Revisiting the Constant-sum Winternitz One-time Signature with Applications to SPHINCS$^+$ and XMSS

Kaiyi Zhang[1 [0000-0002-2294-3523]], Hongrui Cui[1 [0000-0002-6203-413X]], and Yu Yu[1,2 [0000-0002-9278-4521]]

[1] Department of Computer Science, Shanghai Jiao Tong University, 200240 Shanghai {kzoacn,rickfreeman}@sjtu.edu.cn

[2] Shanghai Qizhi Institute, 200232 Shanghai yuyu@yuyu.hk

Abstract. Hash-based signatures offer a conservative alternative to post-quantum signatures with arguably better-understood security than other post-quantum candidates.

As a core building block of hash-based signatures, the efficiency of one-time signature (OTS) largely dominates that of hash-based signatures. The WOTS$^+$ signature scheme (Africacrypt 2013) is the current state-of-the-art OTS adopted by the signature schemes standardized by NIST—XMSS, LMS and SPHINCS$^+$.

A natural question is whether there is (and how much) room left for improving one-time signatures (and thus standard hash-based signatures). In this paper, we show that WOTS$^+$ one-time signature, when adopting the constant-sum encoding scheme (Bos and Chaum, Crypto 1992), is size-optimal not only under Winternitz's OTS framework, but also among all tree-based OTS designs. Moreover, we point out a flaw in the DAG-based OTS design previously shown to be size-optimal at Asiacrypt 1996, which makes the constant-sum WOTS$^+$ the most size-efficient OTS to the best of our knowledge. Finally, we evaluate the performance of constant-sum WOTS$^+$ integrated into the SPHINCS$^+$ (CCS 2019) and XMSS (PQC 2011) signature schemes which exhibits certain degrees of improvement in both signing time and signature size.

Keywords: Hash-Based Signature, Post-Quantum Cryptography, SPHINCS$^+$

1 Introduction

Hash-based signatures are one of the most promising candidates for (and perhaps the most conservative approach to) post-quantum digital signatures. An advantage of hash-based signatures is that its (classical as well as quantum) security strength is better understood (and easier to evaluate) than other candidates, by solely relying on the idealized hardness[3] of the cryptographic hash functions.

[3] The design philosophy of symmetric primitives (including hash functions) is that they should only admit generic attacks, otherwise the design is considered to be flawed.

Lamport [28] and Rabin [36] proposed the first one-time signature (OTS) schemes that can be efficiently built from one-way functions (aka. the minimal assumption). The design was later made more efficient by Winternitz [30], Bos and Chaum [9], Vaudenay [41], and Hülsing's WOTS$^+$ scheme [22], which is the current state of the art. The subsequent work often adopts more complicated structures, and typically relies on hash functions with stronger assumptions.

Another line of works extend OTS to full-fledged signatures capable of signing multiple messages. In the context of hash-based signatures, the goals can be divided into *stateful* signatures and *stateless* ones, depending on whether or not the signer needs a state to keep track of signed messages. As far as *stateful* signatures are concerned, Merkle first proposed to sign multiple messages via a binary hash tree [31]. Merkle's original proposal was improved and optimized to become the eXtended Merkle Signature Scheme (XMSS) [23] and the Leighton-Micali Signature (LMS) [29], which are standardized by NIST [10] and IETF. As for stateless hash-based signatures, Goldreich proposed the first *stateless* construction [16,17], which removes the need for maintaining a local state but results in prohibitively large signatures. SPHINCS [4] offers a practical instantiation of the Goldreich-style stateless hash-based signature and serves as a basis for subsequent works, including Gravity-SPHINCS [2], SPHINCS-Simpira [18], and SPHINCS$^+$ [5]. Recently, SPHINCS$^+$ was selected as future standard signatures by the NIST PQC standardization process [39].

WOTS$^+$ **and hash-based signatures.** The hash-based signatures to be standardized by NIST [5,23,29], whether stateless or stateful, all extensively rely on and invoke many times the WOTS$^+$ one-time signature as an important underlying building block. Therefore, improving the efficiency of WOTS$^+$ will bring about a corresponding increase in the resulting hash-based signature.

