HBSS: (Simple) Hash-Based Stateless Signatures – Hash all the way to the Rescue!

Shlomi Dolev (dolev@cs.bgu.ac.il)
Ben-Gurion University of the Negev
Avraam Yagudaev
Ben-Gurion University of the Negev
Moti Yung
Google (United States)

Research Article

Keywords: Stateless signatures, Hash-based signatures, Cryptographic hash functions, Postquantum security

Posted Date: January 29th, 2024

DOI: https://doi.org/10.21203/rs.3.rs-3890355/v1

Additional Declarations: No competing interests reported.

HBSS: (Simple) Hash-Based Stateless Signatures – Hash all the way to the Rescue! *

Shlomi Dolev[1], Avraam Yagudaev[1], and Moti Yung[2]

[1] Department of Computer Science
Ben-Gurion University of the Negev
Beer Sheva, Israel
dolev@cs.bgu.ac.il, avraamy@post.bgu.ac.il
[2] Google, USA
moti@cs.columbia.edu

Abstract. Recent advancements in post-Quantum secure signing have revitalized interest in one-time signatures, such as Lamport's, and their many signature extensions. Predominantly based on standard hash functions, these signatures avoid reliance on number theoretic assumptions. Existing methods utilize a commitment array, with de-commitment contingent on the hashed message's representation bits. State-of-the-art variants incorporate pseudorandom functions.

This study introduces a novel method utilizing a probabilistic "set membership data structure" derived from hash functions. It involves accessing a long array with k independent hash functions for each message, analogous to Bloom filters. This stateless signature scheme is adjustable to accommodate any pre-set maximum number of signatures by modulating the array's length. The key concept is the partial loading of the de-committed array, ensuring validation of signed messages, non-validation of unsigned messages, and signature unforgeability (forgery equates to decommitment without the private key). This approach extends to improving one-time or bounded-message Constructions, like the Naor-Yung extension, for regular signature applications in the new Hash-Based Stateless Signature (HBSS) scheme.

Keywords: Stateless signatures · Hash-based signatures · Cryptographic hash functions · Post-quantum security

1 Introduction

Binding a public key to an entity is the hardest chain in connecting an entity in the real world, be it a person, a company, or an organization, to its digital representation. It is a cumbersome process that may involve certificate authorities, courts, lawyers, offline and online documents, cameras, biometrics, etc., binding the entity (detailed description) with the public key.

One would like to minimize the number of binding a public key (that fits a public value that is carefully kept privately by the user) to the entity he/she represents (either themselves or another physical or organizational entity).

Post-quantum one-way functions and cryptographic schemes based on: Lattice-based cryptography, multivariate cryptography, hash-based cryptography, code-based cryptography, supersingular elliptic curve isogeny cryptography, see e.g., [1]. Out of the above, non-number-theory based hash primitives, such as the SHAs, are not based on long-standing unsolved (in polynomial time) mathematical problems for which a solution may be (or secretly already has been) found; SHA512 is commonly believed to be post-quantum.

Lamport's signature can be based on any one-way function as a primitive, and in particular, those that are PQ-secure and hence are not proven to be breakable by the Shor algorithm [18]. In particular, the Secure Hashing Algorithm (SHA) family is believed to be quantum-safe. The main possible cavity of Lamport's signature is the one-time usage, requiring a new binding process of another public key for each signature. Merkle trees allow performing many signatures (exponentially growing number with the tree depth) but requires tracking the state, namely the leaves that are already used [14,15].

* Partially supported by the Rita Altura Trust Chair in Computer Science and the Israeli Science Foundation (Grant No. 465/22).

A complicated scheme to yield a stateless signature is presented in [10] whereas lately, more efficient stateless schemes, SPHINCS, were presented, [12,2]. The SPHINCS schemes, in principle, are based on Merkle trees and the idea of trees of trees (where the capability of further signing is increased dynamically), signing the next signing object is a basic idea originating from [16], where this idea was proposed as a way to base signatures on one-way only functions (without the trapdoor property, which was needed till then for secure signatures), yet has now been incorporated into NIST proposals.

See also [6,17,2,4] for more recent (hyper) tree based stateless schemes. We believe the various SPHINCS schemes are more complicated to implement and maintain than HBSS, the scheme we suggest here.

2 Related Work

Many signature schemes exist that do not rely on hash-based cryptography. These are based on functions believed to be one-way, including code-based, lattice-based, and multivariable cryptography. Our focus, however, is on hash-based schemes that can employ any one-way function primitive, particularly SHA and AES, which are planned for use in post-quantum secure communication (e.g., as part of symmetric encryption). Consequently, our primary competitor is the NIST-approved SPHINCS enhancement, SPHINCS+.

SPHINCS is distinguished by its unique "tree of trees" architecture. This structure comprises a complete binary hash tree, where the internal nodes values are hash values derived from the XOR operation on the concatenated values of their direct node descendants, integrated with the mask at their respective levels. The authentication path to a leaf includes all sibling nodes along the path from the selected leaf to the root. Each leaf in the tree functions as a public key for the WOTS+ L-trees, striking a balance between tree size and security, enabling smaller trees without compromising security. SPHINCS+ represents an enhancement over its predecessor, improving upon SPHINCS in terms of speed and signature size.

In contrast, our research proposes the HBSS scheme, which adopts a more streamlined approach. Whereas SPHINCS and SPHINCS+ feature complex cryptographic structures, HBSS emphasizes simplicity in implementation and maintenance, without sacrificing security and efficiency. This simplicity positions HBSS as an attractive alternative in scenarios where ease of implementation is paramount, offering a formidable solution in the field of stateless signatures.

3 Our HBSS Scheme

This scheme is based on several facets involving hash functions and it seems to be simpler (to implement and maintain) as a stateless bounded (multi-time) signature. This is so since it is inherently stateless. Further, HBSS has a new approach: it is inspired by Hash tables and Bloom filters (which is an array data structure that keeps membership of strings compactly), where a load factor α plays an essential performance/correctness factor.

