naina003@e.ntu.edu.sg, arpan.jati@ntu.edu.sg, anubhab.baksi@ntu.edu.sg, jbreier@jbreier.com, anupam@ntu.edu.sg, sdebnath.math@nitjsr.ac.in, xiaolu.hou@stuba.sk

June 1, 2023

Contents

Contents 2

1 Introduction 3

2 Brief Description of Ascon-Hash/Ascon-XOF 3

3 Hash Function Usage in Ascon-Sign 4

4 Proposed Signature Based on ASCON Hash Function Family 6
4.1 Primary Structure: Few Time Signature . . . 6
4.2 Secondary Structure: One Time Signature . . . 6
4.3 Tertiary Structure: Merkle Tree Based Signatures . . . 11
4.4 Quaternary Structure: Hypertree Based Signatures . . . 11
4.5 Ascon-Sign: Combining Everything Together . . . 11

5 Parameters, Size, and Security of Ascon-Sign 15

6 Performance Analysis 18

7 Advantages and Limitations 19

References 20

1 Introduction

As with the progress of quantum computing in recent times, we can see the need to develop a relatively new type of cryptographic primitives which can be considered secure. The underlying security assumptions of these primitives are such that the currently best known algorithms (classical and quantum alike) cannot break it. Since the digital signature schemes which are commonly used in today's electronic communication are not considered secure enough against the quantum computers, there is a push for the so-called post-quantum signatures. Based on the existing literature, we can see a variety of post-quantum signatures, like hash based [13, 11, 2, 16], lattice based [9, 8], code based [14], multivariate polynomial based [6, 15], isogeny based [5].

Each of the varieties has its own strengths and weaknesses, and the choice of which post-quantum signature scheme to use will depend on the specific requirements and constraints of the application. Among these, hash based signatures are considered promising. The concept of hash functions is quite well-known/well-studied in the symmetric key cryptography over the past couple of decades, this gives an edge for these signatures. Symmetric key ciphers typically are known to be quantum resistant (see, e.g., [12]), it makes intuitive sense to use those ciphers for post-quantum application scenario.

Our Contribution

We introduce Ascon-Sign, which is a variant of the SPHINCS+ signature scheme with ASCON [7] as a building block. SPHINCS+ was proposed in [3] as a hash-based signature scheme with post-quantum security. The ASCON cipher suite offers both authenticated encryption with associated data (AEAD) and hashing capabilities. Thus, the primary goal of Ascon-Sign is to offer efficient and secure cryptographic operations for immediate use in a resource-constrained environment.

2 Brief Description of Ascon-Hash/Ascon-XOF

We give a brief description of Ascon-XOF [7] and Ascon-Hash [7]. The 320-bit starting state of Ascon-XOF and Ascon-Hash is determined by a constant value called IV. This constant includes various parameters for the algorithm, such as k (set to 0), the rate r, the number of rounds a, and b (set to 0), all represented as 8-bit integers. Additionally, it includes the maximum output length h in bits, written as a 32-bit integer (where $h = l = 256$ for Ascon-Hash and $h = 0$ for unlimited output in Ascon-XOF). The final part of the constant is a 256-bit value consisting of all zeros.

The state S is initialized by applying the a-round permutation p^a. Blocks of r bits are used to process the message M. In order to make the length of the padded message a multiple of r bits, a single 1 and the fewest number of 0s are appended to M during the padding procedure. The resultant padded message is split into s blocks of r bits, $M_1 \parallel \ldots \parallel M_s$: $M_1, \ldots, M_s \leftarrow r$-bit blocks of $M \parallel 1 \parallel 0^{r-1-(|M| \mod r)}$.

The message block M_i with $i = 1, \ldots, s$ is Xored to the first r bits S_r of the initialized state S. In the following, We apply the b-round permutation p^b to S if $i < s$. In the next step, the state S is transformed by the a-round permutation p^a: $S \leftarrow p^a(S)$. We now extract the hash output from the state in r-bit blocks H_i until the requested output length $o \leq h$ is completed after $t = \lceil d/r \rceil$ blocks. After each extraction, the internal state, denoted as S, undergoes a transformation using a permutation function called p^b:

$$H_i \leftarrow S_r$$

$$S \leftarrow p^b(S), 1 \leq i \leq t = \lceil l/r \rceil$$

The final output block, referred to as H_t, is shortened to a length of $l \mod r$ bits. The resulting truncated block, along with the previous output blocks H_1 through $\tilde{H}_t$, are concatenated together to form the overall output H:

$$\tilde{H}_t \leftarrow \lfloor H_t \rfloor_{l \mod r}$$

The mode of operation for hashing is based on sponges [4]. The hashing algorithm is specified in Algorithm 1.

