Smart Digital Signatures (SDS)

Shahid, Furqan; Khan, Abid

Published in:
Future Generation Computer Systems

DOI:
10.1016/j.future.2020.04.042

Publication date:
2020

Citation for published version (APA):
Shahid, F., & Khan, A. (2020). Smart Digital Signatures (SDS): A post-quantum digital signature scheme for distributed ledgers. *Future Generation Computer Systems*, *111*, 241-253. https://doi.org/10.1016/j.future.2020.04.042

Smart Digital Signatures (SDS): A Post-quantum Digital Signature Scheme for Distributed Ledgers

Furqan Shahid[a], Abid Khan[b]

[a]*COMSATS University Islamabad (CUI), Islamabad, Pakistan*
[b]*Department of Computer Science, Aberystwyth University, Aberystwyth, SY23 3DB, Wales, UK*

Abstract

The upcoming quantum era is believed to be an end for the elliptic curve digital signature algorithm (ECDSA) and other number-theoretic digital signature schemes. Hence, the technologies which incorporate ECDSA would be at risk once quantum computers are available at large scale. Distributed ledger technology is one of the potential victims of powerful quantum computers. Fortunately, post-quantum digital signature schemes are already available. Hash-based digital signatures (HBS) schemes due to their simplicity and efficiency have gained tremendous attention from the research community. However, large size of key and signature are the major drawbacks of HBS schemes. This paper proposes a compact and efficient HBS scheme "Smart Digital Signatures" (SDS), which is closer to an existing popular HBS scheme, XMSS. *SDS* incorporates a novel one-time signature (OTS) scheme in XMSS, namely SDS-OTS. Furthermore, *SDS* uses a slightly modified version of the key compression tree as compared to XMSS. We have compared *SDS* with XMSS-WOTS and XMSS-WOTS$^+$. The results reveal a significant reduction in hash tree construction time compared to XMSS, and key and signature sizes compared to WOTS and WOTS$^+$. Finally, we have also proposed a model for incorporating *SDS* into a distributed ledger, with the help of High-Level Petri-nets.

Keywords: Distributed ledger, Digital signature, Post-quantum cryptography

1. Introduction and Background

The popular digital signature schemes, RSA and ECDSA, would be at significant risk after the invention of the first sufficiently powerful quantum

computer [1]. Recently, it has been proven that the quantum technology has the potential to solve hard computation problems in a matter of few seconds which could otherwise take several hundred years [2]. Although, recently available quantum processors are not powerful enough to break ECDSA, however, the advancement trends of technology allow us to expect a quantum computer being able to break ECDSA after just a decade [3]. An algorithm to break RSA and ECDSA with the help of a sufficiently powerful quantum computer has already been proposed by Peter Shor [4]. So what will be the situation? Will quantum computers erase digital signature technology at all? Thankfully, the answer is "no", because alternate digital signature schemes are already available, which can defeat quantum attacks. The digital signature schemes having the potential to withstand quantum attacks are commonly referred to as post-quantum (PQ) digital signature schemes. There are total five types of post-quantum digital signature schemes available to-date, including, lattice-based digital signature schemes, hash-based digital signature schemes, elliptic curve isogeny based digital signature schemes, multivariate digital signature schemes, and code-based digital signature schemes [5]. Although, PQ signature schemes are not newer, however, none of them could attract practitioners at a large scale, because of their low efficiency, improvable security, and large key/signature sizes [5], [6].

The hash-based digital signature (HBS) schemes are fairly efficient and provably secure [7]. Many popular PQ distributed ledgers, like Tangle [8], QRL [9], PQChain [7], and DL-for-IoT [10] have already adopted HBS schemes. However, HBS schemes suffer from larger key and signature sizes [11]. The three building blocks of a modern HBS scheme include a core one-time signature (OTS) or a few-time signature (FTS) scheme, a key compression tree, and a main hash tree. The core OTS/FTS is used to sign the message. A key compression tree compresses a given OTS/FTS public key into a single value to allow it to exist as a leaf node of the main hash tree. Finally, the main hash tree computes a single public key from a limited or a virtually unlimited number of OTS/FTS public keys. Although a public key encapsulating a large number of OTS public keys is desirable, however, distributed ledgers represent one of the exceptional cases. Distributed ledger technology recommends limited use of a single ledger address to preserve user's financial privacy. Although a pure OTS scheme (which never allows reusing a key) can offer the strongest financial privacy, however, it is exhaustive in many situations. For example, it is exhaustive for an NGO to provide a fresh ledger address to every donor, each time to receive new donations; or, for a coffee shop manager to generate a fresh ledger address each time to

receive payment against a cup of coffee. Therefore, there must be a way for reusing a single ledger address at least for a limited number of times without a security compromise. Hence, the HBS schemes which allow limited reuse of a single public key are suitable for distributed ledgers as compared to the more advanced HBS schemes.

This paper proposes Smart Digital Signatures (SDS) which is a compact and efficient post-quantum digital signature scheme. SDS is closer to an existing popular HBS scheme, XMSS [12]. SDS incorporates a novel one-time signature (OTS) scheme into XMSS, namely "SDS-OTS". Furthermore, SDS uses a slightly modified version of the key compression tree as compared to XMSS. We have compared SDS with two different instantiations of XMSS, namely, XMSS-WOTS and XMSS-WOTS$^+$. In the first instantiation, we incorporated XMSS with the OTS scheme "WOTS" [13], whereas, in the second instantiation, we incorporated XMSS with WOTS$^+$ [11]. The results reveal that SDS is 74% faster than XMSS-WOTS$^+$ and 30% faster than XMSS-WOTS.

The underlying OTS scheme of SDS, i.e. SDS-OTS, is the most compact OTS scheme as compared to all of the existing OTS/FTS schemes. WOTS and WOTS$^+$ are the two most popular OTS schemes which have already been adopted by state of the art PQ ledgers, "Tangle" and "QRL". SDS-OTS offers 87% reduction in key and signatures sizes as compared to WOTS, and more than 80% reductions in key and signature sizes as compared to WOTS$^+$. WOTS$^+$ is a compact variant of WOTS which uses bitmasks and randomizations to achieve compactness. Bitmasking allows WOTS$^+$ to replace a collision-resistant (CR) hash function by a simple pre-image resistant function which finally offers compactness. However, bit-masking is expensive to achieve on quantum processors as compared to collision resistance [1]. Therefore, SDS-OTS avoids the use of bit-masks and randomizations. Finally, this paper also proposes a model for incorporating SDS into a distributed ledger, with the help of High-Level Petri-nets (HLPN). We can summarize our research contributions as:

1. We propose an efficient HBS scheme "SDS", which is:
   (a) 74% faster than XMSS-WOTS$^+$ (i.e. XMSS incorporated with OTS scheme, WOTS$^+$), and,
   (b) 30% faster than XMSS-WOTS
2. We incorporate an efficient novel OTS scheme into SDS, namely, *SDS-OTS*, which is:
   (a) Most compact OTS scheme
   (b) Offers 87% reduction in key and signatures sizes as compared to the WOTS

(c) Offers 85% reduction in key size as compared to the WOTS$^+$
(d) Offers 83% reduction in signature size as compared to the WOTS$^+$
(e) Based on CR hash functions, hence, avoids the use of bitmasks (which are expensive to achieve on quantum processors)

3. We provide HLPNs to present a road map for the researchers and end-users for incorporating SDS into a distributed ledger.

The rest of the paper is organized as Section-2 gives a preliminary knowledge of HBS schemes. Section-3 discusses our proposed HBS scheme *SDS* in detail. Section-4 provides details for incorporating *SDS* into a distributed ledger. Sections 5 provides security and performance evaluation. Finally, Section-6 concludes this paper.

2. Hash Based Digital Signature (HBS) Schemes

An HBS scheme is either a one-time (OTS), a few-time (FTS), or a many-time (MTS) signature scheme [1]. MTS scheme uses one or more OTS/FTS scheme as its building block. Lamport proposed the very first OTS scheme which suffered from impractically larger key and signature sizes [14]. Being an OTS scheme, the Lamport scheme does not allow for signing two or more messages using the same key. So, each time the user has to generate a fresh key pair to sign a new message. Merkle signature scheme (MSS) [13] reduced key and signature sizes to a practical level and incorporated hash trees to allow for reusing a single public key for a specific number of times (many times). MSS uses Winternitz-OTS (WOTS) scheme as its building block. XMSS [12] further reduced key and signature sizes and incorporated a more secure version of the hash tree. XMSS uses a compact variant of WOTS (i.e. WOTS^{PRF}) as its building block. More advance MTS schemes, like XMSS^{MT} [15], XMSS-T [16], SHPINCS [17], and its variants [18], [19] allow reusing a single public key for an unlimited number of times. Fig. 1 classifies existing HBS schemes into OTS, FTS, or MTS schemes, whereas, Fig. 2 provides a mapping between MTS and OTS/FTS schemes.