How WOTS$^+$ **encodes its message.** A line of works [9,11,34,26] focused on optimizing the message encoding scheme of the WOTS$^+$ in order to build more efficient OTS. The encoding problem in the Winternitz's OTS framework can be informally summarized as: every message m parsed as the base-w representation $m = (m_1, \ldots, m_{l_1}) \in \mathcal{M} \subseteq [w]^{l_1}$, where $[w] \stackrel{\text{def}}{=} \{0, 1, \ldots, w-1\}$, should be injectively mapped into codeword $(c_1, \ldots, c_{l_1+l_2}) \in \mathcal{C} \subseteq [w]^{l_1+l_2}$ such that there exist no distinct $(c_1, \ldots, c_{l_1+l_2}), (c'_1, \ldots, c'_{l_1+l_2}) \in \mathcal{C}$ satisfying $\forall i \in \{1, \ldots, l_1+l_2\}: c_i \leq c'_i$. Otherwise, it leads to a trivial forgery attack on the OTS scheme. Note that the encoding rate $(1+l_2/l_1)$ translates to the average signature size per message bit[4]. The current WOTS$^+$ scheme [22] adopts a simple yet efficient encoding scheme by simply appending a checksum to the message, i.e., fix message space $\mathcal{M} = [w]^{l_1}$ and let the encoding be $(m_1, \ldots, m_{l_1}) \mapsto (m_1, \ldots, m_{l_1}, c)$, where the checksum $c = \sum_{i=1}^{l_1}(w - 1 - m_i)$ is represented in base-w as well. A natural idea to improve the encoding rate is to choose only those m with a fixed

[4] In fact, the average number of hash function evaluations during KeyGen is $(w-1) \cdot (1 + l_2/l_1)$ (which is equal to the total number of hash function evaluations during Sign and Verify), and therefore the encoding rate is also related to computational efficiency, which is consistent with the experimental results in Sect. 5.

(constant) checksum value c (so that c doesn't need to appear in the codeword explicitly), i.e., let $\mathcal{C} = \{(m_1, \ldots, m_l) \in [w]^l : \sum_{i=1}^{l} m_i = s\} \subseteq [w]^l$ and construct an efficient encoding algorithm $\mathsf{Enc} : \mathcal{M} \to \mathcal{C}$, where the message space is maximized when $s = \lfloor \frac{l(w-1)}{2} \rfloor$ (among all possible values for s). This encoding is referred to as the constant-sum encoding. Bos and Chaum [9] first proposed the constant-sum encoding in the binary setting (i.e., $w = 2$). Vaudenay [41] extended it to the arbitrary w setting but did not give an explicit encoding algorithm. Curz et al. [11] proposed a probabilistic encoding algorithm. More recently, Perin et al. [34] introduced an efficient deterministic encoding algorithm. Kudinov et al. [26] introduce an efficient encoding method (via rejection sampling) for constant-sum encoding, and integrated it into the SPHINCS$^+$ algorithm to achieve performance improvement.

Motivation. It is therefore natural to ask the following questions in the pursuit of more efficient digital signatures or in order to avoid further futile efforts.

Question 1: Does the constant-sum encoding already achieve the optimal encoding rate or is there a better encoding scheme in the WOTS$^+$ framework?

Question 2: Are there OTS schemes with better signature size and computational efficiency in a more general framework?

Our contributions. We answer the first question affirmatively, and provide both positive and negative results for the second one.

- For Question 1, we show that the constant-sum encoding achieves the optimal encoding rate among all encoding schemes in the Winternitz-style OTS framework. Following previous observation [6], we show this by first interpreting the problem of maximizing the message space (for fixed-length codewords) as an order-theoretic problem of finding the largest anti-chain in the induced partially ordered set. Then, using Dilworth's theorem, we show that the anti-chain size is maximized when the elements sums to half of the maximally allowable value, which corresponds to the constant-sum encoding.
- For Question 2, we first show that the DAG-based OTS design previously considered asymptotically optimal [7,13] contains a security flaw, which may lead to trivial forgery attacks. On the positive side, we show that the constant-sum WOTS$^+$ maximizes message space among all *tree-based* OTS schemes. We prove this result by adapting the technique of Bleichenbacher and Maurer [8] in the binary tree setting to the arbitrary tree structure.