Algorithm 1 ASCON Hashing

Input: message $M \in \{0,1\}^*$, output bit size $o \leq h$ or o arbitrary if $h = 0$
#Output: hash $H \in \{0,1\}$
function INITIALIZATION
    $S \leftarrow p^a(IV_{h,r,a} \parallel 0^c)$
end function
function ABSORBING
    $M_1 \dots M_s \leftarrow M \parallel 1 \parallel 0*$
    for $i = 1, \dots, s$ **do**
        $S \leftarrow p^a((S_r \oplus M_i) \parallel S_c)$
    end for
end function
function SQUEEZING
    for $i = 1, \dots, t = \lceil o/r \rceil$ **do**
        $H_i \leftarrow S_r$
        $S \leftarrow p^a(S)$
    end for
    return $\lfloor H_1 \parallel \dots \parallel H_t \rfloor_o$
end function

ASCON Security Claim

Both Ascon-Hash and Ascon-XOF provide 128-bit security against collision attacks and (second) pre-image attacks, as stated in Table 1. Note that the security of Ascon-XOF is reduced if the output size is less than 256 bits. Like other sponge based hash functions, both Ascon-Hash and Ascon-XOF also resist other attacks, including length extension attacks and second pre-image attacks for long messages.

Table 1: Security claims for recommended parameter configurations of Ascon-Hash and Ascon-XOF

Requirement	Security in bits	
	Ascon-Hash	Ascon-XOF
Collision resistance	128	$\min(128, o/2)$
(Second) Pre-image resistance	128	$\min(128, o)$

Ascon-Hash gives 256-bit output
Ascon-XOF gives o-bit output

As the designers of Ascon-Hash claimed, ideal properties for the permutations are not necessary regarding security features [7]. For more details about the security claims and the state-of-the-art analysis of Ascon-Hash and Ascon-XOF, we refer to [7, Section 6.4].

3 Hash Function Usage in Ascon-Sign

The hash function usage in Ascon-Sign is summarized in Table 2.

Table 2: Hash function calls in Ascon-Sign

Task	Input	Notation
Generation of pseudorandom string from the message	Secret seed SK.prf, optional random value OptRand, message M	$\mathbf{PRF_{msg}}(\mathsf{SK.prf}, \mathsf{OptRand}, \mathsf{M})$
Computation of message digest	R, public seed PK.seed, public XMSS-MT root PK.root, message M	$\mathbf{H_{msg}}(R, \mathsf{PK.seed}, \mathsf{PK.root}, \mathsf{M})$
Generation of FTS secret key elements	Secret seed SK.seed, element address ADRS	$\mathbf{PRF}(\mathsf{SK.seed}, \mathsf{ADRS})$
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	$\mathbf{H}(\mathsf{PK.seed}, \mathsf{ADRS}, M_1, M_2)$
FTS tree roots compression	Public seed PK.seed, address in XMSS^{MT} tree ADRS, k roots of FORS trees roots[]	$\mathbf{T_{len}}(\mathsf{PK.seed}, \mathsf{ADRS}, roots[])$
Generation of underlying OTS secret key	Secret seed SK.seed, WOTS+ key element address ADRS	$\mathbf{PRF}(\mathsf{SK.seed}, \mathsf{ADRS})$
Chain function iteration in WOTS+	Public seed PK.seed, chain address of node to compute ADRS, previous element in chain	$\mathbf{F}(T, \mathsf{PK.seed}, \mathsf{ADRS})$
Compression of public keys of the underlying OTS	Public seed PK.seed, WOTS+ keypair address ADRS, WOTS+ public key elements $pub[]$	$\mathbf{T_{len}}(\mathsf{PK.seed}, \mathsf{ADRS}, pub[])$
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	$\mathbf{H}(\mathsf{PK.seed}, \mathsf{ADRS}, M_1, M_2)$

We define the functions for Ascon-Sign as

$$\mathbf{H_{msg}}(\mathsf{R}, \mathsf{PK.seed}, \mathsf{PK.root}, \mathsf{M}) = \mathsf{Ascon\text{-}XOF}(\mathsf{R}||\mathsf{PK.seed}||\mathsf{PK.root}||\mathsf{M}, 8m),$$

$$\mathbf{PRF}(\mathsf{SEED}, \mathsf{ADRS}) = \mathsf{Ascon\text{-}Hash}(\mathsf{SEED}||\mathsf{ADRS}),$$

$$\mathbf{PRF_{msg}}(\mathsf{SK.prf}, \mathsf{OptRand}, \mathsf{M}) = \mathsf{Ascon\text{-}Hash}(\mathsf{SK.prf}||\mathsf{OptRand}||\mathsf{M}).$$

For the robust variant, we further define the tweakable hash functions as

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

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

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

For the simple variant, we instead define the tweakable hash functions as

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

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

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

Generating the Masks. Ascon-Hash can be used to construct Ascon-XOF. For a message M with l bytes we compute

$$\mathsf{M}^{\oplus} = \mathsf{M} \oplus \mathsf{Ascon\text{-}XOF}(\mathsf{PK.seed}||\mathsf{ADRS}, l).$$

Variants of Ascon-Sign: Simple and Robust

In the case of Ascon-Sign, two variants are proposed, namely the 'simple' version and the 'robust' version, similar to the approach used in SPHINCS+[3]. For the 'robust' instances, the process involves generating pseudorandom bitmasks, which are then XORed with the input message. These masked messages are represented as $\mathsf{M}^{\oplus}$. On the other hand, the 'simple' instances do not include the generation of bitmasks. The 'simple' instantiations offer faster performance since they eliminate the need for additional calls to the PRF to generate bitmasks. The advantage of the 'simple' instantiations lies in their improved speed, but the security argument for these instances relies entirely on the assumption of the random oracle model. In contrast, the 'robust' instantiations provide a more conservative security argument but are slower in terms of performance.