2.1. OTS/FTS Schemes

Lamport scheme [14] signs each bit of the message-hash individually, therefore, the total number of signature values are equal to the total number of bits in the message-hash. If message-hash is M-bit long and each of the individual signature values is n-bit long then, the bit-length of the signature can be computed using the formula given in Eq. (1). The key size is even double of the signature size because each bit of the message-hash has two

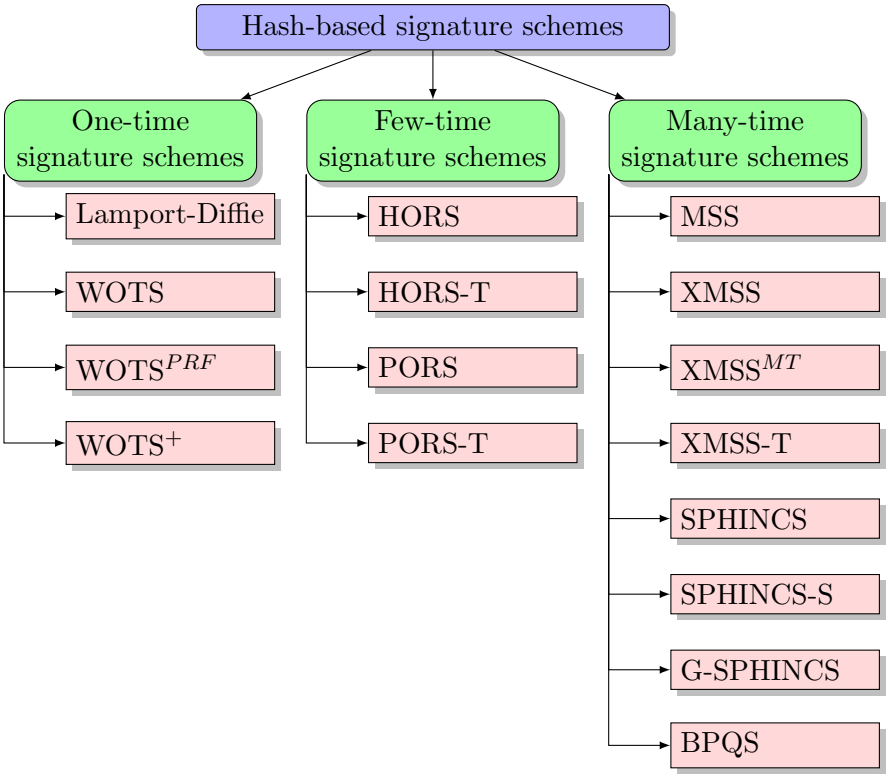


Figure 1: Classification of HBS schemes

key-values associated to it. Eq. (2) provides the formula for computing the key-length (in bits). In the Lamport scheme, a public-key (pk) value is simply the first post-image of the corresponding private key (sk) value (we denote it as $|h| = 1$). Throughout the paper, we will use $|key|$ to denote the total number of values in key and $|\sigma|$ to denote the total number of values in the signature. Table 1 explains all the symbols used in the paper.

$$\sigma_{(Lamport)} = (M)(n) \tag{1}$$

$$key_{(Lamport)} = 2(M)(n) \tag{2}$$

WOTS scheme [13] divides message-hash into groups or patches of bits and signs a complete patch of bits simultaneously. Thus the total number of signature values decrease significantly. For a 4-bit patch-size, there will be a total $\frac{M}{4}$ signature values (i.e. just one-fourth of the Lamport scheme). The patch-size (let we denote it as p) is customizable, i.e. user can select

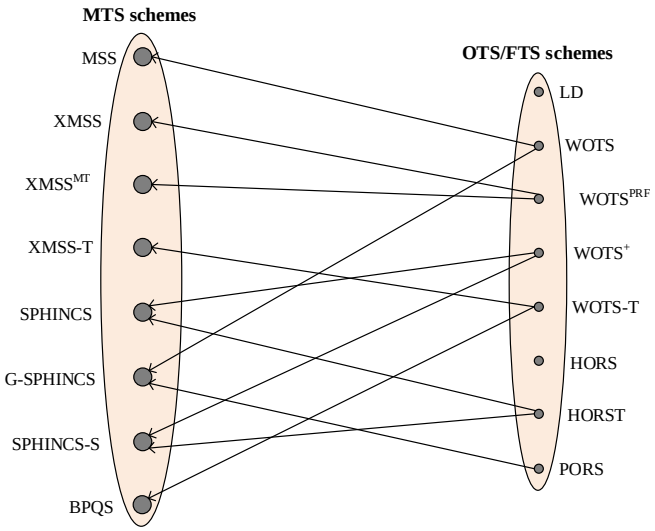


Figure 2: OTS/FTS schemes mapping to the MTS schemes

the patch-size, he wants. The patch size is inversely proportional to the key and signature sizes, however, it is directly proportional to the computational cost. Therefore, a balance must be established. A typical patch-size is 4-bit. In WOTS, there are total $\frac{M}{p}$ patches in the message-hash, so we can write the message-hash (H) like given in Eq. (3). An individual patch of bits can produce a value in the range *zero* to $2^p - 1$. The actual number of signature values is slightly greater than $\frac{M}{p}$ because of a *checksum* that is appended to the message-hash. Eq. (4) provides the formula used to compute the checksum (c). Finally, key and signature sizes of WOTS can be computed using the formula given in Eq. (5). For WOTS we have, $|h| = 2^p$ which means that a pk-value is $(2^p)^{th}$ post-image of the corresponding sk-value, and $|\sigma| = |key| = \frac{M}{p} \mid\mid c$ (the number of key and signature values are equal in WOTS).

$$H = m_1||m_2||m_3||.....||m_{\frac{M}{p}} \tag{3}$$

$$c = \sum_{i=1}^{\frac{M}{p}} (2^p - 1) - m_i \tag{4}$$

$$\sigma_{(WOTS)} = \left(\frac{M}{p} \mid\mid c\right) n \tag{5}$$

WOTS^{PRF} [12] and WOTS^{+} [11] are the two compact variants of WOTS, which achieve compactness by replacing the CR hash function by a simple *one-way* function. The CR property being vulnerable to the birthday paradox attack is relatively harder to achieve [20]. The PQ security level of a *512*-bit long CR function is just equal to that of a *341*-bit long oneway function [6]. Although, both WOTS^{PRF} and WOTS^{+} offer compactness, however, they are computationally expensive. Both these variants incorporate bitmasking, which is relatively expensive for quantum processors [1]. The signature sizes of both these variants are the same, however, the key size of WOTS^{+} is relatively larger because of an additional set of randomization elements. Eq. (6) provides the formula for computing key-size for WOTS^{+}.

HORS scheme [17] is the most efficient scheme w.r.t the signature creation process, however, it suffers from an impractically large key size. Eq. (7) provides the formula for computing the key size in HORS. The patch size (p) in HORS must be sufficiently larger because the scheme will not be secure otherwise. A large patch size in HORS causes an extremely large key size. A *pk*-value is simply the first post-image of the corresponding *sk*-value. PORS [19] is a secure variant of HORS which offers an enhanced security at a marginal computational overhead. The key and signature sizes of both HORS and PORS are the same. The only difference is that in the case of HORS, multiple bit-patches may correspond to the same signature value, however, in PORS there is always a distinct signature value against each of the bit-patches.

$$key_{(WOTS^{+})} = \left(\frac{M}{p} \mid\mid c\right) n + (2^{p})n \tag{6}$$

$$key_{(HORS)} = (2^{p})n \tag{7}$$

The proposed OTS scheme, i.e. *SDS OTS*, uses a 4-bit patch size. Eq. (8) provides the formula for computing key and signature sizes of SDS OTS. Because p is fairly smaller (just 4-bit), therefore, both key and signature lengths are significantly smaller. Both key and signature consist of *17* values in total. Each of the first sixteen *pk*-values is a 96^{th} post-image of the corresponding *sk*-value. However, the last *pk*-value (which is used for

checksum) is the 1536^{th} post-image of the corresponding *sk*-value.

$$\sigma_{(SDS)} \quad = \quad (2^p + 1)n \tag{8}$$

Table 1: Symbols and their description

Symbol	Description
n	Security parameter (bit-length of an individual key/signature item)
H	Hash of the message to be signed
M	Bit-length of the hash of the message to be signed
p	Size/length of an individual patch of bits
m	An individual patch of the message hash
c	Checksum appended to the message hash by WOTS
$\|\sigma\|$	Total no. of items/values in the signatures
$\|key\|$	Total no. of items/values in the private/public key
$\|h_x\|$	Total no. hash iterations required to transform a normal (except checksum) sk-item to the pk-item
$\|h_c\|$	Total no. hash iterations required to transform the checksum sk-item to the pk-item
$\|h\|$	Average no. hash iterations required to transform an sk-item to the pk-item
b	The security level offered by an OTS/FTS scheme
f_H	Oneway hash function
$\mathcal{F}_{SDS-OTS}$	Forger trying to breaks SDS-OTS
$\mathcal{A}_{onewayness}$	Adversary trying to breaks onewayness of f_H
$m^{\mathcal{F}}, \sigma^{\mathcal{F}}$	Message/signature pair returned by forger
$H^{\mathcal{Q}}$	Message queried by the forger for signatures
$f_H^x(y)$	f_H computed on y for x-times

3. Smart Digital Signatures (SDS)

SDS incorporates two customization in XMSS; *firstly*, it replaces the underlying OTS scheme of SDS by a novel OTS scheme "*SDS OTS*", *secondly*, SDS proposes a modified version of XMSS-Ltree, namely *SDS Ltree*. *SDS-OTS* offers a magical reduction in both key and signature sizes. The next two subsections respectively explain our proposed OTS scheme and our modified L-tree construction. After this discussion, we will provide the algorithm for creating the main SDS tree (algorithm 5). We use the term "main hash tree" to refer the tree which finally encapsulates many compressed OTS public keys into a single SDS public key (*SDSpk*).