We conclude that the constant-sum WOTS$^+$ scheme not only achieves optimal encoding rate in the WOTS$^+$ framework, it also maximizes the message space among all tree-based OTS schemes. Further, after refuting the DAG-based designs [7,21,13] we're not aware of any other more size-efficient DAG-based design.

On the practical side, we replace the WOTS$^+$ component in SPHINCS$^+$ and XMSS with constant-sum WOTS$^+$, and evaluate the corresponding performance improvement[5]. For SPHINCS$^+$, by carefully adjusting the parameters, the resulting stateless signature scheme exhibits up to 12.4% reduction in signature size compared to the size-optimized variant of SPHINCS$^+$ at the 128-bit security level. We note that our experiment takes into account the fix [24] of the latest attack [27]. For XMSS, we simply change the encoding scheme to constant-sum while keeping the original parameter sets, which results in up to 1.78% reduction in signature size.

2 Preliminary

In this section, we define the notations, provide some basic background of order theory, and recall some previous constructions in the literature.

2.1 Notations

We use $[w] \stackrel{\text{def}}{=} \{0, 1, \ldots, w-1\}$ for $w \in \mathbb{N}^+$. We denote the i-th element of a vector $\boldsymbol{v}$ by v_i. By $\log(x)$ we refer to the binary logarithm, i.e., $\log_2(x)$. We denote the concatenation of strings (vectors) $\boldsymbol{a}$ and $\boldsymbol{b}$ by $\boldsymbol{a} \| \boldsymbol{b}$ or $(\boldsymbol{a}, \boldsymbol{b})$. For a set $\mathcal{S}$, we denote the size of $\mathcal{S}$ and the power set of $\mathcal{S}$ by $|\mathcal{S}|$ and $P(\mathcal{S})$ respectively. We let λ be the security parameter, and refer to a λ-bit value as a block. Let $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^*$ be a hash function.

2.2 Preliminaries of Order Theory

Definition 1 (Poset). *A poset $(\mathcal{S}, \leq)$ consists of a set $\mathcal{S}$ together with an anti-symmetric, transitive and reflexive binary relation '$\leq$', according to which certain pairs $(x, y) \in \mathcal{S}$ are comparable ($x \leq y$ or $y \leq x$).*

Note that a poset does not require all pairs in $\mathcal{S}$ to be comparable, and thus it is also known as a partially ordered set.

Definition 2 ((Anti)chain and decomposition). *A chain (resp., antichain) refers to a subset of a poset, for which every pair of elements is comparable (resp., incomparable). A chain decomposition is a partition of a poset into disjoint chains.*

Theorem 1 (Dilworth's theorem [12]). *For any finite poset $\mathcal{S}$, the size of $\mathcal{S}$'s maximum antichain equals the size of $\mathcal{S}$'s minimum chain decomposition.*

[5] We dub the optimized SPHINCS$^+$ scheme as SPHINCS-α, and a self-contained description of that hash-based signature is available in [45]. We stress that focus of this paper is one-time signatures and thus we do not include the additional details of SPHINCS-α other than the OTS component in this paper.

2.3 Hash-based One-time Signature

Here we recall the original construction of one-time signature by Lamport [28] and various optimizations that leads to the currently widely used WOTS$^+$ scheme [22].

Lamport One-Time Signature. Suppose the length of the message is λ, we describe the Lamport signature scheme as follows:

- **KeyGen**: On input 1^λ, for each $i \in \{1, \ldots, \lambda\}$ choose two uniform strings $x_{i,0}, x_{i,1} \leftarrow \{0,1\}^\lambda$ and compute $y_{i,0} = \mathsf{H}(x_{i,0}), y_{i,1} = \mathsf{H}(x_{i,1})$. Define the public key as $\mathsf{pk} := \{(y_{i,0}, y_{i,1})\}_{i\in[1,\lambda]}$ and private key as $\mathsf{sk} := \{(x_{i,0}, x_{i,1})\}_{i\in[1,\lambda]}$.
- **Sign**: On input a private key sk and a message $m \in \{0,1\}^\lambda$. Interpret m as a string of base-2 values $(m_1, m_2, \ldots, m_\lambda)$. Output the signature $\sigma = (x_{1,m_1}, \ldots, x_{\lambda,m_\lambda})$.
- **Verify**: On input a public key pk, a message $m \in \{0,1\}^\lambda$ and a signature $\sigma = (x_1, x_2, \ldots, x_\lambda)$, output 1 iff $\mathsf{H}(x_i) = y_{i,m_i}$ for all $i \in [\lambda]$.