4 Proposed Signature Based on ASCON Hash Function Family

In this section, we describe the design of Ascon-Sign. Ascon-Sign comprises four level of structure: primary, secondary, tertiary and quaternary. The idea behind Ascon-Sign sign is that we replace the internal hash function in SPHINCS+ by Ascon-Hash and Ascon-XOF.

4.1 Primary Structure: Few Time Signature

We first discuss the primary structure of Ascon-Sign. At the bottom level of Ascon-Sign hyper tree, we have a level of a few time signature (FORS, see [3]). It contains the private keys used for signing messages. When a message needs to be signed, Ascon-Sign selects a FORS tree to sign the message and generates the signature SIG_{FORS}. Algorithm 2 describes the computation of trees. Algorithm 3 and Algorithm 6 describes respectively the public key and private key generation of FORS. Algorithm 4 presents the signature generation algorithm for FORS, while Algorithm 5 describes the computation of public key from the signature.

4.2 Secondary Structure: One Time Signature

As discussed before, Ascon-Sign like SPHINCS+ uses a hypertree structure. These subtrees are generated using the one time signature, namley WOTS+ [10, 2]. We use the compressed public keys as the leaves of the subtree. The private keys of WOTS+ are used to sign the roots of the subtrees at the lowest level. The fundamental building block used in WOTS+ is the chaining function. We describe the computation of chaining function in Algorithm 7. The key generation and signature generation of one time signature employed in the secondary structure of Ascon-Sign is presented in Algorithm 9, 8, and 10 respectively. In the end, we give the process of computing the public from the WOTS+ signature in Algorithm 11. Algorithm 11 will be used as sub process during Ascon-Sign verification.

Algorithm 2 FORS tree hash

```
# Input: SK.seed, s, z, PK.seed, ADRS
# Output: n-byte root node-top node on Stack
function FORS-TREE-HASH(SK.seed, s, z, PK.seed, ADRS)
    if s%(1 ≪ z)! = 0 then
        return -1;
    end if
    for i = 0; i < 2^z; i = i + 1 do
        ADRS.setTreeHeight(0);
        ADRS.setTreeIndex(s + i);
        sk = PRF(SK.seed, ADRS);
        node = F(PK.seed, ADRS, sk);
        ADRS.setTreeHeight(1);
        ADRS.setTreeIndex(s + i);
        while Top node on Stack has same height as node do
            ADRS.setTreeIndex((ADRS.getTreeIndex( ) −1) / 2)
            node = H(PK.seed, ADRS, (Stack.pop() ||node));
            ADRS.setTreeHeight(ADRS.getTreeHeight() +1);
        end while
        return Stack.push(node)
    end for
    return Stack.pop()
end function
```

Algorithm 3 FORS public key

```
# Input: SK.seed, PK.seed, ADRS
# Output: PK_FORS
function FORS-PK-GEN(SK.seed, PK.seed, ADRS)
    forspkADRS = ADRS
    for i = 0; i < k; i = i + 1 do
        root[i] = FORS-TREEHASH(SK.seed, i × t, a, PK.seed, ADRS)
    end for
    forspkADRS.setType(FORSROOTS)
    forspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
    PK_FORS = T_k(PK.seed, forspkADRS, root)
    return PK_FORS
end function
```

Algorithm 4 FORS signature

```
# Input: Bit string M, SK.seed, ADRS, PK.seed
# Output: FORS signature SIG_FORS
function FORS-SIGNATURE(M, SK.seed, PK.seed, ADRS)
    for i = 0; i < k; i++ do
        unsigned int idx = bits i × log(t) to (i + 1) × log(t) − 1 of M
        ADRS.setTreeHeight(0)
        ADRS.setTreeIndex(i × t + idx);
        SIG_FORS = SIG_FORS || PRF(SK.seed, ADRS)
        for j = 0; j < a; j = j + 1 do
            s = floor(idx/(2^j)) ⊕ 1;
            AUTH[j] = FORS-TREEHASH(SK.seed, i × t + s × 2^j, j, PK.seed, ADRS);
        end for
        SIG_FORS = SIG_FORS || AUTH
    end for
    return SIG_FORS
end function
```