3.1. SDS OTS Scheme

In this section, we will explain our newly proposed OTS scheme "SDS-OTS". The proposed scheme works as follows:

3.1.1. Key generation

The private and the public keys both consist of 17 values. We create all of the private-key-values from a single initial value known as the *seed*. It is safe to generate the whole private key from a single seed because our scheme never uncovers any of the private key values to the verifier. Each of the public-key-values is an n^{th} post-image of the corresponding private-key-value. The first 16 public-key-values are the 96^{th} post-images of their corresponding private-key-values, whereas, the last (i.e. 17^{th}) public-key-value is the 1536^{th} post-image of the corresponding private-key-value. The first 16 private-key-values are used to sign the message, whereas, the last value is used to sign the checksum. The length of the hash function used to generate the post-images has a direct impact on the level of security, key/signature sizes, and processing time. For an appropriate level of security, we recommend using the hash function SHA384. SDS OTS offers 0.82KB key and signature sizes with hash function SHA384. Algorithm 1 provides complete pseudo-code for the key generation process.

3.1.2. Signature creation

The signature creation process starts with computing hash of the message (msg) to be signed. We recommend using the hash function "SHA384" thus, the message-hash consists of 96 hexadecimal symbols in total. We index those symbols as $\{1 \rightarrow 96\}$. Each index refers to a hexadecimal symbol. Then we classify the indexes into sixteen different strings, like this: the string (str_0) of the indexes which contain the symbol 0, the string (str_1) of the indexes which contain the symbol 1, and *up to* the string (str_f) of the indexes which contain the symbol f. Next, we sum-up the digits of each of the strings to generate sixteen different *string-sums*, we denote these sums as $\sum_{i=0}^{15} str_iSum$. We use the modulus operator to ensure that all of the string-sums are in the range $\{1 \rightarrow 96\}$. We compute the first 16 signature-values by computing post-images of the corresponding private-key-values for the corresponding str_iSum number of times. The 17^{th} signature-value is for checksum for which we first compute the *checksum* following the rule given in Eq. (9). Finally, we compute the 17^{th} signature-value by computing the post-image of the corresponding private-key-value for *checksum* number of times. Fig. 3 explains the process of signature creation for an example message "Good Message 707070". Algorithm 2 provides complete pseudo-code for signature creation process.

$$\textbf{Checksum} = \sum_{i=0}^{15}(96 - str_iSum) \tag{9}$$

Algorithm 1 Key Generation

Input: *Security Parameter* (1^{384})
Output: $\sum_{i=0}^{16}(sk[i],\ pk[i])$

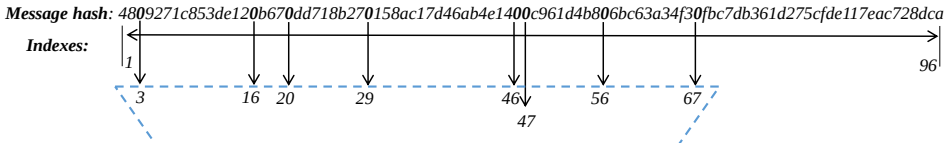
```
seed ← os.urandom(48)          ▷ Generates a 384-bit random "seed" for private key
sk ← [ ]
for a = 0 → 16 do              ▷ All 17 sk-elements are computed from a single "seed"
    seed ← sha384(seed)
    sk.append(seed)
end for
pk ← [ ]
for a = 0 → 15 do              ▷ Transforms first "16" sk-elements into pk-elements
    x ← sk[a]
    for b = 1 → 96 do
        x ← sha384(x)
    end for
    pk[a] ← x
end for
x ← sk[16]                     ▷ Transforms the 17^th sk-element into pk-element
for b = 1 → 1536 do
    x ← sha384(x)
end for
pk[16] ← x
```

3.1.3. Signature verification

The verifier starts the verification process by computing the hash of the message (which signatures are to be verified). Then the verifier computes "*string-sums*" and the "*checksum*" by following the procedure described in the "signature creation" phase. The *string-sums* and the *checksum* allow the verifier to transform the signatures into a verification key (*vfKey*). The verification key consists of 17 values, each of which is some of the post-image of the corresponding signature-value. Algorithm 3 explains the procedure of computing the verification key from the signatures. The verification-key must be equal to the public-key otherwise signature verification will be failed.

3.2. SDS L-tree

XMSS uses L-tree to compress the OTS public keys to allow them to exist as leaf nodes of the main hash tree. Our proposed variant of L-tree is more compact and efficient than XMSS L-tree. SDS L-tree is compact because unlike XMSS L-tree, SDS L-tree does not involve additional randomization elements while computing a parent node from the corresponding child nodes. SDS L-tree uses the last (i.e. 17^{th}) element of the OTS public key for randomization purposes. The plain 17^{th} pk-element substitutes the left randomization element, whereas, the hash of the 17^{th} pk-element substitutes the right randomization element. Because SDS L-tree engages the 17^{th}



Symbol (m)	String of Indexes with an "m" symbol (Str_m)	$Sum(Str_m)\%96+1$ ($Str_m Sum$)	Signatures (σ_m)
0	**316202946475667**	69	$hash^{69}(sk_m)$
1	7142430354451768586	84	$hash^{84}(sk_m)$
2	515277892	47	$hash^{47}(sk_m)$
3	1161636674	42	$hash^{42}(sk_m)$
4	13842455364	46	$hash^{46}(sk_m)$
5	103180	14	$hash^{14}(sk_m)$
6	183950576075	57	$hash^{57}(sk_m)$
7	61923283671798791	90	$hash^{90}(sk_m)$
8	2925325593	46	$hash^{46}(sk_m)$
9	449	18	$hash^{18}(sk_m)$
a	3340628996	51	$hash^{51}(sk_m)$
b	17264154586973	69	$hash^{69}(sk_m)$
c	834485970819095	81	$hash^{81}(sk_m)$
d	122122375272778394	75	$hash^{75}(sk_m)$
e	13438488	40	$hash^{40}(sk_m)$
f	656882	36	$hash^{36}(sk_m)$
16		$\Sigma(96 - Str_m Sum) = 671$	$hash^{671}(sk_m)$

Figure 3: SDS OTS: Signature Creation Example for the Message "Good Message 707070"

Algorithm 2 Signature Creation

Input: $Message(Msg)$, $\sum_{i=0}^{16} sk[i]$
Output: $\sum_{i=0}^{16} \sigma[i]$

$H[\,] \leftarrow sha384(Msg)$
$hex \leftarrow$ "0123456789$abcdef$" ▷ The set of hex alphabets stored as string "hex"
$hashCounts \leftarrow [\,]$
for x *in* hex **do** ▷ A loop iterating for each of the hexadecimal alphabet
    $str \leftarrow$ " "
    for $i = 1 \rightarrow 96$ **do** ▷ A loop to parse each hexadecimal character in "H"
        if $H[i] == x$ **then**
            $str \leftarrow str + i$ ▷ Appends index of the selected alphabet of H to str
        end if
    end for
    $strSum \leftarrow 0$
    for d *in* str **do** ▷ A loop to compute sum of the digits in string "str"
        $strSum \leftarrow strSum + int(d)$
    end for
    $strSum \leftarrow strSum\%96 + 1$ ▷ To impose $1 \leq strSum \leq 96$
    $hashCounts.append(strSum)$
end for
$checksum \leftarrow 0$
for $j = 0 \rightarrow 15$ **do** ▷ A loop to compute the checksum
    $checksum \leftarrow checksum + (96 - hashCounts[j])$
end for
$hashCounts.append(checksum)$
$\sigma \leftarrow [\,]$
for $r = 0 \rightarrow 16$ **do** ▷ A loop to compute the signatures on "M"
    $k \leftarrow sk[r]$
    for $q = 1 \rightarrow hashCounts[r]$ **do**
        $k \leftarrow sha384(k)$
    end for
    $\sigma.append(k)$
end for

pk-element (which is basically for checksum purposes) into the randomization process, therefore, it is no more required to store and process the 17^{th} pk-element as a leaf node. Removal of the 17^{th} pk-element from leaf nodes generates a balanced binary hash tree. On the other side, XMSS L-tree is an unbalanced binary hash tree because of the checksum appended to the message-hash by WOTS^{PRF}. SDS L-tree being a balanced binary tree, does not involve the processing of the additional unbalanced leaf nodes therefore it is more efficient than XMSS L-tree.