A long array *Preimage* of dimensions $2 \times m$ is created (where m is a parameter), together with a *Commitment* array of the same dimensions. Then $2m$ random (or pseudorandom) numbers are created and assigned to the $2m$ entries of the *Preimage* array. Lastly, each *Commitment*$[i, j]$ entry is assigned by the SHA result of the *Preimage*$[i, j]$ entry. The value of the *Commitment* array is made public, binding with the entity description that holds the *Preimage* private.

A comment: usually, a certificate with both a public key (commitment array) and the entity description is signed by (the private key of) a certificate authority for which, in turn, the public key is known worldwide. Obviously, exposure of a private key of certificate authority can cause tremendous harm to internet security. This PKI infrastructure weakness is out of the scope of our paper (see, e.g., [7] for a relevant discussion).

Back to our technique: To sign a signature to a message, a message digest D of length k is computed, by using, for example, SHA512 (where the other particular hash functions can be chosen, e.g., SHA256) is the size of the hash function output which is tuned to the security level chosen.

For each message, it will be hashed by the collection of k independent random hash functions (assuming SHA is a random oracle) into indices in the array, and these indices will open (i.e., expose) their primages corresponding to the committed value at that index. In detail: k indices, $i_0, i_1, \ldots i_{k-1}$

are computed, by assigning the value of SHA(j, D) mod m, $0 \leq j \leq k-1$ to i_j. The signer then exposes $Preimage[D[j], i_j]$ for every $0 \leq j \leq k-1$.

Note that some of the *Preimage* entries can be already exposed; however, m is chosen to be large enough to yield a small ratio (say, less than 1/2) of exposed entries of *Preimage* with relation to m, taking in account all the signatures made in a lifetime.

There is a tradeoff between storing all the values of *Preimage* array versus computing them from s_0 and s_1 as needed. Note that it is possible to store several representative values of nested hashes for s_0 and s_1, spread in the $2m$ domain, and continue the nested hash from them. Thus, keeping a small memory and reducing the needed number of nested hashes when an entry of *Preimage* should be revealed.

Pseudocode for initializing the public and private arrays, *Commitment* and *Preimage*, respectively, appears in Algorithm 1. Algorithm 1 allocates two arrays, each of two dimensions. The first dimension corresponds to a digest bit being either 0 or 1, just as Lamport's signature does. The second dimension corresponds to the length of the digest. Random numbers, in the pseudocode, a particular concrete choice of 512 bits long is made, are assigned to the $2m$ entries of *Preimage* (lines 4-5). Then each of the entries of *Preimage* is hashed. Again a particular concrete choice of SHA512 is made to compute the corresponding entry of *Commitment* array (lines 6-7).

The pseudocode for signing appears in Algorithm 2. The pseudocode starts with the digest, D, computation, here too, SHA512 is the particular concrete hash function used (line 1). Then, an array, *Signature*, of $k = |D|$ entries, each of size *KeySize* bits, is allocated (line 2). Each *Signature* entry is assigned by an entry from the *Preimage* array. Where the j'th index of the *Signature* array is assigned by one of the two i_j entries of *Preimage*, where i_j is a result of a hash function[3] over the signed *Message* (line 4). The choice of which of the two entries of the i_j index is assigned to the j'th entry of *Signature* is made by the value of the j'th bit in the digest (lines 5 to 9).

The pseudocode for signature verification appears in Algorithm 3. First, the message digests D is calculated (line 1). Then the *Signature* entries are verified to correspond to the indexes i_j, as computed in the signature process, and the value of the j'th bit of the digest. A *True* value is returned only when all entries of *Signature* are correct, and *False* is returned otherwise.

Algorithm 1 HBSS – keygen()

1: $Preimage \leftarrow array\ [2, m]$ of $KeySize$
2: $Commitment \leftarrow array\ [2, m]$ of $KeySize$
3: **for** $i = 0$ to $m - 1$ **do**
4: $\quad Preimage[0][i] \leftarrow Random(512bits)$
5: $\quad Preimage[1][i] \leftarrow Random(512bits)$
6: $\quad Commitment[0][i] \leftarrow SHA512(Preimage)[0][i]$
7: $\quad Commitment[1][i] \leftarrow SHA512(Preimage)[1][i]$
8: **end for**
return $key(Preimage, Commitment)$

Algorithm 2 HBSS – sign(Message, Preimage)

1: $D \leftarrow$ hash with SHA512$(Message)$
2: $Signature \leftarrow array(k \cdot KeySize)$
3: **for** $j = 0$ to $k - 1$ **do**
4: $\quad i_j \leftarrow$ SHA512$(j, Message)$ mod m
5: $\quad$ **if** $D[j] = 0$ **then**
6: $\quad\quad Signature[j] \leftarrow Preimage[0, i_j]$
7: $\quad$ **else**
8: $\quad\quad Signature[j] \leftarrow Preimage[1, i_j]$
9: $\quad$ **end if**
10: **end for**
return $Signature$

[3] Here SHA512 over the index j together with the message, is used to make each index pseudo-random and pseudo-independent. Many other possibilities for using k results of hash functions as done in Bloom filter are possible as well.

Algorithm 3 HBSS – Verify(Message,Signature,Commitment)

$D \leftarrow$ hash with SHA512($Message$)
for $j = 0$ to $k - 1$ **do**
  $i_j \leftarrow$ SHA512$(j, Message)$ mod m
  if $D[j] = 0$ **then**
    if (SHA512$(Signature[j]) \neq Commitment[0, i_j]$) **then return** *False*
    end if
  else
    if (SHA512$(Signature[j]) \neq Commitment[1, i_j]$) **then return** *False*
    end if
  end if
end for
return $True$

The size of m is a function of the upper bound on n, the number of signatures made during the lifetime of the system/entity, and k the number of entries of *Preimage* exposed (αm of which, for the first time) with each signature. If we chose $m \geq 2nk$, then even the last signature of the n exposes, for the first time, an expected number of $k/2$ entries of the *Preimage* array. Thus, if k is 512 bits (when using SHA512), then it is expected that 256 of them are verified to be originated by the signer, which in turn may imply a sufficient security level. Hence, the choice of k, the upper bound on α, and the size of each entry of the *Preimage* array can be tuned to imply the required security level.