Algorithm 5 FORS public key from signature

```
# Input: SIG_FORS, k log(t)-bit string M, PK.seed, ADRS
# Output: FORS public key PK_FORS
function FORS-PK-FROM-SIGN(SIG_FORS,M, PK.seed, ADRS)
    for i = 0; i < k; i = i + 1 do
        unsigned int idx = bits i × log(t) to (i + 1) × log(t) − 1 of M;
        sk = SIG_FORS.getSK(i)
        ADRS.setTreeHeight(0)
        ADRS.setTreeIndex(i × t + idx)
        node[0] = F(PK.seed, ADRS, sk)
        auth = SIG_FORS.getAUTH(i);
        ADRS.setTreeIndex(i × t + idx);
        for j = 0; j < a; j = j + 1 do
            ADRS.setTreeHeight(j + 1);
            if (floor(idx/(2^j))%2) == 0 then
                ADRS.setTreeIndex(ADRS.getTreeIndex() / 2)
                node[1] = H(PK.seed, ADRS, (node[0] || auth[j]))
            else
                ADRS.setTreeIndex((ADRS.getTreeIndex() −1)/2)
                node[1] = H(PK.seed, ADRS, (auth[j] || node[0]))
            end if
            node[0] = node[1];
        end for
        root[i] = node[0];
    end for
    forspkADRS = ADRS
    forspkADRS.setType(FORSROOTS);
    forspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress());
    PK_FORS = T_k(PK.seed, forspkADRS, root)
    return PK_FORS
end function
```

Algorithm 6 FORS private key

\# Input: SK.seed, ADRS, idx
\# Output: FORS private key sk_{FORS}
function FORS-SK-GEN(SK.seed, ADRS, idx)
    ADRS.setTreeHeight(0);
    ADRS.setTreeIndex(idx);
    sk_{FORS}= **PRF**(SK.seed, ADRS);
    return sk_{FORS}
end function

Algorithm 7 WOTS+ chaining function

\# Input: Input string X, start index i, number of steps s, public seed PK.seed, ADRS
\# Output: Computation of **F** iterated s times on the input string X
function WOTS-CHAIN(X, i, s, PK.seed, ADRS)
    if $s = 0$ **then**
        return X
    end if
    if $i + 1 > w - 1$ **then**
        return NULL
    end if
    byte$[n]$ tmp = WOTS-Chain($X, i, s - 1$, PK.seed, ADRS)
    ADRS.setHashAddress($i + s - 1$)
    tmp = **F**(PK.seed, ADRS, tmp);
    return tmp
end function

Algorithm 8 Private key generation WOTS+

\# Input: SK.seed, ADRS
\# Output: sk_{WOTS+}
function SKGEN-WOTS+(SK.seed, ADRS)
    for $i = 0; i < len; i = i + 1$ **do**
        ADRS.setChainAddress(i)
        ADRS.setHashAddress(0)
        $\mathsf{sk}_{WOTS+}[i]$ = **PRF**(SK.seed, ADRS)
    end for
    return sk_{WOTS+}
end function

Algorithm 9 Public key generation WOTS+

```
# Input: SK.seed, ADRS, PK.seed
# Output: pk_{WOTS+}
wotspkADRS = ADRS
function PKGen-WOTS+(SK.seed,PK.seed, ADRS)
    for i = 0; i < len, i = i + 1 do
        ADRS.setChainAddress(i)
        ADRS.setHashAddress(0)
        sk_{WOTS+}[i] = PRF(SK.seed, ADRS)
        tmp[i] = WOTS-Chain( pk_{WOTS+}[i], 0, w − 1, PK.seed, ADRS)
    end for
    wotspkADRS.setType(WOTS-PK)
    wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
    pk_{WOTS+}= T_{len}(PK.seed, wotspkADRS, tmp)
    return pk_{WOTS+}
end function
```

Algorithm 10 Signature generation WOTS+

```
# Input: M, SK.seed, PK.seed, ADRS
# Output: SIG_{WOTS+}
function WOTS-Sign(M, SK.seed, PK.seed, ADRS)
    csum=0
    M'=base_ w(M, w, l_1)
    for i = 0, i < l_1, i = i + 1 do
        csum = csum + w − 1 − M′[i]
    end for
    if log(w)%8! = 0 then
        csum = csum ≪ (8 − ((l_2 × log(w))%8))
    end if
    l_2_bytes = ceil(( l_2 × log(w))/8)
    M′ = M′ || base_w(toByte(csum, l_2_bytes), w, l_2)
    for i = 0, i < len, i = i + 1 do
        ADRS.setChainAddress(i)
        ADRS.setHashAddress(0)
        sk_{WOTS+}[i] = PRF(SK.seed, ADRS)
        SIG_{WOTS+}[i] = WOTS-Chain( sk_{WOTS+}[i], 0, M'[i], PK.seed, ADRS)
    end for
    return SIG_{WOTS+}
end function
```