3.2.1. Key compression using SDS L-tree

The proposed key compression algorithm is the same as *XMSS L-tree* with just one difference, i.e. we use the last key element pk_{16} for randomization purposes rather than introducing new randomization elements. Fig. 4 shows the complete structure of an SDS L-tree and explains the process of

Algorithm 3 Signature Verification

Input: $Message(Msg)$, $\sum_{i=0}^{16} \sigma[i]$, $\sum_{i=0}^{16} pk[i]$
Output: $Succeeded/Failed$

Follow steps 1 to 22 of **algorithm 2** to compute "hashCounts"
$vfKey \leftarrow [\,]$ ▷ Verifier computes "vfKey" from the signatures
for $i = 0 \rightarrow 15$ **do** ▷ To compute first 16-elements of the vfKey
    $sig = \sigma[i]$
    for $j = 1 \rightarrow 96 - hashCounts[i]$ **do**
        $sig \leftarrow sha384(sig)$
    end for
    $vfKey.append(sig)$
end for
$sig \leftarrow \sigma[16]$ ▷ To compute the 17^{th} element of the vfKey
for $j = 1 \rightarrow 1536 - hashCounts[16]$ **do**
    $sig \leftarrow sha384(sig)$
end for
$vfKey.append(sig)$
if $vfKey == pk$ **then**
    $output : Succeeded$
else
    $output : Failed$
end if

producing a parent node from the corresponding child nodes. Algorithm 4 provides the complete pseudo-code of the key compression process. We use a binary tree of height "5" (max. level is 4), in which leaf nodes store the public-key-values. There are a total of 17 public-key-values of which the first sixteen values are stored as leaf nodes of the L-tree, whereas, the 17^{th} value is used for randomization purposes while constructing a parent node from the corresponding child nodes. The formula to construct a parent node from the corresponding child nodes is given in Eq. (10). The same procedure is iterated to devise all of the upper-level nodes of the tree up to the *root node* which represents the compressed form of the public key.

$$\begin{aligned} N_{(i,j)} = & hash\big[\, [N_{(2i,j-1)} \;\oplus\; pk_{16}\,] + \\ & [N_{(2i+1,j-1)} \;\oplus\; hash(pk_{16})]\,\big] \end{aligned} \tag{10}$$

4. SDS OTS based Distributed Ledger

This section provides guidelines for incorporating SDS OTS into a distributed ledger (DL). We have formally modeled our discussion with the help of HLPN. Fig. 5 provides HLPN for the proposed DL. Table 2 explains the places of the corresponding HLPN. We use the Z-specification language to

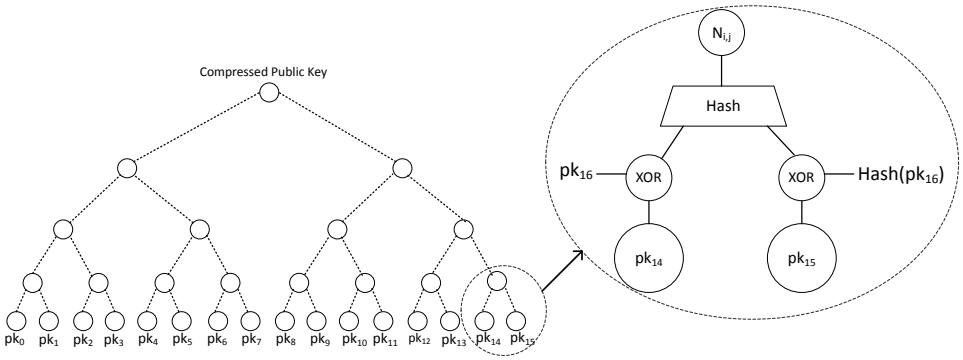


Figure 4: Public Key Compression using SDS L-tree

Algorithm 4 Key compression using SDS L-tree

Input: $\{pk_0, pk_1, pk_2 \cdots pk_{16}\}$
Output: *key*

define array $N[1 \cdots 31]$ ▷ "N" is a binary tree (SDS L-tree)
for $i = 0 \rightarrow 15$ **do**
    $N.insert(i + 16, pk[i])$ ▷ pk_0 to pk_{15} form leaf nodes of "N"
end for
$leftRand \leftarrow pk_{16}$ ▷ pk_{16} forms the randomization elements
$rightRand \leftarrow hash(pk_{16})$
$k \leftarrow 16$
for $j = 0 \rightarrow 3$ **do** ▷ Generates levels 1 to 4 of the L-tree
    for $(i = k;\ i < 2k;\ i = i + 2)$ **do**
        $r1 \leftarrow N[i] \oplus leftRand$
        $r2 \leftarrow N[i + 1] \oplus rightRand$
        $N[\frac{i}{2}] \leftarrow hash(r1\ +\ r2)$
    end for
    $k \leftarrow \frac{k}{2}$
end for
$key \leftarrow N[1]$ ▷ The root node produces the compressed *pk*

define rules for the transitions included in the HLPN. For large-sized rules involving multiple steps, we provide complete algorithms. The three major parts of the HLPN include, ledger address (LA) generation, coins receiving, and coins spending.

The process of LA generation starts from a private key which is a set of 17 cryptographic values all generated from a single seed (Place-1, Eq. (11)). The private key is used to generate the public key. Each of the first 16 private-key-values is hashed for 96 times to generate the corresponding public-key-value, however, the 17^{th} private-key-value is hashed for 1536 times to generate the corresponding public-key-value (Place-2; Eq. (12)). Finally, the compressed form of the public key represents LA of the user

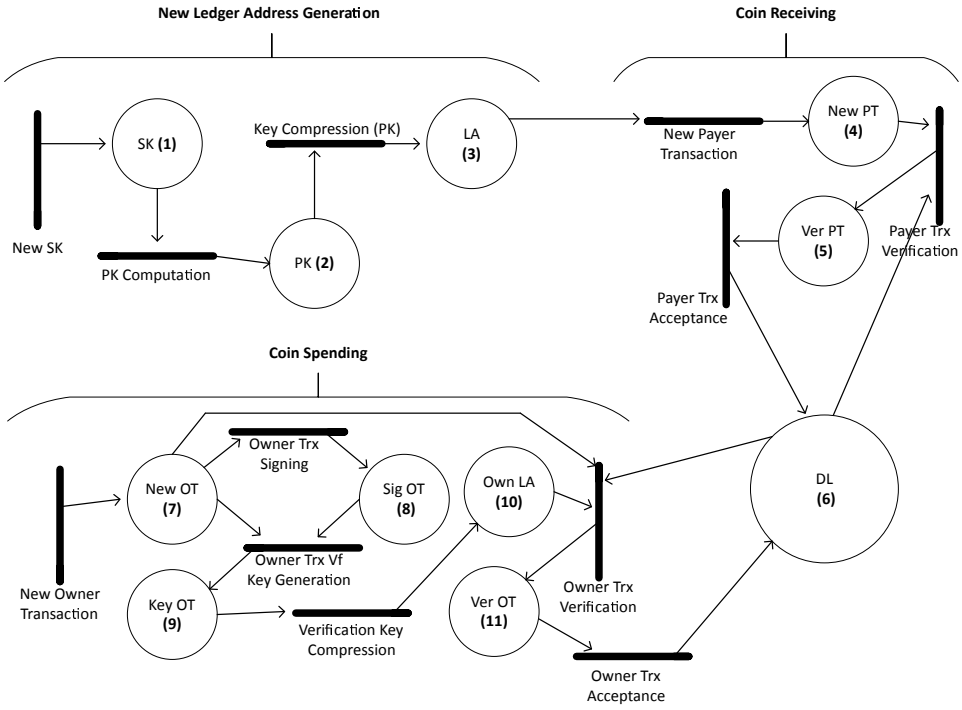


Figure 5: High Level Petri Net (HLPN): Incorporating SDS OTS into a DL

Algorithm 5 SDS main hash tree creation

Input: *tree height (h)*
Output: *public key (SDSpk), Randomization Elements (leftRE, rightRE)*

$maxLevel \leftarrow (h-1)$
$totalNodes \leftarrow 0$
for $(i \leftarrow maxLevel;\ i \geq 0;\ i \leftarrow i-1)$ **do**
    $totalNodes \leftarrow totalNodes + 2^i$ ▷ Computing total no. of nodes in SDS tree
end for
define array $T[1 \cdots totalNodes]$
for $j = 2^{maxLevel} \rightarrow totalNodes$ **do** ▷ Generating leaf nodes of SDS tree
    Generate a new SDS OTS key-pair
    Compress the OTS public key
    $T[j] \leftarrow compressedOTSpk$
end for
$leftRE \leftarrow os.urandom(48)$ ▷ To generate a 384-bit random value
$rightRE \leftarrow os.urandom(48)$
$k \leftarrow 2^{maxLevel}$
for $j = 1 \rightarrow maxLevel$ **do** ▷ Generating upper levels of SDS tree
    for $(i = k;\ i < 2k;\ i = i+2)$ **do**
        $r1 \leftarrow T[i] \oplus leftRE$
        $r2 \leftarrow T[i+1] \oplus rightRE$
        $T[\frac{i}{2}] \leftarrow hash(r1\ +\ r2)$
    end for
    $k \leftarrow \frac{k}{2}$
end for
$SDSpk \leftarrow T[1]$ ▷ The root node is the pk

(Place-3; Eq. (13)). The DL uses an LA to preserve assets of a user.