The above scheme can be proved secure if H is a one-way function [17]. Nevertheless, it can only sign one message since given two signatures an adversary can forge a new signature by reordering those preimage of hash values.

From OTS to General Signature. To enable signing multiple messages (of length λ), Goldreich [16] proposes to use a binary tree of depth λ where each node is associated with an OTS public/secret key pair and authenticates the public keys of its children nodes. Therefore, every message in $\{0,1\}^\lambda$ can be signed by a unique OTS public key on the corresponding leaf node.

Nevertheless, generating this tree takes exponential time. Instead we use a pseudorandom function to generate it "on-the-fly". That is, we use a pseudorandom function to compress all the randomness of the tree. To sign a message $m \in \{0,1\}^\lambda$, the signer computes the path from root to a leaf corresponding to the binary representation of m. For each node u in the path, the signer generate the node and its two children $u0, u1$ and add $\sigma_u = \text{Sign}(sk_u, pk_{u0}||pk_{u1})$ to the final signature. To verify the signature, the verifier checks that each node except the root is correctly signed by its parent and the path corresponds to the message m.

Improved OTS from Sperner Family. In Lamport's OTS scheme signing a λ-bit message takes λ hash blocks but message space can be enlarged in the following way. Briefly speaking, the Sperner family is defined by $\mathcal{S} = \{S : S \subseteq [n] \wedge |S| = \lfloor n/2 \rfloor\}$. It has some properties:

- $|\mathcal{S}| = \binom{n}{\lfloor n/2 \rfloor}$.
- It is (one of) the largest family in which no set contains any other set (in this family).

Let n be the smallest integer such that $\binom{n}{\lfloor n/2 \rfloor} \geq 2^\lambda$. Informally, the second property ensures that given any valid signature, it is computationally infeasible for any adversary to forge a new valid signature since signature patterns do not cover each other. We describe this improved OTS scheme below:

– **KeyGen**: On input 1^λ, for each $i \in [n]$ choose a uniform string $x_i \in \{0,1\}^\lambda$ and compute $y_i = \mathsf{H}(x_i)$. The public key and secret key are defined similar to Lamport OTS.
– **Sign**: On input a private key sk and a message $m \in \{0,1\}^\lambda$. Encode m into $S \in \mathcal{S}$, output the signature $\sigma = \{x_i\}_{i \in S}$.
– **Verify**: On input a public key pk, a message $m \in \{0,1\}^\lambda$ and a signature σ, output 1 if $\mathsf{H}(x_i) = y_i$ for all $i \in S$.

Looking ahead to Sect. 3, we will see this encoding method is a special case of the constant-sum encoding method (for $w = 2$), where we show that the more general scheme achieves maximum message space, and provide an efficient encoding algorithm.

Winternitz One-Time Signature. Denote w as the Winternitz parameter. Let l be the number of blocks in an uncompressed WOTS$^+$ private key, public key, and signature, where

$$l = l_1 + l_2, l_1 = \left\lceil \frac{\lambda}{\log(w)} \right\rceil, l_2 = \left\lfloor \frac{\log(l_1(w-1))}{\log(w)} \right\rfloor + 1 \ .$$

Let $\mathsf{H}^a(x) \stackrel{\text{def}}{=} \mathsf{H}(\mathsf{H}^{a-1}(x))$ and $\mathsf{H}^0(x) = x$. We present WOTS$^+$ as follows:

– **KeyGen**: On input 1^λ, for each $i \in \{1,...,l\}$ choose a uniform string $x_i \in \{0,1\}^\lambda$ and compute $y_i = \mathsf{H}^{w-1}(x_i)$. Define the public key and private key as $\mathsf{pk} := \mathsf{H}(y_1, \dots, y_l)$ and $\mathsf{sk} := \{x_i\}_{i \in [1,l]}$.
– **Sign**: On input a private key sk and a message $m \in \{0,1\}^\lambda$. Encode m into its base-w representation $(m_1, \dots, m_{l_1})$. Then compute the checksum $c = \sum_{i=1}^{l_1}(w - 1 - m_i)$ and represent c in base-w as $(c_1, \dots, c_{l_2})$. Let $M = (m_1, \dots, m_{l_1}, c_1, \dots, c_{l_2})$. For each $i \in [l]$ output the signature $\sigma_i = \mathsf{H}^{M_i}(x_i)$.
– **Verify**: On input a public key pk, a message $m \in \{0,1\}^\lambda$ and a signature σ, output 1 if $\mathsf{H}(\mathsf{H}^{w-1-M_1}(\sigma_1), \dots, \mathsf{H}^{w-1-M_l}(\sigma_l)) = \mathsf{pk}$.

The reason that the WOTS$^+$ (as well as other Winternitz-type OTS) scheme introduces the checksum is that in absence of the checksum the adversary can efficiently forge signatures given a single pair of valid message signature. That is, given (σ, m) he forges any m' satisfying $\forall i, m_i \leq m'_i$ by computing $\mathsf{H}^{m'_i}(\mathsf{sk}_i) = \mathsf{H}^{m'_i - m_i}\big(\mathsf{H}^{m_i}(\mathsf{sk}_i)\big)$.

The checksum addresses the issue: an increase in any m_i leads to decreasing at least one c_i (recall $c = \sum_{i=1}^{l_1}(w - 1 - m_i)$). Therefore, the adversary cannot forge any (m', c') simultaneously satisfying both $m_i \leq m'_i$ and $c_i \leq c'_i$ for $i \in [l]$.

3 Constant-sum WOTS$^+$

In this section, we recall the constant-sum encoding scheme, and prove the size optimality of constant-sum in WOTS$^+$ using order theory.

3.1 Size-optimal Encoding

More formally, the problem of constructing one-time signature reduces to that of building an efficient encoding scheme $\mathsf{Enc} : \mathcal{M} \to \mathcal{C} \subseteq [w]^l$ for some incomparable codeword set $\mathcal{C}$ (see Definition 3). In case of WOTS$^+$, the encoding function Enc simply appends the checksum to the original message. Note that WOTS$^+$ fixes the size of the message to l_1 (i.e., $\mathcal{M} = [w]^{l_l}$) and then constructs as small codewords as possible (minimizing $l - l_1$).

Definition 3 ((In)comparability). *For $\boldsymbol{c}, \boldsymbol{c}' \in [w]^l$, we denote by $\boldsymbol{c} \leq \boldsymbol{c}'$ if for every $i \in [l]$ we have $c_i \leq c'_i$. If $\boldsymbol{c} \leq \boldsymbol{c}'$ or $\boldsymbol{c}' \leq \boldsymbol{c}$ we say that $\boldsymbol{c}$ and $\boldsymbol{c}'$ comparable, or otherwise they are incomparable. A set $S \subseteq [w]^l$ is said to be incomparable (or called an "antichain" in order theory terminology) if any two elements of S are incomparable.*

We take a slightly different approach to encoding the messages. That is, we first fix the size of the codewords to l, $\mathcal{C} \subseteq [w]^l$, and strive to accommodate as large message space $\mathcal{M}$ as possible. Given that Enc is an injection it is essentially to maximize the size of $\mathcal{C} \subseteq [w]^l$. A natural approach is to encode the codewords such that all elements of every codeword sum to the same value, and therefore the checksum is not explicitly needed.