Algorithm 11 WOTS public key from signature

```
# Input: M, SIG_{WOTS+}, PK.seed, ADRS
# Output: pk_SIG_{WOTS+}
function WOTS-PK-FROM-SIGN(M, SIG_{WOTS+}, PK.seed, ADRS)
    csum = 0
    wotspkADRS = ADRS
    M' = base-w(M, w, l_1)
    for i = 0, i < l_1, i = i + 1 do
        csum = csum + w − 1 − M'[i]
    end for
    csum = csum ≪ (8 − ((l_2 × log(w))%8))
    l_2_bytes = ceil((l_2 × log(w))/8)
    M' = M' ||base_w(toByte(csum, l_2_bytes), w, l_2)
    for i = 0, i < len, i = i + 1 do
        ADRS.setChainAddress(i)
        ADRS.setHashAddress(0)
        sk_{WOTS+}[i] = PRF(SK.seed, ADRS)
        tmp[i] = WOTS-Chain( SIG_{WOTS+}[i], 0, M[i], PK.seed, ADRS)
    end for
    wotspkADRS.setType(WOTS-PK)
    wotspkADRS.setKeyPairAddress(ADRS.getKeyPairAddress())
    pk_SIG_{WOTS+} = T_{len}(PK.seed, wotspkADRS, tmp)
    return pk_SIG_{WOTS+}
end function
```

4.3 Tertiary Structure: Merkle Tree Based Signatures

In tertiary structure of Ascon-Sign, we use a Merkle tree based signature. Ascon-Sign combines WOTS+ with binary hash tree to construct subtrees inside the hypertree. The leaves of these trees are public keys of WOTS+. To compute the internal nodes of binary hash trees, we use the Algorithm 12. Algorithm 13 and Algorithm 14 describes the process of key generation and signature generation corresponding to the subtree. Additionally, the Algorithm 15 is used as a subroutine in the verification process of Ascon-Sign. It provies a method of computing public keys from the signature.

4.4 Quaternary Structure: Hypertree Based Signatures

At the quaternary level, we have hypertree which consists of several layers of XMSS trees described in the tertiary structure (Section 4.3). The Key generation, signature generation, and verification algorithm of hypertree is described respectively in Algorithm 16, Algorithm 17 and Algorithm 18.

4.5 Ascon-Sign: Combining Everything Together

In the end, primary, secondary, tertiary, and quaternary structures combines together to give the design of Ascon-Sign. The key generation algorithm of Ascon-Sign is presented in Algorithm 19. Algorithm 20 and Algorithm 21 contains the description of signature generation and verification algorithm of Ascon-Sign.

Algorithm 12 Tree hash

\# Input: SK.seed, s, z, $PK.seed$, ADRS
\# Output: n-byte root node-top node on Stack
function TREEHASH(SK.seed, s, z, PK.seed, ADRS)
  if $s\%(1 \ll z)! = 0$ **then**
    return -1;
  end if
  for $i = 0; i < 2^z; i = i + 1$ **do**
    ADRS.setType(WOTS_ HASH)
    ADRS.setKeyPairAddress($s + i$);
    node = PKGEN-WOTS+(SK.seed, PK.seed, ADRS)
    ADRS.setType(TREE)
    ADRS.setTreeHeight(1)
    ADRS.setTreeIndex($s + i$)
    while Top node on Stack has same height as node **do**
      ADRS.setTreeIndex((ADRS.getTreeIndex() -1) / 2)
      node = **H**(PK.seed, ADRS, (Stack.pop() || node))
      ADRS.setTreeHeight(ADRS.getTreeHeight() $+1$)
    end while
    return Stack.push(node)
  end for
  return Stack.pop()
end function

Algorithm 13 XMSS key generation

\# Input: SK.seed, PK.seed, ADRS
\# Output: XMSS public key pk
function XMSS-PK-GEN(SK.seed, PK.seed, ADRS)
  pk = TREEHASH(SK.seed,$0, h'$, PK.seed, ADRS)
  return pk
end function

Algorithm 14 XMSS signature generation

\# Input: M, SK.seed idx, PK.seed, ADRS
\# Output: XMSS signature SIG$_{XMSS} = (SIG||AUTH)$
function XMSS-SIGN(M, SK.seed idx, PK.seed, ADRS)
  for $j = 0; j < h'; j = j + 1$ **do**
    k = floor$(idx/(2^j)) \bigoplus 1$;
    $AUTH[j]$ = TREEHASH(SK.seed, $k \times 2^j$, j, PK.seed, ADRS)
  end for
  ADRS.setType(WOTS_ HASH)
  ADRS.setKeyPairAddress(idx)
  SIG = WOTS-SIGN(M, SK.seed, PK.seed, ADRS)
  SIG$_{XMSS} = SIG||AUTH$
  return SIG$_{XMSS}$
end function