$$\textbf{R(new SK)} = SK' = \sum_{i=0}^{16} \{sk_i = seed \wedge seed' = hash(seed)\} \tag{11}$$

$$\textbf{R(PK Computation)} = \left[\sum_{i=0}^{15(f)} \{pk'_i = hash^{96}(sk_i)\}\right] \wedge \left[pk'_{16} = hash^{1536}(sk_{16})\right] \tag{12}$$

Table 2: HLPN Places Description

No.	Place	Description	Data-type
1	SK	Private key	A set of 17 values each 48 byte long
2	PK	Public key	A set of 17 values each is a hash output
3	LA	Ledger address	A single hash output
4	NewPT	New payer transaction	$\mathbb{P}(ID, TrxTime, Inputs, Outputs, \sigma)$
5	VerPT	Verified payer transaction	$\mathbb{P}(ID, TrxTime, Inputs, Outputs, \sigma)$
6	DL	Distributed ledger	$\mathbb{P}(Transactions)$
7	NewOT	New owner transaction	$\mathbb{P}(ID, TrxTime, Inputs, Outputs, \sigma)$
8	SigOT	Signatures on the owner's transaction	A set of 17 values each is a hash output
9	KeyOT	Verification key for the owner transaction	A set of 17 values each is a hash output
10	OwnLA	Owner's ledger address computed by the verifier	A single hash output
11	VerOT	Verified owner transaction	$\mathbb{P}(ID, TrxTime, Inputs, Outputs, \sigma)$

$$\begin{aligned}\mathbf{R[Key\ Compression(PK)]} =& \left[\sum_{i=0}^{15} N'_{(i+16,0)} = pk_i\right] \wedge \\ & \left[\sum_{\substack{j=0 \\ k=16}}^{j\leq 3} \left\{ \sum_{\substack{k\leq i\leq (2k-2) \\ i\ a\ multiple\ of\ 2}} N'_{(\frac{i}{2},j+1)} = hash(N_{(i,j)} \oplus pk_{16}|| \right.\right. \\ & \left.\left. N_{(i+1,j)} \oplus hash(pk_{16}))\right\} \wedge \left\{k' = \frac{k}{2}\right\}\right] \wedge \left[LA' = N_{(\frac{i}{2},j+1)}\right]\end{aligned} \tag{13}$$

To allocate coins to a new LA, the payer originates a new transaction, we denote it as payer transaction (new-PT) [Place-4; Eq. (14)]. The verifiers verify the new-PT and upon a successful verification, we change its status from new-PT to verified PT (ver-PT) [Place-5; Eq. (15)]. Finally, a miner adds the ver-PT to the DL [Eq. (16)]. The verification process is the same for both, payer and the owner transactions.

$$\begin{aligned}\mathbf{R(New\ Payer\ Transaction)} =& NewPT' = \{ID, Time, \\ & Inputs, Outputs\}\end{aligned} \tag{14}$$

$$\begin{aligned}&\mathbf{R(Verified\ Payer\ Transaction)} = \\ &VerPT' = NewPT \cup \{\sigma_{newPT}\}\end{aligned} \tag{15}$$

$$\mathbf{R(Payer\ Transaction\ Acceptance)} = DL' = DL \cup VerPT \tag{16}$$

To spend coins, the owner generates a new transaction, we denote it as a new owner transaction (new-OT) [Place-7; Eq. (17)]. Next, the owner adds

signatures to his transaction, we denote it as sig-OT (Place-8); Algorithm 6 provides rules for signing the transaction. During verification of the owner transaction, the verifiers use *new-OT* and *sig-OT* to generate the verification key, i.e. key-OT (Place-9; Algorithm 7). Next, verifiers compress the verification key to generate the owner's LA, denoted as own-LA (Place-10; Eq. (18)). The own-LA must already be stored in the ledger with enough coins allocated to it. A valid own-LA turns the owner transaction to a verified owner transaction ver-OT (Place-11; Eq. (19)). Finally, miner accepts the ver-OT by posting it into the ledger (Eq. (20)).

$$\begin{aligned}\textbf{R(New Owner Transaction)} = & NewOT' = \{ID, Time, \\ & Inputs, Outputs\}\end{aligned} \tag{17}$$

Algorithm 6 Owner Transaction Signing

Input: $NewOT$, $\{sk_0, sk_1, sk_2 \cdots sk_{16}\}$
Output: $SigOT$

1: $\sum_{i=0}^{15}$ define string str_i
2: $\forall x \in hash(newOT) \bullet str_x.concat(index(x))$
3: $\sum_{i=0}^{15}$ define $str_iSum = 0$
4: $\sum_{i=0}^{15(f)} \left[\sum_{j=0}^{str_i.length} str_iSum' = str_iSum + int(str_i[j])\right] \wedge$
$\left[str_iSum = str_iSum\%96 + 1\right]$
5: $\sum_{i=0}^{15} \sigma_i = hash^{str_iSum}(sk_i)$
6: define $checksum = 0$
7: $\sum_{i=0}^{15} checksum = checksum + (96 - str_iSum)$
8: $\sigma_{16} = hash^{checksum}(sk_{16})$
9: $SigOT' = \{\sigma_0, \sigma_1, \sigma_2 \cdots \sigma_{16}\}$

Algorithm 7 Owner Transaction Verification Key Generation

Input: $NewOT$, $SigOT$
Output: $KeyOT$

1: Follow steps $1 \rightarrow 4$ of algorithm 6 to generate str_iSum for $0 \leq i \leq 15$
2: define $vf\_key[17]$
3: $\sum_{i=0}^{15} vf\_key_i = hash^{96 - str_iSum}(\sigma_i)$
4: Follow steps $6 \rightarrow 7$ of algorithm 6 to generate $checksum$
5: $vf\_key_{16} = hash^{1536 - checksum}(\sigma_{16})$
6: $KeyOT' = \{vf\_key_0, vf\_key_1, vf\_key_2 \cdots vf\_key_{16}\}$

$$\begin{aligned}&\textbf{R(Verification Key Compression)} = \\ &OwnLA' = \textbf{R[Key Compression (KeyOT)]}\end{aligned} \tag{18}$$

$$\textbf{R(Owner Trx Verification)} = \\ OwnLA \in DL \implies VerOT' = \{NewOT \cup \sigma_{newOT}\} \vee \\ OwnLA \notin DL \implies VerOT = \phi \tag{19}$$

$$\textbf{R(Owner Trx Acceptance)} = \; DL' = DL \cup VerOT \tag{20}$$

5. Evaluations of SDS

This section compares the compactness and computational efficiency of SDS with other HBS schemes. Furthermore, we provide a formal security proof of SDS.

5.1. Security Evaluation of SDS

SDS requires a secure hash function for its secure execution. A secure hash function is the one which can resist three types of attacks, pre-image resistant, second pre-image resistant, and collision resistant. In case of pre-image attack, the challenge for the adversary (A) is to find such an input (x) which corresponding output (y) is known to him [Eq. 21]. In case of second pre-image attack, the adversary knows an input-output pair (x, y), where the challenge for him is to find another input (x') which must be different from x, however its output should be the same (i.e. y) [Eq. 22]. Finally, in collision attack, the adversary has to find any two different inputs (x, x') which must map to a same output [Eq. (23)].

$$Pr[y = h(x); x' \leftarrow A(y) : x = x'] \leq \epsilon \tag{21}$$

$$Pr[y = h(x); x' \leftarrow A(x, y) : x' \neq x \wedge y = h(x')] \leq \epsilon \tag{22}$$

$$Pr[x, x' \leftarrow A : x \neq x' \wedge h(x) = h(x')] \leq \epsilon \tag{23}$$

The resistance power of a cryptographic protocol against different types of attacks is generally known as the security-level (we denote it as b) offered by that protocol. The security level of a family of hash functions depends on its output-length. An 'n-bit' long hash function offers '(n/2)-bit' PQ security against both of the pre-image and second pre-image resistant. However, an n-bit long hash function offers '(n/3)-bit' PQ security against collision attacks [1]. Table 3 provides classical and quantum security levels of popular hash functions.

Table 3: Hash functions security levels [6], [20], [1]

Hash function	Classical security		Quantum security	
	Pre-image	**Collision**	**Pre-image**	**Collision**
SHA160	160-bit	80-bit	80-bit	53-bit
SHA256	256-bit	128-bit	128-bit	85-bit
SHA384	384-bit	192-bit	192-bit	128-bit
SHA512	512-bit	256-bit	256-bit	171-bit

5.1.1. Formal Security Proof of SDS OTS

We formally prove that SDS OTS is a "chosen plaintext attack" (CPA)-secure scheme; and that, security of SDS OTS is a security reduction of the used hash function. A number of existing studies have used the CPA model to prove security of the proposed schemes. Some latest studies which use CPA model include, ABE with enhanced leakage resistance [21, 22], self-adaptive big data storage [23], anti-quantum blockchain [24], PQ-blockchain [25], *and* compact PQ-blockchain [26]. Other less common approaches used to establish security proofs include simulations [27] and experimental results [28].