Note that several one-way functions (OWF) can be chosen; (a) one that is used to define the entries of *Commitment* array as the OWF function of the entries of *Preimage* array, (b) one that is used to define D the digest of messages, and (c) one that defines the k indices in m for which a corresponding entry (of the two) of *Preimage* should be revealed.

Next, we outline the claims that ensure the post-quantum security of our scheme.

Theorem 1. *HBSS is post-quantum secure.*

Proof.

We establish our signing scheme parameters by the following guidelines: (a) Entries of *Preimage* are random or generated by a sufficiently strong pseudorandom mechanism, potentially utilizing SHA256. Each entry consists of an adequate number of bits, for example, 256 bits. Predicting or calculating the values of the *Preimage* entries is computationally infeasible unless they are explicitly disclosed. (b) Entries of *Commitment* result from a robust one-way cryptographic hash, such as SHA256, rendering them computationally infeasible to invert. (c) α is maintained small, ideally $\alpha < 1/2$, or confined to a small proportion relative to an upper bound on n. (d) k is sufficiently large, for instance, $k \geq 512$. To achieve 512 bits digest from the message msg, one might concatenate the result of SHA256$(1, msg)$ with SHA256$(2, msg)$. Under these stipulations, even for the final signature, the expected number of newly exposed entries in the Preimage array exceeds 256. This number is determined by the signer. (e) The probing indices for revealing entries of the *Preimage* array are the outcomes of a post-quantum hash function, such as SHA256.

Consider M as the set of messages signed thus far, exposing entries of the *Preimage* array. Our policy selections imply that it is fundamentally impossible, even with a quantum computer employing Grover's algorithm [11], to find a message $msg \notin M$ that utilizes only the previously exposed entries of the *Preimage* array. Note that the necessity to invert SHA256 arises even with one unexposed entry. Mining a message (analogous to a blockchain preimage search) that results in indices of only already exposed entries in the *Preimage* array necessitates scanning an expected number of potential messages on the order of 2^{256}. This task is virtually impossible, even when equipped with a quantum computer. Thus, the following requirements hold:

• *Soundness: If the signer did not sign, there is a way to show it, given access to the signer's signing history.*

This is clear when at least one of the signature entries does not correspond, via hash, to the public key array entry. If all signature entries correspond to the public key array, finding a message with a signature that is only mapped to the revealed entries is essentially impossible.

- *Correctness: If the signer signed, there is no way to repudiate.*
  See the non-deniability arguments above.

- *Unforgeability: The signer or any poly-time adversary cannot forge a new message.*
  The choice of parameter choices implies that forging a new message is essentially impossible.

□

4 One Dimensional Bloom Filter

The new technique used here is the hash mapping of the indexes to the array. Apparently, a single-dimension Bloom filter will fully capture the power of our suggestion. Instead of mapping the digest value 0 digits to a single dimension array and the value one digit to a separate one, we can use k hash functions from the message itself or from the digest of the message.

The single-dimension solution is equivalent to the two dimensions solution. We choose one of the two single-dimensional arrays with equal probability and then choose indexes within the chosen array.

The security level, the resources, and the performance are functions of the particular choices made; for which the detailed discussion is omitted from this short version.

Bloom filter [5] common analysis exhibits a false positive probability for n items each using k hash mappings in an array of size m to be less than: $(1 - e^{-(k(n+0.5)/(m-1)})^k$.

The probability for an entry of *Preimage* to be not yet revealed is $p = (1 - 1/m)^{kn} \approx e^{-kn/m}$. Which yield $me^{-kn/m}$ expected number of unrevealed *Preimage* entries, or expected $\alpha = me^{-kn/m}/m$.

Choosing k to be optimal for n and m, one should choose $k = (m \ln 2)/n$, yielding $\alpha = 1/2$ after the last signature is issued. Thus, the expected number of entries of the *Preimage* revealed for the last signature, when $k = 1024$, is 512.

A malicious signer may act toward a deniability claim, choosing many messages that together reveal all the entries a message msg reveals. By doing so, the signer wishes to claim later that he/she did not sign msg. However, the malicious signer may need to "mine" (in the head) messages that (together) expose the k indexes that msg exposes. The probability of hitting one index of these k indexes is $1/m$. The expected number of messages that should be examined to have full coverage for the k entries of msg is m^k/k; for example, when $m = 2$ Billion and $k = 1024$, the adversary is doomed to surrender.

If $k = 1024$, then the number of signatures n, is approximate to be $n = m/1477$, say we use one TeraBytes of memory; in fact, for now, we use even two TeraBytes for *Preimage* and *Commitment*, in the sequel, we suggest ways to reduce the need for storage for the *Preimage* array or using Merkle tree (and re-computation of the tree) for the *Commitment* array. Divided the two TeraBytes by 512 bits for each entry for both *Preimage* and *Commitment*, we get $m = 2$ billion entries (as one TeraByte divided by half KiloByte is more than two billion), which implies n, to be more than 1.3 million signatures.