Algorithm 15 Public key from signature

```
# Input: idx, SIG_XMSS, M, PK.seed, ADRS
# Output: n-byte root value node[0]
function XMSS-PK-FROM-SIG(idx, SIG_XMSS, M, PK.seed, ADRS)
    ADRS.setType(WOTSHASH)
    ADRS.setKeyPairAddress(idx)
    SIG = SIG_XMSS.getWOTSSig()
    AUTH = SIG_XMSS.getXMSSAUTH();
    node[0] = WOTS-PK-FROM-SIGN(SIG, M, PK.seed, ADRS);
    ADRS.setType(TREE);
    ADRS.setTreeIndex(idx);
    for k = 0; k < h'; k++ do
        ADRS.setTreeHeight(k + 1)
        if (floor(idx/(2^k))%2) == 0 then
            ADRS.setTreeIndex(ADRS.getTreeIndex() /2);
            node[1] = H(PK.seed, ADRS, (node[0] || AUTH[k]));
        else
            ADRS.setTreeIndex((ADRS.getTreeIndex() −1)/2);
            node[1] = H(PK.seed, ADRS, (AUTH[k] || node[0]));
        end if
        node[0] = node[1];
    end for
    return node[0]
end function
```

Algorithm 16 Hypertree key generation

```
# Input: SK.seed, PK.seed
# Output: PK_HT
function HT-PK-GEN(SK.seed, PK.seed)
    ADRS = toByte(0, 32);
    ADRS.setLayerAddress(d − 1);
    ADRS.setTreeAddress(0);
    root = XMSS-PK-GEN(SK.seed, PK.seed, ADRS);
    return root;
end function
```

Algorithm 17 Hypertree signature

Input: Message M, SK.seed, PK.seed, tree index idxtree, leaf index idxleaf
Output: SIG_{HT}

function HT-SIGN(M, SK.seed, PK.seed, idxtree, idxleaf)
    ADRS = toByte(0, 32);
    ADRS.setLayerAddress(0);
    ADRS.setTreeAddress(idxtree);
    SIGtmp = XMSS-SIGN(M, SK.seed, idxleaf, PK.seed, ADRS);
    SIG_{HT} = SIG_{HT} || SIGtmp
    root = XMSS-PK-FROM-SIGN(idxleaf, SIGtmp, M, PK.seed, ADRS);
    for $j = 1; j < d; j = j + 1$ **do**
        idxleaf = (h/d) least significant bits of idxtree;
        idxtree = $(h - (j + 1) \times (h/d))$ most significant bits of idxtree;
        ADRS.setLayerAddress(j);
        ADRS.setTreeAddress(idxtree);
        SIGtmp = XMSS-SIGN(root, SK.seed, idxleaf, PK.seed, ADRS);
        SIG_{HT} = SIG_{HT} || SIGtmp
        if $j < d - 1$ **then**
            root = XMSS-PK-FROM-SIGN(idxleaf, SIGtmp, M, PK.seed, ADRS);
        end if
    end forreturn SIG_{HT}
end function

Algorithm 18 Hypertree verification

Input: Message M, signature SIG_{HT}, public seed PK.seed, tree index idxtree, leaf index idxleaf, PK_{HT}.
Output: Boolean

function HT-VERIFY(M, SIG_{HT}, PK.seed, idxtree, idxleaf, PK_{HT})
    ADRS = toByte(0, 32);
    SIGtmp = SIG_{HT}.getXMSSSignature(0);
    ADRS.setLayerAddress(0);
    ADRS.setTreeAddress(idxtree)
    node = XMSS-PK-FROM-SIGN(idxleaf, SIGtmp, M, PK.seed, ADRS);
    for $j = 1; j < d; j = j + 1$ **do**
        idxleaf = (h/d) least significant bits of idxtree
        idxtree = $(h - (j + 1) \times h/d)$ most significant bits of idxtree
        SIGtmp = SIG_{HT}.getXMSSSignature(j)
        ADRS.setLayerAddress(j)
        ADRS.setTreeAddress($idxtree$)
        node = XMSS-PK-FROM-SIGN(idxleaf, SIGtmp, node, PK.seed, ADRS)
    end for
    if node= PK_{HT} **then**
        return True;
    else
        return False;
    end if
end function

Algorithm 19 Ascon-Sign key generation

```
  # Output: Ascon-Sign key pair (SK,PK)
function Ascon-Sign-KG
    SK.seed= sec_rand(n)
    SK.PRF= sec_rand(n)
    PK.seed= sec_rand(n)
    PK.seed= HT-PK-GEN(SK.seed, PK.seed)
    return ((SK.seed, SK.prf, PK.seed, PK.root), (PK.seed, PK.root))
end function
```

5 Parameters, Size, and Security of Ascon-Sign

Ascon-Sign has the following 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

Note that $a = \log t$. Moreover, from these values the values m and len are computed as

- m: the message digest length in bytes. It is computed as

$$m = \lfloor (k \log t + 7)/8 \rfloor + \lfloor (h - h/d + 7)/8 \rfloor + \lfloor (h/d + 7)/8 \rfloor$$

While only $h + k \log t$ bits would be needed, using the longer m as defined above simplifies implementations significantly.