CPA Model: CPA model allows a forger $\mathcal{F}$ to query signatures on his chosen message $\left(m^{\mathcal{Q}1}, m^{\mathcal{Q}1}, \cdots, m^{\mathcal{Q}n}\right)$. A signing oracle $\mathcal{O}$ responds forger's queries. In the end, forger has to return a message/signature pair $\left(m^{\mathcal{F}}, \sigma^{\mathcal{F}}\right)$, such that $\sigma^{\mathcal{F}}$ are valid signatures of $m^{\mathcal{F}}$ and $m^{\mathcal{F}} \notin \{m^{\mathcal{Q}1}, m^{\mathcal{Q}2}, \cdots, m^{\mathcal{Q}n}\}$. A CPA secure scheme means that the success probability of the forger is negligible.

Existential unforgeability of SDS-OTS: SDS-OTS is a triple (*Keygen*, *Sign*, *Verify*); *Keygen* takes a security parameter n as input and returns a key-pair (sk, pk). *Sign* takes a message-hash (H) and private key (sk) as input and returns signatures (σ) of H. *Verify* takes message-hash (H), signatures (σ), and public key (pk) as input and returns either *Succeeded* or *Failed*.

KeyGen generates a new key pair (sk, pk). A signing oracle $\mathcal{O}$ having knowledge of the private key (sk) responds the forger's queries. Forger ($\mathcal{F}$) can submit at most one query to $\mathcal{O}$. $\mathcal{F}$ has knowledge of pk. Upon querying of a message $H^{\mathcal{Q}}$ from $\mathcal{F}$, $\mathcal{O}$ must return valid signatures of $H^{\mathcal{Q}}$, i.e. $\sigma^{\mathcal{Q}}$. The challenge for $\mathcal{F}$ is to return a message/signature pair ($H^{\mathcal{F}}$, $\sigma^{\mathcal{F}}$) such that, $\sigma^{\mathcal{F}}$ are valid signatures of $H^{\mathcal{F}}$, and, $H^{\mathcal{F}} \neq H^{\mathcal{Q}}$. SDS-OTS is existentially unforge-able under CPA model if the probability that $\mathcal{F}$ wins the above game in a time $\mathcal{T}$, is at most ϵ. We formally write it as, SDS-OTS is a ($\mathcal{T}$,

ϵ, 1)-existentially unforgeable OTS scheme.

Reduction Proof: Algorithm 8 explains that, how an adversary $\mathcal{A}_{onewayness}$ can exploit a forger ($\mathcal{F}_{SDS-OTS}$) to break onewayness of the underlying hash function. $\mathcal{A}_{onewayness}$ generates a new SDS-OTS key pair. Then he chooses two random values (α, β) and tampers the corresponding key-element (steps 2 - 3). Then adversary runs forger. When forger queries a message ($H^{\mathcal{Q}}$) then, either adversary responds with the corresponding signatures ($\sigma^{\mathcal{Q}}$) [steps 9 - 11] or aborts the process (steps 6 - 7). When forger returns a message-signature pair ($H^{\mathcal{F}}$, $\sigma^{\mathcal{F}}$) then adversary will be able to win the game, only under certain conditions (steps 13 - 17).

The success probability of $\mathcal{A}$ is an aggregation of the three success probabilities, the probability that $\mathcal{A}$ is able to respond $\mathcal{F}$'s signature-query (steps 6 - 8); the probability that $\mathcal{F}$ is able to win the game (step 13); and the probability that the message returned by $\mathcal{F}$ allows $\mathcal{A}$ to compute the challenged pre-image (steps 14 - 16). Eq. 24 computes the success probability of $\mathcal{A}$. The approximated success probability of $\mathcal{A}$ ($\epsilon_{\mathcal{A}}$) is 0.001 times the success probability of $\mathcal{F}$ ($\epsilon_{\mathcal{F}}$). Hence, a negligible success property of the adversary ($\mathcal{A}$) is not possible without a negligible success probability of the forger ($\mathcal{F}$). The total time taken by $\mathcal{A}$ includes, SDS-OTS key generation time (step1), SDS-OTS signing time (steps 9 - 10), and the forgers time (step 12). Eq. 25 computes the total time taken by $\mathcal{A}$. Our algorithm allows just one query to $\mathcal{F}$ (step 5), which leads to onetime-ness. Finally the message returned by $\mathcal{F}$ must be different from the queried message (step 13), which leads to existential unforgeability. Hence it proved that, SDS-OTS is a ($\mathcal{T}$,ϵ,1)-existentially unforgeable signature scheme.

$$\epsilon_{\mathcal{A}} = \frac{\big(|h_x| - \beta\big)}{\big(|key| - 1\big)\big(|h_x|\big)}\ \epsilon_{\mathcal{F}}\ \frac{\big(\beta - 1\big)}{\big(|key| - 1\big)\big(|h_x|\big)} \tag{24}$$

$$\mathcal{T}_{\mathcal{A}} = \mathcal{T}_{KeyGen} + \mathcal{T}_{Sign} + \mathcal{T}_{\mathcal{F}} \tag{25}$$

5.2. Compactness Evaluation of SDS

The underlying OTS scheme of SDS, i.e. SDS OTS, offers minimum key and signature sizes as compared to all of the existing OTS/FTS schemes. SDS-OTS offers 87% reduction in key and signatures sizes as compared to WOTS (adopted by IoTA [8]), and more than 80% reduction in both key and signature sizes as compared to WOTS^{+} (adopted by QRL [9]). Table 4 allows comparing "key and signature" sizes of SDS OTS with the existing OTS/FTS schemes. The formulas for computing key and signature sizes of different OTS/FTS schemes have already been provided in Section-2 (see

Algorithm 8 $\mathcal{A}_{onewayness}$

Input: SDS-OTS(*Keygen*,*Sign*,*Verify*), hash function (f_H), forger $\mathcal{F}_{SDS-OTS}$, a post-image y
Output: Pre-image x, such that $f_H(x) = y$

Generate a new SDS-OTS key pair $\left(\sum_{i=0}^{|key|-1}(sk_i, pk_i)\right)$
Randomly choose $\alpha \in \{0 \dots (|key| - 2)\}$ and $\beta \in \{1 \cdots |h_x|\}$
Tamper the key-element pk_α like this: $pk_\alpha \leftarrow f_H^{(|h_x|-\beta)}(y)$
Run the forger $\mathcal{F}_{SDS-OTS}$
if $\mathcal{F}_{SDS-OTS}$ queries a message ($H^{\mathcal{Q}}$) **then**
    if $hashCounts_\alpha$ for $H^{\mathcal{Q}} < \beta$ **then**
        return *failed*
    else
        Compute $\sum_{i=0}^{|key|-1}\left(\sigma_i \leftarrow f_H^{hashCount_i}(sk_i)\right)$ for $i \neq \alpha$
        Compute $\sigma_\alpha \leftarrow f_H^{(hashCount_\alpha - \beta)}(y)$
        Respond $\mathcal{F}_{SDS-OTS}$ with $\sigma^{\mathcal{Q}}$
if $\mathcal{F}_{SDS-OTS}$ returns a message-signature pair $(H^{\mathcal{F}}, \sigma^{\mathcal{F}})$ **then**
    if $\sigma^{\mathcal{F}}$ are valid signatures of $H^{\mathcal{F}}$ **and** $H^{\mathcal{F}} \neq H^{\mathcal{Q}}$ **then**
        if $hashCount_\alpha$ for $H^{\mathcal{F}} > \beta$ **then**
            return *failed*
        else
            return $f_H^{(\beta - hashCount_\alpha - 1)}(\sigma_\alpha^{\mathcal{F}})$
In any other case **return** *failed*

Equations (1) - (8)). The key and signature sizes of an OTS/FTS scheme depend on several parameters, like, bit-length of an individual key-element (n) and bit-length of the message hash (M), etc. Table 1 explains all those parameters.

5.3. Computational Efficiency Evaluation of SDS

We have compared SDS with two different instantiations of XMSS. In the first instantiation, we used WOTS$^+$ as the core OTS scheme, whereas, in the second instantiation we used WOTS as the core OTS scheme. We generated hash trees of all heights between *two* and *eleven*. For these implementations, we used Python language in the environment "JetBrains PyCharm Community Edition 2018.3.3". The testbed consists of an Intel Core i5 CPU (2.4 GHz) with 4GB RAM, running Windows 8.1 32-bit release. The results reveal that SDS is *on average* 74% more efficient than XMSS-WOTS$^+$ (i.e. XMSS incorporated with the OTS scheme "WOTS$^+$"), and 30% more efficient than XMSS-WOTS. The graphs in Fig. 6 and 7 allow comparing tree generation time of SDS with both instantiations of XMSS.