To avoid storing the *Preimage* array, one may use the technique suggested in [8] that uses two random seeds s_0 and s_1 that are used to produce the $2m$ entries of *Preimage*. Namely, to produce a pseudo-random sequence from a one-way function, such as SHA. To simplify the discussion, consider the entries of *Preimage* as a single sequence where $Preimage[1, i]$ immediately follows $Preimage[0, i]$. To produce the j'th entry in the above $2m$ (or m, in terms of the one-dimensional case) length sequence, s_0 is hashed j times. Namely, the result of $r_1 = \text{SHA}(s_0)$ is hashed again to obtain $r_2 = \text{SHA}(r_1)$ and so on, until r_j is reached. Then s_1 is hashed $2m - j$ times to obtain t_{2m-j}. At last the corresponding entry of *Preimage* is assigned by the xor of r_j and t_{2m-j}, namely, $r_j \oplus t_{2m-j}$. The xor operation serves as a "lock" for the possibility of predicting other entries of *Preimage* (by the use of SHA) once the value of *Preimage* in the j'th index is revealed.

We utilized this approach to implement the HBSS* scheme, which conserves memory space by retaining only a portion of the *Preimage* array.

For the HBSS* scheme, a parameter named *Step* is introduced. This parameter dictates the size of the *Preimage* array. After each *Step*, two preimage values are computed based on two seeds. With these, any desired preimage can be regenerated, showcasing a tradeoff in memory space. During the signature process, the targeted preimage value is deduced. The signature size is consistent with the HBSS scheme, and the verification process is analogous to that of HBSS.

Algorithm 4 HBSS* – keygen()

$Preimage \leftarrow [2, \text{Step}]$
$Seeds \leftarrow [2, 2m/\text{Step}]$
$Commitment \leftarrow$ array $[2, m]$ of $KeySize$
$Seeds[0][0] \leftarrow$ Random(256 bits)
$Seeds[1][0] \leftarrow$ Random(256 bits)
for $i = 0$ to $2m/\text{Step} - 1$ **do**
    $Seeds[0][i+1] \leftarrow Seeds[0][i]$
    $Seeds[1][i+1] \leftarrow Seeds[1][i]$
    for $j = 0$ to Step **do**
        $Seeds[0][i+1] \leftarrow \text{SHA256}(Seeds[0][i+1])$
        $Seeds[1][i+1] \leftarrow \text{SHA256}(Seeds[1][i+1])$
    end for
end for
for $i = 0$ to $2m/\text{Step}$ **do**
    $Preimage[0][0] \leftarrow Seeds[0][i]$
    $Preimage[1][0] \leftarrow Seeds[1][2m/\text{Step} - i - 1]$
    for $j = 0$ to $\text{Step} - 1$ **do**
        $Preimage[0][j+1] \leftarrow \text{SHA256}(Preimage[0][j])$
        $Preimage[1][j+1] \leftarrow \text{SHA256}(Preimage[1][j])$
    end for
    for $j = 0$ to Step **do**
        $Commitment[j + i \cdot \text{Step}] \leftarrow \text{SHA256}(Preimage[0][j] \oplus Preimage[1][\text{Step} - j - 1])$
    end for
end for
return $key(Seeds, Commitment)$

Algorithm 5 HBSS* – sign(Message, Seeds)

$D \leftarrow$ hash with $\text{SHA256}(Message)$
$Signature \leftarrow \text{array}(k \cdot \text{KeySize})$
for $j = 0$ to $k - 1$ **do**
    $i_j \leftarrow \text{SHA256}(j, Message) \mod 2m$
    $Preimage[0] \leftarrow Seeds[0][i_j/\text{Step}]$
    $Preimage[1] \leftarrow Seeds[1][(2m - i_j)/\text{Step} - 1]$
    $bit \leftarrow D[j]$
    for $k = 0$ to $(i_j + bit) \mod \text{Step}$ **do**
        $Preimage[0] \leftarrow \text{SHA256}(Preimage[0])$
    end for
    for $k = 0$ to $\text{Step} - ((i_j + bit) \mod \text{Step}) - 1$ **do**
        $Preimage[1] \leftarrow \text{SHA256}(Preimage[1])$
    end for
    $Signature[j] \leftarrow Preimage[0] \oplus Preimage[1]$
end for
return $Signature$

Algorithm 6 HBSS* – verify(Message, Signature, Commitment)

$D \leftarrow$ hash with $\text{SHA256}(Message)$
for $j = 0$ to $k - 1$ **do**
    $i_j \leftarrow \text{SHA256}(j, Message) \mod 2m$
    $bit \leftarrow D[j]$
    if $\text{SHA256}(Signature[j]) \neq Commitment[i_j + bit]$ **then return** $False$
    end if
end for
return $True$

5 HBSS with Committeed Merkle (Tree and) Root

When we are restricted in the commitment storage, we can employ the Merkle tree scheme over our *Commitment* array entries and publish only the tree's root. Computing and exposing in the signature the relevant paths (in fact, a subtree) to the root from the leaves commitments and the corresponding preimage of these leaves.

There is a tradeoff between the storage and processing used in different settings for the preimages (whether seed(s) for reproducible pseudorandom or actual random values), the storage used by the signing party for the commitment array (whether actual commitment array or Merkle tree for the commitment), the storage is publicly verified and maintained by authority/blockchain to be associated with the signer (whether the entire commitment array or only the Merkle root value). The particular setting implies different messages signature lengths.

To illustrate, the HBSS** scheme seamlessly integrates the $Merkletree$ with the $Preimage$ array, emphasizing the unique use of the tree's root for signature verification.