- 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

$$l_2 = \lceil \log(len_1(w-1))/\log(w) \rceil$$

Table 3: Hash calls in Ascon-Sign

	F	**H**	**PRF**	$\mathbf{T}_{\mathsf{len}}$
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

Table 3 gives a brief overview of the number of hash function calls we require for each operation in Ascon-Sign. Single calls to $\mathbf{H_{msg}}$, $\mathbf{PRF_{msg}}$, and $\mathbf{T_k}$ for signing and single calls to $\mathbf{H_{msg}}$ and $\mathbf{T_k}$

Algorithm 20 Ascon-Sign signature generation

```
# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SIG_ASCON
function ASCON-SIGN-SIGN(M, SK)
    Intialize ADRS

    # Generate randomizer
    opt = toByte(0, n)
    if Randomize then
        opt = rand(n)
    end if
    R = PRF_msg(SK.prf, opt, M);
    SIG_ASCON = SIG_ASCON || R

    # Compute message digest and index
    digest = H_msg(R, PK.seed, PK.root, M);
    tmp_md = first floor((ka + 7)/8) bytes of digest;
    tmp_idx_tree = next floor((h − h/d + 7)/8) bytes of digest;
    tmp_idx_leaf = next floor((h/d + 7)/8) bytes of digest;

    md = first ka bits of tmp_md;
    idx_tree = first h − h/d bits of tmp_idx_tree;
    idx_leaf = first h/d bits of tmp_idx_leaf

    # FORS sign
    ADRS.setLayerAddress(0);
    ADRS.setTreeAddress(idx_tree);
    ADRS.setType(FORS_TREE);
    ADRS.setKeyPairAddress(idx_leaf);
    SIG_FORS = FORS-SIGNATURE(md, SK.seed, PK.seed, ADRS);
    SIG_ASCON = SIG_ASCON||SIG_FORS;

    # Get FORS public key
    PK_FORS = FORS-PK-FROM-SIGN(SIG_FORS, M, PK.seed, ADRS);

    # Sign FORS public key with hypertree
    ADRS.setType(TREE);
    SIG_HT = HT-SIGN(PK_FORS, SK.seed, PK.seed, idx_tree, idx_leaf);
    SIG_ASCON = SIG_ASCON || SIG_HT;
    return SIG_ASCON
end function
```

Algorithm 21 Ascon-Sign verification

```
    # Input: Message M, public key PK = (PK.seed, PK.root), SIG_ASCON
    # Output: Boolean
function ASCON-VERIFY(M, PK, SIG_ASCON)
    Intialize ADRS
    R = SIG_ASCON.getR()
    SIG_FORS = SIG_ASCON.getSIG_FORS()
    SIG_HT = SIG_ASCON.getSIG_HT()

    # Compute message digest and index
    digest = H_msg(R, PK.seed, PK.root, M);
    tmp_md = first floor((ka + 7)/8) bytes of digest;
    tmp_idx_tree = next floor((h − h/d + 7)/8) bytes of digest;
    tmp_idx_leaf = next floor((h/d + 7)/8) bytes of digest;

    md = first ka bits of tmp_md;
    idx_tree = first h − h/d bits of tmp_idx_tree;
    idx_leaf = first h/d bits of tmp_idx_leaf

    ADRS.setLayerAddress(0);
    ADRS.setTreeAddress(idx_tree);
    ADRS.setType(FORS_TREE);
    ADRS.setKeyPairAddress(idx_leaf);

    PK_FORS = FORS-PK-FROM-SIGN(SIG_FORS, M, PK.seed, ADRS);

    ADRS.setType(TREE);
    return HT-VERIFY(M, SIG_HT, PK.seed, idxtree, idxleaf, PK_HT )
end function
```

for verification are omitted. because their effect on speed is negligible. Table 4 summarizes the size of secret key, public key, and signature in bytes for a given set of parameters.

Table 4: Key and signature sizes for Ascon-Sign

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

Table 5 discusses the example parameter for Ascon-Sign targeting different security levels and different tradeoffs between size and speed. Since the design is basically the same of SPHINCS+, we expect the same security claims [1, Table 3] would hold. Here, the suffix 's' denotes that the parameter set focus on size of the signature on the cost of lower speed, while the suffix 'f' denotes that the given parameter set focus on speed rather than the size of the signature. Based on the application scenario, appropriate parameter set can be chosen.

Table 5: Example parameter sets for Ascon-Sign

	n	h	d	$\log(t)$	k	w	Expected security level	Signature size
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 claim for Ascon-Sign

Ascon-Sign is based on the SPHINCS+ [3] signature framework with Ascon-Hash and Ascon-XOF as the internal hash function. Similar to SPHINCS+ [3], the security of Ascon-Sign is achieved through the inherent properties of the function families described in Section 3. These properties are derived from the characteristics of the ASCON hash functions used to instantiate those function families. Note that ASCON cipher suite is well analyzed, and therefore, Ascon-Sign is expected to have the same security strength as SPHINCS+.

6 Performance Analysis

To obtain performance benchmarks, we assess our reference implementation and optimized implementation on a machine with following hardware and software specification:

- 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`

For the parameter sets mentioned in Table 5, the *cycle counts* for reference and optimized implementation of Ascon-Sign 'simple' variant are mentioned in Table 6. In addition, We also list the performance result for reference and optimized implementations for robust version of Ascon-Sign in the Table 7. In Table 8, we list the key and signature sizes (in bytes) for the defined parameter sets.