5.3.1. Execution Time of SDS OTS

The execution time of *SDS OTS* is comparable to the other OTS/FTS schemes. Figures 8, 9, and 10 allow comparing the execution time of SDS OTS with the other OTS/FTS schemes. The results reveal that all

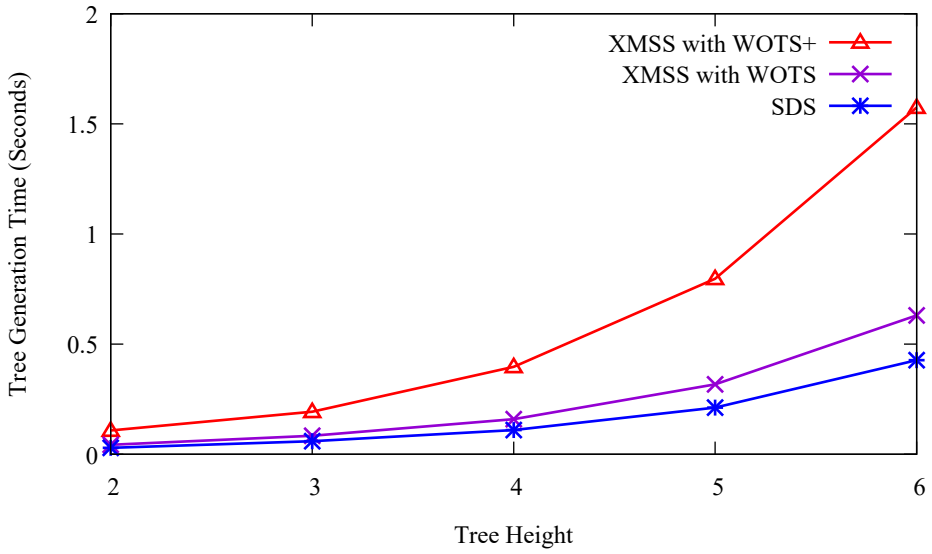


Figure 6: Main Hash Tree Generation Time: SDS vs XMSS (Tree height 2 to 6)

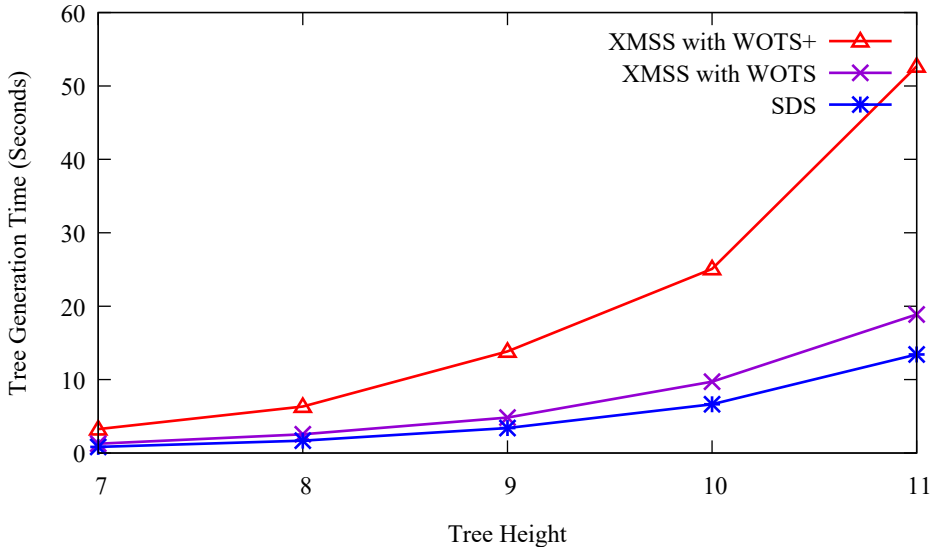


Figure 7: Main Hash Tree Generation Time: SDS vs XMSS (Tree height 7 to 11)

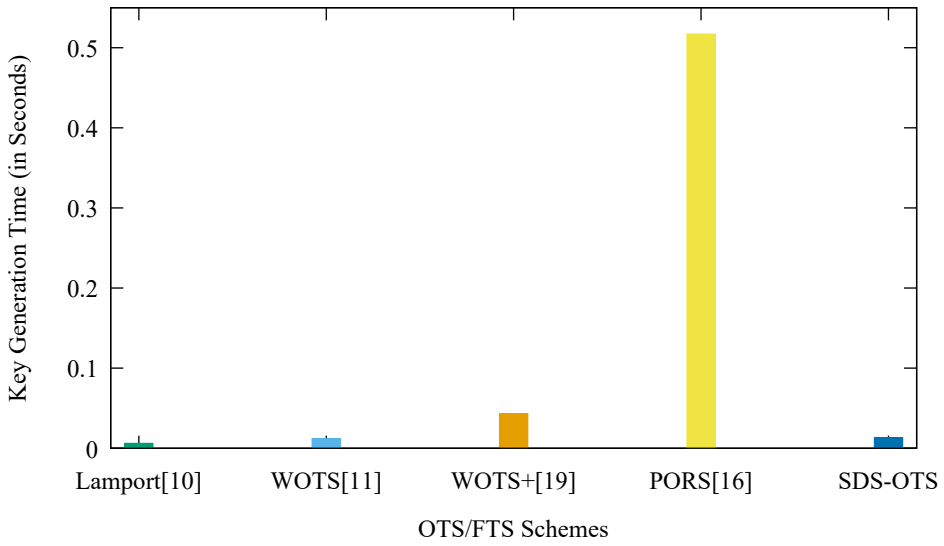


Figure 8: Key Generation Time (in seconds) of OTS/FTS Schemes

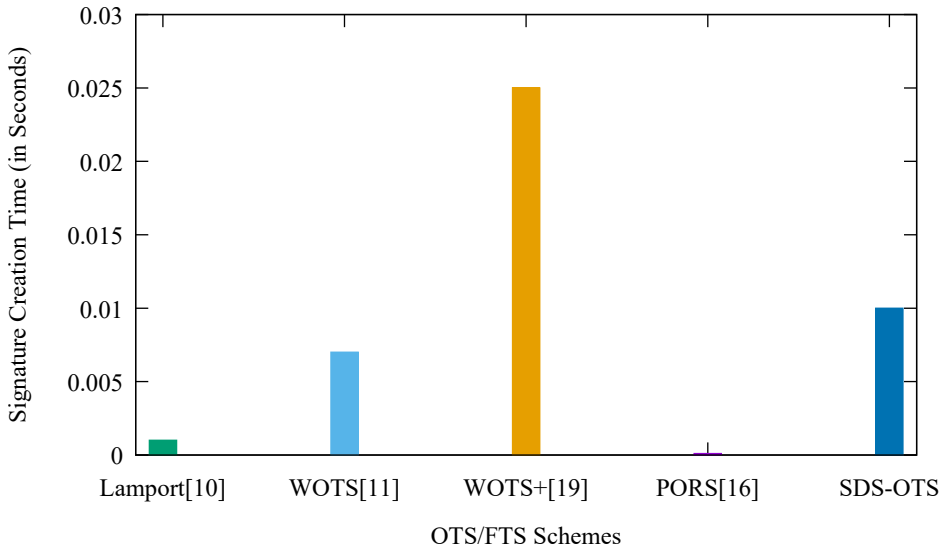


Figure 9: Sig. Creation Time (in seconds) of OTS/FTS Schemes

Table 4: Key and signature sizes: OTS/FTS schemes

Scheme	Parameters				PQ-SL*	Sig. Size	Key Size
	n	**m**	**p**	$\|key\|$			
[14]	256	256	n/a	512	85-bit	8.2KB	16.4KB
[13]	256		4	67	85-bit	2.1KB	2.1KB
[12]	210		4	67	87-bit	1.8KB	1.8KB
[11]	190		4	83	88-bit	1.6KB	2.0KB
[19]	256		16	65536	85-bit	2.1MB	2.1MB
SDS-OTS	256		4	17	79-bit	0.5KB	0.5KB
[14]	384	384	n/a	768	128-bit	18.4KB	36.9KB
[13]	384		4	99	128-bit	4.8KB	4.8KB
[12]	280		4	99	127-bit	3.5KB	3.5KB
[11]	270		4	115	128-bit	3.3KB	3.9KB
[19]	384		16	65536	128-bit	3.1MB	3.1MB
SDS-OTS	384		4	17	122-bit	0.8KB	0.8KB
[14]	512	512	n/a	1024	171-bit	32.8KB	65.5KB
[13]	512		4	131	171-bit	8.4KB	8.4KB
[12]	370		4	131	172-bit	6.1KB	6.1KB
[11]	360		4	147	172-bit	5.9KB	6.6KB
[19]	512		16	65536	171-bit	4.2MB	4.2MB
SDS-OTS	512		4	17	164-bit	1.1KB	1.1KB

*Post Quantum Security Level

three algorithms of SDS OTS, i.e. key generation, signature creation, *and* signature verification, are fairly efficient. SDS OTS offers 70% and 60% reductions in key-generation and signature-creation times respectively as compared to the existing most compact OTS scheme "WOTS $^+$".