Algorithm 7 HBSS** – keygen()

$Preimage \leftarrow$ array[$2m$] of $KeySize$
for $i = 0$ to $2m$ **do**
    $Preimage[i] \leftarrow$ Random(256 bits)
end for
$Tree \leftarrow$ MerkleTree($Preimage$)
return $key(Tree, Preimage)$

Algorithm 8 HBSS** – sign(Message,Tree,Preimage)

$D \leftarrow$ hash with SHA512($Message$)
$Signature \leftarrow$ array($k \cdot (2 + \log_2(m))$ of $KeySize$
for $j = 0$ to $k - 1$ **do**
    $i_j \leftarrow$ SHA256($j, message$) mod $2m$
    $bit \leftarrow D[j]$
    $Signature[j] \leftarrow Preimage[i_j + bit] +$ authpath($i_j + bit$)
end for
return $Signature$

Algorithm 9 HBSS** – verify(Message, Signature, root)

$D \leftarrow$ hash with SHA512($Message$)
for $j = 0$ to $k - 1$ **do**
    $i_j \leftarrow$ SHA256($j, message$) mod $2m$
    $path\_number = i_j + bit$
    $path \leftarrow$ SHA256($signature[j][0]$)
    for $i = 1$ to $\log_2(2m) + 1$ **do**
        if $path\_number$ mod $2 == 0$ **then**
            $path \leftarrow$ SHA256($path||signature[j][i]$)
        else
            $path \leftarrow$ SHA256($signature[j][i]||path$)
        end if
        $path\_number = path\_number \div 2$
    end for
    if $path \neq root$ **then return** $False$
    end if
end for
return $True$

In the HBSS** scheme, instead of generating a *Commitment* array, a comprehensive Merkle tree is built in tandem with the Preimage array. During the signing phase, the path of the preimage's hash value in the Merkle tree—leading up to the root—is disclosed alongside the desired preimage value. For verify, only the root value of the Merkle tree is necessary.

6 Experiments

Table 1. Speed Evaluation

Scheme	**Key gen [cycles]**	**Sign [cycles]**	**Verify [cycles]**	**Pub key [Bytes]**	**Sec key [Bytes]**	**Sig [Bytes]**	**Signatures**	**NIST level**	**Method**
dilithium5	300,140	513,640	296,120	2,592	4,864	4,595		5	Lattice
sphincs-sha256-256f-robust	19,856,910	384,169,000	8,565,840	64	128	49,856	2^{64}	5	Hash
sphincs-sha256-256f-simple	4,899,140	102,604,610	3,472,430	64	128	49,856	2^{64}	5	Hash
sphincs-sha256-256s-robust	315,851,400	1,875,061,810	3,649,740	64	128	29,792	2^{64}	5	Hash
sphincs-sha256-256s-simple	77,007,920	566,442,880	1,287,760	64	128	29,792	2^{64}	5	Hash
HBSS* (1)	559,954,374,550	930,920	968,570	$32 \cdot 2^{30}$	$32 \cdot (2^{31}+2)$	16,384	$\approx 2^{19}$	5	Hash
HBSS* (4)	759,982,140,480	1,148,950	955,970	$32 \cdot 2^{30}$	$32 \cdot (2^{29}+2^3)$	16,384	$\approx 2^{19}$	5	Hash
HBSS**	4,038,277,757,770	7,483,620	4,304,980	32	$32 \cdot 3 \cdot 2^{30}$	507,904	$\approx 2^{19}$	5	Hash

The speed evaluation tests were conducted using the AMD EPYC 7702P 64-Core Processor. The results for the dilithium5 and sphincs+ schemes are consistent with those reported in publications [9] and [3]. We have also included the results of our schemes, which are elucidated further in the subsequent figures.

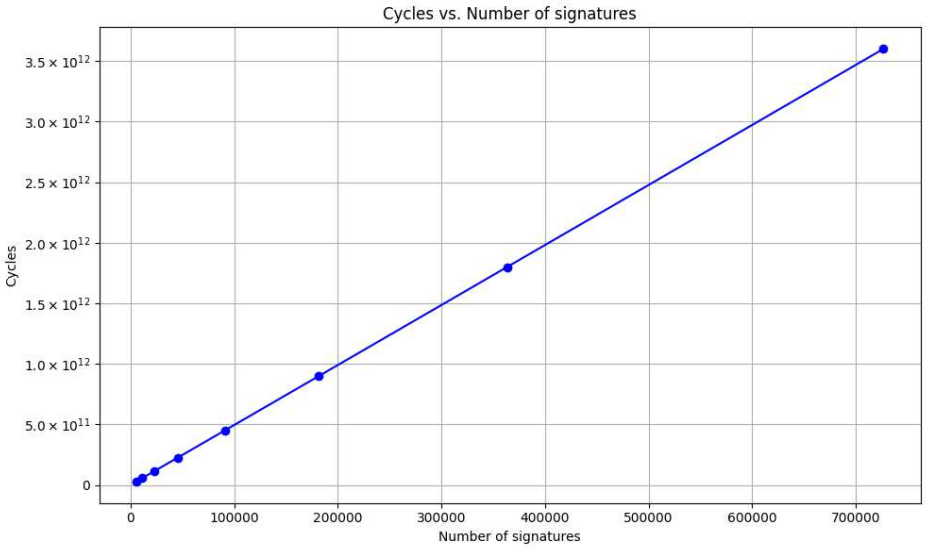

Fig. 1. HBSS-KeyGen
The HBSS key generation process involves the calculation of preimage and commitment values. As noted, both the Preimage and Commitment sizes are identical in HBSS, encompassing all values.