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

	Key generation	Signing	Verification
	Reference Implementation		
Ascon-Sign-128s	315,840,896	2,413,174,678	2,429,047
Ascon-Sign-128f	5,939,611	115,382,780	6,972,950
Ascon-Sign-192s	599,392,072	5,458,909,051	4,696,353
Ascon-Sign-192f	10,939,221	243,023,163	13,058,030
	Optimized Implementation		
Ascon-Sign-128s	291,925,878	2,224,377,542	2,137,821
Ascon-Sign-128f	5,506,606	107,020,221	6,535,295
Ascon-Sign-192s	557,050,751	5,046,224,790	4,357,430
Ascon-Sign-192f	10,117,696	226,197,880	12,333,664

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

	Key generation	Signing	Verification
Ascon-Sign-128s	554,679,600	4,225,825,170	5,516,617
Ascon-Sign-128f	10,156,899	198,139,090	12,469,524
Ascon-Sign-192s	1,046,162,651	9,916,984,141	10,281,218
Ascon-Sign-192f	18,827,117	419,872,255	23,006,148
	Optimized Implementation		
Ascon-Sign-128s	530,089,300	4,038,032,800	4,232,362
Ascon-Sign-128f	10,678,534	182,601,975	11,279,318
Ascon-Sign-192s	970,639,431	8,893,090,510	7,664,451
Ascon-Sign-192f	17,174,517	381,735,599	21,408,883

Table 8: Key and signature sizes in bytes for Ascon-Sign

	Public key	Secret key	Signature
Ascon-Sign-128s	32	64	7856
Ascon-Sign-128f	32	64	17088
Ascon-Sign-192s	48	96	16224
Ascon-Sign-192f	48	96	35664

7 Advantages and Limitations

- Ascon-Sign is based on ASCON [7]. It is a lightweight AEAD which is recently selected by NIST for standardization of the lightweight cryptography[1].

[1] https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon

- Ascon-Sign is a variant of SPHINCS+ where the internal hash function is replaced by Ascon-Hash and Ascon-XOF. Therefore, the advantages and limitations SPHINCS+ is also inherited by Ascon-Sign.

References

[1] Aumasson, J.P., Bernstein, D.J., Beullens, W., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Westerbaan, B.: Sphincs+ – submission to the 3rd round of the nist post-quantum project. v3.1. NIST PQC (2022), https://sphincs.org/data/sphincs+-r3.1-specification.pdf 18

[2] Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: Sphincs: practical stateless hash-based signatures. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 368–397. Springer (2015) 3, 6

[3] Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs+ signature framework. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. pp. 2129–2146 (2019) 3, 6, 18

[4] Bertoni, G., Daemen, J., Peeters, M., van Assche, G.: Sponge functions. ecrypt hash workshop (2007) 4

[5] Beullens, W., Kleinjung, T., Vercauteren, F.: Csi-fish: efficient isogeny based signatures through class group computations. In: Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part I. pp. 227–247. Springer (2019) 3

[6] Ding, J., Schmidt, D.: Rainbow, a new multivariable polynomial signature scheme. In: ACNS. vol. 5, pp. 164–175. Springer (2005) 3

[7] Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1. 2: Lightweight authenticated encryption and hashing. Journal of Cryptology **34**, 1–42 (2021) 3, 4, 19

[8] Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 238–268 (2018) 3

[9] Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z., et al.: Falcon: Fast-fourier lattice-based compact signatures over ntru. Submission to the NIST's post-quantum cryptography standardization process **36**(5) (2018) 3

[10] Hülsing, A.: W-ots+–shorter signatures for hash-based signature schemes. In: International Conference on Cryptology in Africa. pp. 173–188. Springer (2013) 6

[11] Hülsing, A., Rausch, L., Buchmann, J.: 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. pp. 194–208. Springer (2013) 3

[12] Jang, K., Baksi, A., Song, G., Kim, H., Seo, H., Chattopadhyay, A.: Quantum analysis of AES. IACR Cryptol. ePrint Arch. p. 683 (2022), https://eprint.iacr.org/2022/683 3

[13] Merkle, R.C.: A certified digital signature. In: Advances in cryptology—CRYPTO'89 proceedings. pp. 218–238. Springer (2001) 3

[14] Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Prob. Contr. Inform. Theory **15**(2), 157–166 (1986) 3

[15] Patarin, J.: Hidden fields equations (hfe) and isomorphisms of polynomials (ip): Two new families of asymmetric algorithms. In: Advances in Cryptology—EUROCRYPT'96: International Conference on the Theory and Application of Cryptographic Techniques Saragossa, Spain, May 12–16, 1996 Proceedings 15. pp. 33–48. Springer (1996) 3

[16] Srivastava, V., Baksi, A., Debnath, S.K.: An overview of hash based signatures. Cryptology ePrint Archive, Paper 2023/411 (2023), https://eprint.iacr.org/2023/411 3