6. Conclusions

We have proposed a novel hash-based digital signature scheme "smart digital signatures (SDS)", which is a compact and efficient variant of the popular hash-based scheme XMSS. The comparison results reveal that SDS is 74% more efficient than XMSS-WOTS$^+$ and 30% more efficient than XMSS-WOTS. The underlying OTS scheme of SDS, i.e. SDS-OTS is the most compact OTS scheme as compared to all of the existing OTS/FTS schemes. SDS-OTS offers 87% reduction in key and signatures sizes as compared to WOTS (adopted by IoTA), and more than 80% reduction in key and signature sizes as compared to WOTS$^+$ (adopted by QRL). SDS-OTS is also

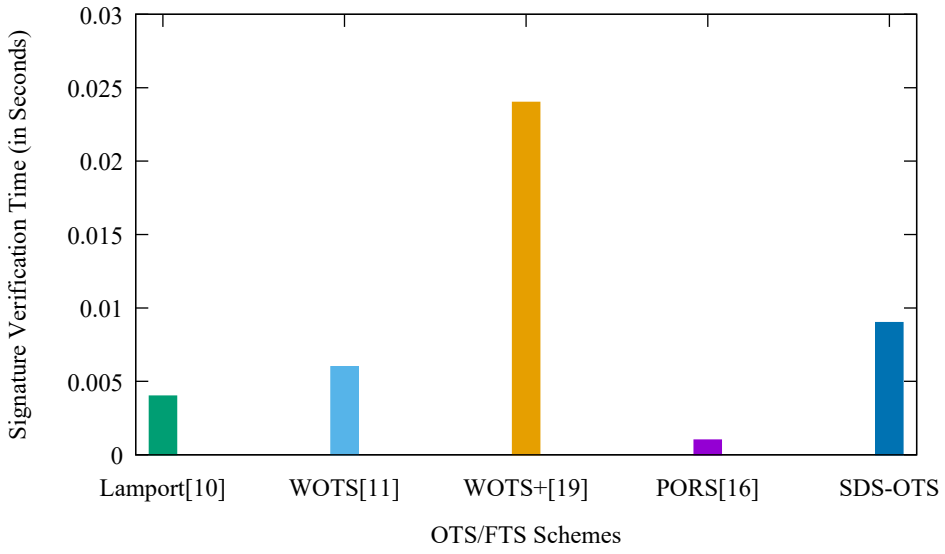


Figure 10: Sig. Verification Time (in seconds) of OTS/FTS Schemes

a computationally efficient scheme that offers 70% and 60% reductions in key-generation and signature-creation times respectively as compared to the existing most compact OTS scheme "WOTS $^{+}$". The paper also provides a road map for incorporating SDS into a distributed ledger, with the help of HLPN. For the future, we intend to incorporate SDS into the blockchain-technology beyond cryptocurrencies, for example, blockchain for Industrial Internet of Things or blockchain for the Internet of Energy, etc.

References

[1] K. Chalkias, J. Brown, M. Hearn, T. Lillehagen, I. Nitto, T. Schroeter, Blockchained post-quantum signatures, 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) (2018) 1196–1203doi:10.1109/Cybermatics_2018.2018.00213.

[2] F. Arute, K. Arya, R. Babbush *et al.*, Quantum supremacy using a programmable superconducting processor, Nature 574 (7779) (2019) 505–510. doi:10.1038/s41586-019-1666-5.
URL https://doi.org/10.1038/s41586-019-1666-5

[3] D. Aggarwal, G. Brennen, T. Lee, M. Santha, M. Tomamichel, Quantum attacks on bitcoin, and how to protect against them, Ledger 3 (10 2017). doi:10.5195/LEDGER.2018.127.

[4] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM J. Comput. 26 (5) (1997) 1484–1509. doi:10.1137/S0097539795293172.
URL http://dx.doi.org/10.1137/S0097539795293172

[5] W. Buchanan, A. Woodward, Will quantum computers be the end of public key encryption?, Journal of Cyber Security Technology (2016) 1–22doi:10.1080/23742917.2016.1226650.

[6] E. Dahmen, K. Okeya, T. Takagi, C. Vuillaume, Digital signatures out of second-preimage resistant hash functions, in: J. Buchmann, J. Ding (Eds.), Post-Quantum Cryptography, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 109–123. doi:10.1007/978-3-540-88403-3_8.

[7] R. E. Bansarkhani, M. Geihs, J. Buchmann, Pqchain: Strategic design decisions for distributed ledger technologies against future threats, IEEE Security Privacy 16 (4) (2018) 57–65. doi:10.1109/MSP.2018.3111246.

[8] S. Popov, The tangle version 1.4.3, in: Academic papers IoTA, 2018.
URL https://www.iota.org/research/academic-papers

[9] theQRL, The quantum resistant ledger, in: QRL white paper, 2016.
URL https://github.com/theQRL/Whitepaper/blob/master/QRL_whitepaper.pdf

[10] F. Shahid, A. Khan, G. Jeon, Post-quantum distributed ledger for internet of things, Computers Electrical Engineering 83 (2020) 106581. doi:https://doi.org/10.1016/j.compeleceng.2020.106581.
URL http://www.sciencedirect.com/science/article/pii/S004579061932659X

[11] A. Hülsing, W-ots+ – shorter signatures for hash-based signature schemes, in: A. Youssef, A. Nitaj, A. E. Hassanien (Eds.), Progress in Cryptology – AFRICACRYPT 2013, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 173–188. doi:10.1007/978-3-642-38553-7_10.

[12] J. Buchmann, E. Dahmen, A. Hülsing, Xmss - a practical forward secure signature scheme based on minimal security assumptions, in: B.-Y. Yang (Ed.), Post-Quantum Cryptography, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 117–129. doi:10.1007/978-3-642-25405-5_8.

[13] R. C. Merkle, A certified digital signature, in: G. Brassard (Ed.), Advances in Cryptology — CRYPTO' 89 Proceedings, Springer New York, New York, NY, 1990, pp. 218–238. doi:10.1007/0-387-34805-0_21.

[14] L. Lamport, Constructing digital signatures from a one-way function, in: Tech. Rep, 1979.

[15] A. Hülsing, L. Rausch, J. Buchmann, Optimal parameters for xmssmt, in: A. Cuzzocrea, C. Kittl, D. E. Simos, E. Weippl, L. Xu (Eds.), Security Engineering and Intelligence Informatics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 194–208. doi:10.1007/978-3-642-40588-4_14.

[16] A. Hülsing, J. Rijneveld, F. Song, Mitigating multi-target attacks in hash-based signatures, in: C.-M. Cheng, K.-M. Chung, G. Persiano, B.-Y. Yang (Eds.), Public-Key Cryptography – PKC 2016, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 387–416.

[17] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, Z. Wilcox-O'Hearn, Sphincs: Practical stateless hash-based signatures, in: E. Oswald, M. Fischlin (Eds.), Advances in Cryptology – EUROCRYPT 2015, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 368–397. doi:10.1007/978-3-662-46800-5_15.

[18] S. Gueron, N. Mouha, Sphincs-simpira: Fast stateless hash-based signatures with post-quantum security, IACR Cryptology ePrint Archive 2017 (2017) 645.

[19] J.-P. Aumasson, G. Endignoux, Improving stateless hash-based signatures, in: N. P. Smart (Ed.), Topics in Cryptology – CT-RSA 2018, Springer International Publishing, Cham, 2018, pp. 219–242. doi:10.1007/978-3-319-76953-0_12.

[20] A. K. Lenstra, Key length. contribution to the handbook of information security (2004).

[21] Z. Wang, C. Cao, N. Yang, V. Chang, Abe with improved auxiliary input for big data security, Journal of Computer and System Sciences 89 (2017) 41 – 50. doi:https://doi.org/10.1016/j.jcss.2016.12.006.
URL http://www.sciencedirect.com/science/article/pii/S0022000016301349

[22] Z. Wang, S. M. Yiu, Attribute-based encryption resilient to auxiliary input, in: M.-H. Au, A. Miyaji (Eds.), Provable Security, Springer International Publishing, Cham, 2015, pp. 371–390.

[23] Y. Yang, X. Zheng, W. Guo, X. Liu, V. Chang, Privacy-preserving smart iot-based healthcare big data storage and self-adaptive access control system, Information Sciences 479 (2019) 567 – 592. doi:https://doi.org/10.1016/j.ins.2018.02.005.
URL http://www.sciencedirect.com/science/article/pii/S0020025518300860

[24] W. Yin, Q. Wen, W. Li, H. Zhang, Z. Jin, An anti-quantum transaction authentication approach in blockchain, IEEE Access 6 (2018) 5393–5401. doi:10.1109/ACCESS.2017.2788411.

[25] Y. Gao, X. Chen, Y. Chen, Y. Sun, X. Niu, Y. Yang, A secure cryptocurrency scheme based on post-quantum blockchain, IEEE Access 6 (2018) 27205–27213. doi:10.1109/ACCESS.2018.2827203.

[26] C. Li, X. Chen, Y. Chen, Y. Hou, J. Li, A new lattice-based signature scheme in post-quantum blockchain network, IEEE Access 7 (2019) 2026–2033. doi:10.1109/ACCESS.2018.2886554.

[27] W. Kong, J. Shen, P. Vijayakumar, Y. Cho, V. Chang, A practical group blind signature scheme for privacy protection in smart grid, Journal of Parallel and Distributed Computing 136 (2020) 29 – 39. doi:https://doi.org/10.1016/j.jpdc.2019.09.016.
URL http://www.sciencedirect.com/science/article/pii/S0743731519301285

[28] A. S. Sohal, R. Sandhu, S. K. Sood, V. Chang, A cybersecurity framework to identify malicious edge device in fog computing and cloud-of-things environments, Computers & Security 74 (2018) 340 – 354. doi:https://doi.org/10.1016/j.cose.2017.08.016.
URL http://www.sciencedirect.com/science/article/pii/S0167404817301827