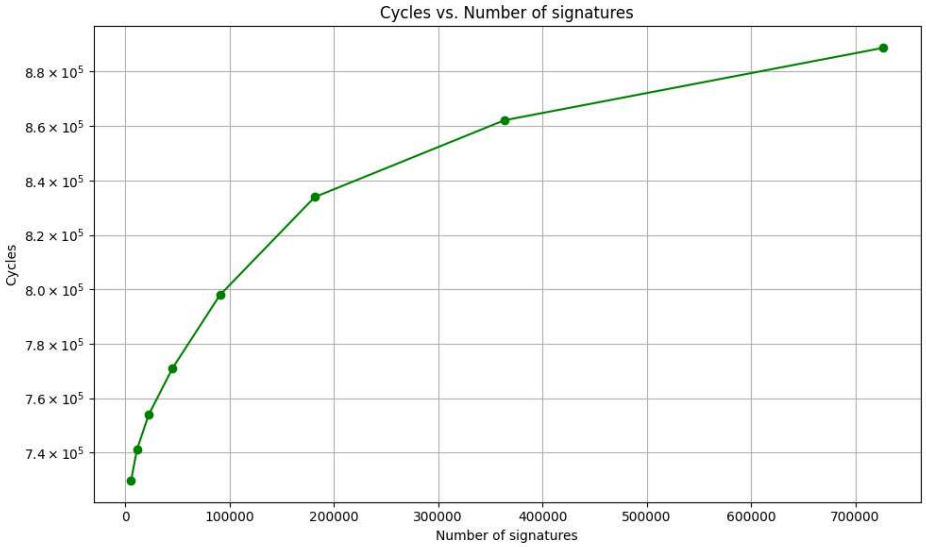

Fig. 2. HBSS-Sign
HBSS's signing process is influenced by the randomness of the preimage values, accounting for variations in cycle times. The signature size in HBSS remains consistent.

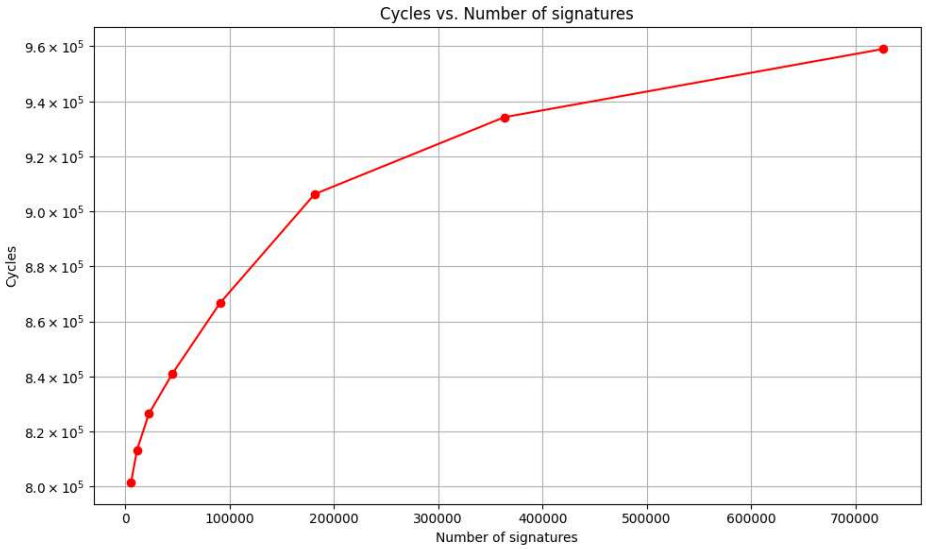

Fig. 3. HBSS-Verify
The verification time for HBSS remains analogous to that of the HBSS* scheme.

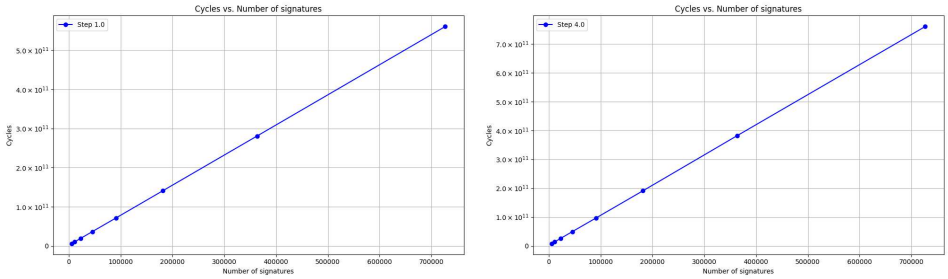

Fig. 4. HBSS*-KeyGen

In the HBSS* variant, the key generation time remains consistent regardless of the *Step* parameter. Notably, HBSS* boasts the most efficient key generation process among all the investigated variants. The extended duration observed in HBSS and HBSS** is due to the requirement of computing a pseudo-random value for each preimage. Conversely, HBSS* requires only two pseudo-random value calculations for the seeds and subsequently employs SHA256. In our specific implementation, the execution of SHA256 is faster than the generation of a pseudo-random value.

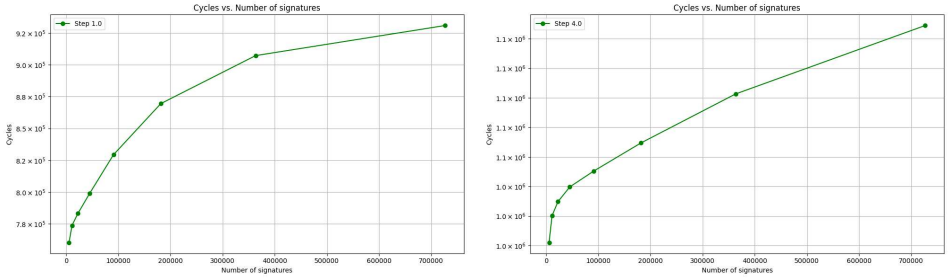

Fig. 5. HBSS*-Sign

There is a direct linear correlation between the size of the *Step* and the signing cycle time. The signature size for HBSS* is consistent with that of the original HBSS. Notably, in HBSS and HBSS**, the signing duration hinges on the size of the *Preimage* array, which is already established and merely requires access. In contrast, for HBSS*, the signing time is determined by the chosen *Step* size. To identify the desired preimage in HBSS*, one must execute *Step* iterations.

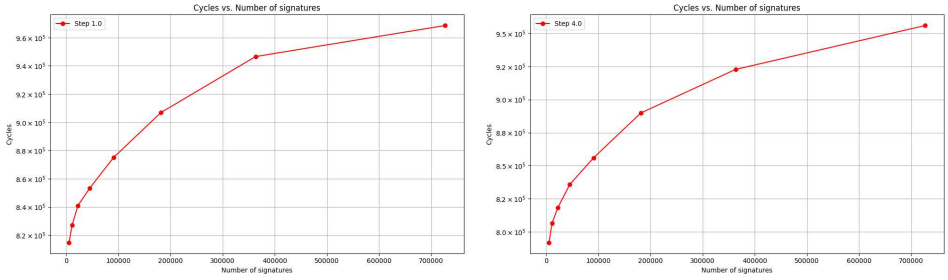

Fig. 6. HBSS*-Verify

The verification time for HBSS* mirrors that of the original HBSS scheme.

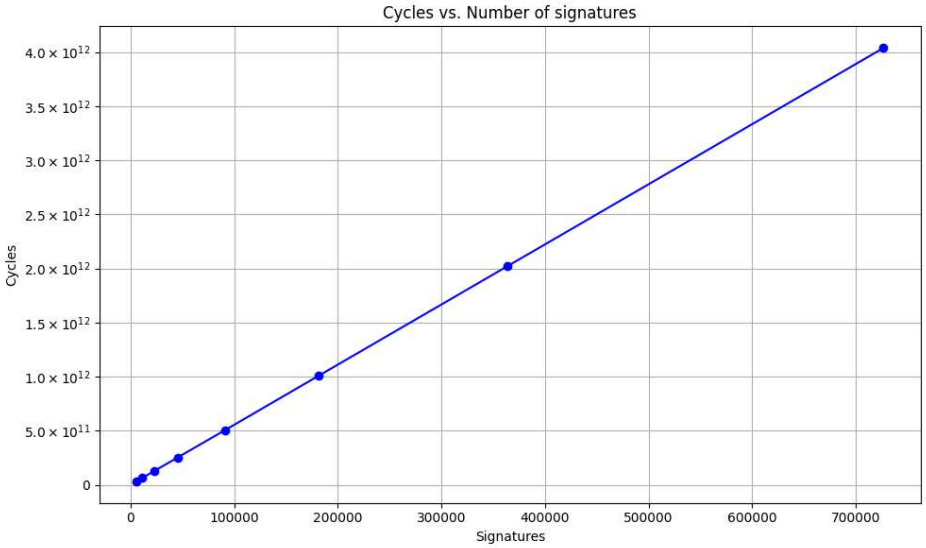

Fig. 7. HBSS**-KeyGen
HBSS** takes more time during key generation compared to its predecessors. This increase is attributed to the necessity of computing a larger Merkle tree in lieu of the Commitment array present in HBSS.

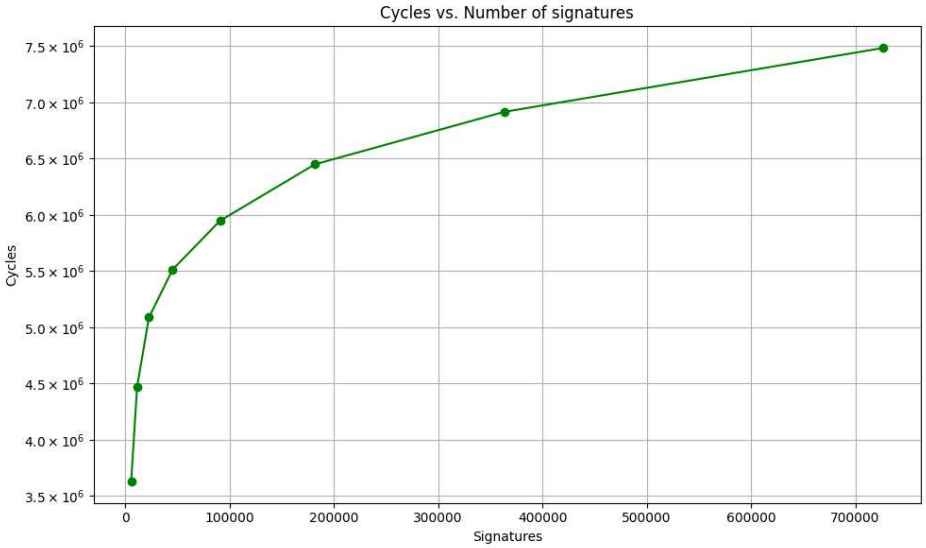

Fig. 8. HBSS**-Sign
The signing process in HBSS** reveals not only the desired preimage value but also the path of the preimage's hash value in the Merkle tree, up to the root.

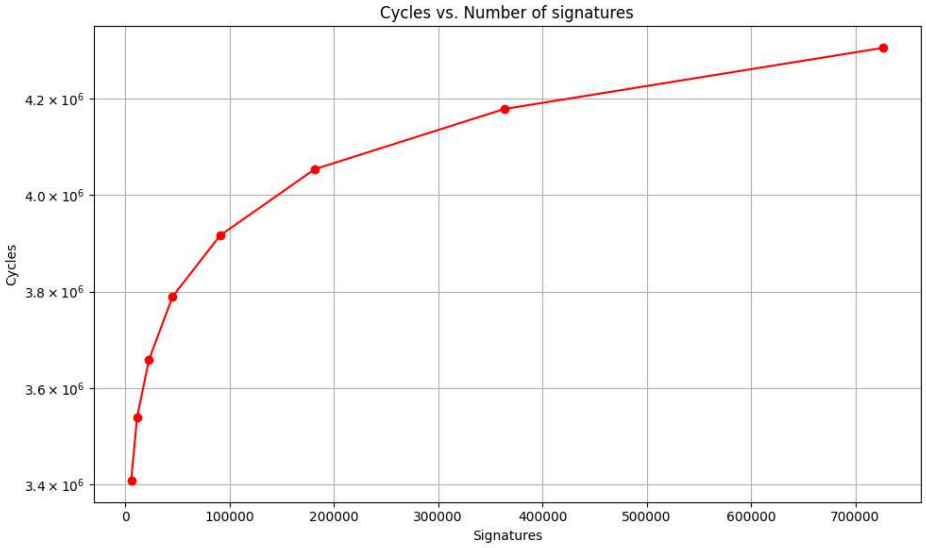

Fig. 9. HBSS**-Verify
The verification process in HBSS** is inherently more extensive due to its reliance on the Merkle tree's root value for verification. Coupled with its larger signature size, this makes the HBSS** verification time longer compared to HBSS and HBSS*.

7 Extensions

The entity may sign a new public key (extension) once the ratio α of the number of exposed entries of *Preimage*, over m, is bigger than a threshold. Thus, the signer may upper bound the number of signatures made during a period, say a year, and sign and commit to a new additional *Commitment* array at the end of the period. Such a signature over the new *Commitment* implies the binding of the new *Commitment* to the entity. The verifier will not accept signatures from a *Commitment* array for which the threshold is violated and will verify the binding chain of the (entries indexes in the) *commitment* array used for the signature; a binding chain that leads to the first *Commitment* array. The signature will be regarded as valid only when the binding chain leads to the (first) trusted authentication process. Obviously, the signer will not reveal too many entries of any *Preimage*, avoiding the possibility of forging his/her signature by others.

Note that finding an input (collision) when several (say, half of the) bits of the inputs for the cryptographic hash function, e.g., SHA512, are known resembles the mining task in blockchain; mining for an input with so many zeroes. The level of security can be tuned to be harder when choosing SHA with more bits. To facilitate, for our purposes, an SHA result of 1024 bits (or more), we can define D to be a concatenation of i results of SHA512, where, say, the j'th, $1 \leq j \leq i$ concatenated result is SHA512(j, m).

All along, we suggest using a post-quantum hash function, such as SHA512, for which there exist fast and efficient implementations, see, e.g., [13].

The complete code can be found in [19] .

References

1. Bernstein, D.J.: Post-quantum cryptography. In: van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, 2nd Ed, pp. 949–950. Springer (2011). https://doi.org/10.1007/978-1-4419-5906-5_386, https://doi.org/10.1007/978-1-4419-5906-5_386
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: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9056, pp. 368–397. Springer (2015). https://doi.org/10.1007/978-3-662-46800-5_15, https://doi.org/10.1007/978-3-662-46800-5_15

3. 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: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. pp. 368–397. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
4. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs$^+$ signature framework. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 2129–2146. ACM (2019). https://doi.org/10.1145/3319535.3363229, https://doi.org/10.1145/3319535.3363229
5. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970). https://doi.org/10.1145/362686.362692, https://doi.org/10.1145/362686.362692
6. Bos, J.N., Chaum, D.: Provably unforgeable signatures. In: Brickell, E.F. (ed.) Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings. Lecture Notes in Computer Science, vol. 740, pp. 1–14. Springer (1992). https://doi.org/10.1007/3-540-48071-4_1, https://doi.org/10.1007/3-540-48071-4_1
7. Dolev, S.: Overlay Security: Quantum-Safe Communication over the Internet Infrastructure. IntechOpen (2019). https://doi.org/http://dx.doi.org/10.5772/intechopen.78088
8. Dolev, S.: System and method for Merkle puzzles symmetric key establishment and generation of Lamport Merkle signatures (May 2019), uS Patent 0140819
9. 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 **2018**(1), 238–268 (Feb 2018). https://doi.org/10.13154/tches.v2018.i1.238-268, https://tches.iacr.org/index.php/TCHES/article/view/839
10. Goldreich, O.: Two remarks concerning the goldwasser-micali-rivest signature scheme. In: Odlyzko, A.M. (ed.) Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings. Lecture Notes in Computer Science, vol. 263, pp. 104–110. Springer (1986). https://doi.org/10.1007/3-540-47721-7_8, https://doi.org/10.1007/3-540-47721-7_8
11. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Miller, G.L. (ed.) Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996. pp. 212–219. ACM (1996). https://doi.org/10.1145/237814.237866, https://doi.org/10.1145/237814.237866
12. Hülsing, A., Kudinov, M.A.: Recovering the tight security proof of sphincs$^+$. In: Agrawal, S., Lin, D. (eds.) Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 13794, pp. 3–33. Springer (2022). https://doi.org/10.1007/978-3-031-22972-5_1, https://doi.org/10.1007/978-3-031-22972-5_1
13. Martino, R., Cilardo, A.: SHA-2 acceleration meeting the needs of emerging applications: A comparative survey. IEEE Access **8**, 28415–28436 (2020). https://doi.org/10.1109/ACCESS.2020.2972265, https://doi.org/10.1109/ACCESS.2020.2972265
14. Merkle, R.: Secrecy, authentication and public key systems pp. 32–61 (1979)
15. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings. Lecture Notes in Computer Science, vol. 293, pp. 369–378. Springer (1987). https://doi.org/10.1007/3-540-48184-2_32, https://doi.org/10.1007/3-540-48184-2_32
16. Naor, M., Yung, M.: Universal one-way hash functions and their cryptographic applications. In: Johnson, D.S. (ed.) Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA. pp. 33–43. ACM (1989). https://doi.org/10.1145/73007.73011, https://doi.org/10.1145/73007.73011
17. Reyzin, L., Reyzin, N.: Better than biba: Short one-time signatures with fast signing and verifying. In: Batten, L.M., Seberry, J. (eds.) Information Security and Privacy, 7th Australian Conference, ACISP 2002, Melbourne, Australia, July 3-5, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2384, pp. 144–153. Springer (2002). https://doi.org/10.1007/3-540-45450-0_11, https://doi.org/10.1007/3-540-45450-0_11
18. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994. pp. 124–134. IEEE Computer Society (1994). https://doi.org/10.1109/SFCS.1994.365700, https://doi.org/10.1109/SFCS.1994.365700
19. Yagudaev, A.: Hbss. https://github.com/avraamya/HBSS (2023), gitHub repository