Theorem 2 ([9,41]). *For any $s \in [l(w-1)+1]$, $\mathcal{C}_s \stackrel{\text{def}}{=} \{\boldsymbol{c} \in [w]^l : \sum_{i=1}^{l} c_i = s\}$ is incomparable.*

Proof: Suppose towards contradiction that $\mathcal{C}_s$ (for some fixed $s \in [l(w-1)+1]$) is not incomparable, then there exist distinct $\boldsymbol{c}, \boldsymbol{c}' \in \mathcal{C}_s$ s.t. $\boldsymbol{c} \leq \boldsymbol{c}'$. There must be an index j such that $c_j < c'_j$ (otherwise $\boldsymbol{c} = \boldsymbol{c}'$). However, due to equal sum $\sum_i c_i = \sum_i c'_i$ we have $\sum_{1 \leq i \leq l \wedge i \neq j}(c_i - c'_i) > 0$, and there must exist some $1 \leq k \leq l$ such that $c_k > c'_k$, which is a contradiction to $\boldsymbol{c} \leq \boldsymbol{c}'$. ∎

Every $\mathcal{C}_s$ gives an encoding scheme but with different size. For $s = 0$ or $s = l(w-1)$, $\mathcal{C}_s$ consists of only a single codeword. We argue that the size of $\mathcal{C}_s$ reaches its maximal in the middle, i.e., when $s = \lfloor \frac{l(w-1)}{2} \rfloor$. One easily verifies that this holds in the binary case (i.e., $w = 2$) where $|\mathcal{C}_s| = \binom{l}{s}$. We note that this encoding method appears previously in the literature [9,41], and Perin et al. [34] proved that $|\mathcal{C}_s|$ reaches its maximum when $s = \lfloor \frac{l(w-1)}{2} \rfloor$. But to the best of our knowledge, we are the first to present a size-optimality proof over all encoding schemes in WOTS$^+$. In particular, we prove in Theorem 3 that the size of $\mathcal{C}_s$, when $s = \lfloor \frac{l(w-1)}{2} \rfloor$, is not only the largest in all $\mathcal{C}_s$ for $s \in [l(w-1)+1]$ but the largest among all valid sets of codewords.

Theorem 3 (Size-optimal encoding). *For every incomparable $\mathcal{C}^* \in P([w]^l)$, it holds that*

$$|\mathcal{C}^*| \leq |\mathcal{C}_{\lfloor \frac{l(w-1)}{2} \rfloor}| \ .$$

We defer its proof to Theorem 4, which rephrases Theorem 3 in the language of order theory. Prior to that, we discuss how to compute $|\mathcal{C}_s|$ by recursion, and give an explicit construction of encoding messages into $\mathcal{C}_s$ for $s = \lfloor \frac{l(w-1)}{2} \rfloor$. Hereafter, we denote such $\mathcal{C}_s$ with maximal size by $\mathcal{C}$ for brevity.

Counting the size. Now we need to figure out the size of $\mathcal{C}$. As a special case, $|\mathcal{C}| = \binom{l}{\lfloor l/2 \rfloor}$ when $w = 2$. Fix w, let

$$D_{l,s} = |\{\boldsymbol{c} \in [w]^l : \sum_{i=1}^{l} c_i = s\}| \ ,$$

we have their initial values

$$\begin{aligned} D_{0,0} &= 1, \\ D_{0,s} &= 0, \ \text{for } s \in \{1, 2, \ldots, w-1\} \\ D_{l,s} &= 0, \ \text{for } 1 \leq l \in \mathbb{Z}, s \in \mathbb{Z}^- \ , \end{aligned}$$

and recurrence relation

$$D_{l,s} = \sum_{i=0}^{w-1} D_{l-1,s-i}, 2 \leq l \in \mathbb{Z}, s \in \{0, 1, \ldots, l(w-1)\} \ .$$

Note when $w = 2$, this method is equivalent to recurrence relation of binomial coefficient, i.e., $\binom{l}{s} = \binom{l-1}{s-1} + \binom{l-1}{s}$.

Let us explain the recurrence relation. To compute $D_{l,s}$, consider the value of its last summand, which could be any value in $\{0, 1, \ldots, w-1\}$. If this value is set to i, the sum of the first $l-1$ elements must be $s-i$. Therefore, we notice that the problem "l elements with sum to s" into those "$l-1$ elements with sum to $s-i$". Thus we can simply count $D_{l,s}$ by accumulating $D_{l-1,s-i}$. Following this method, $D_{l,\lfloor l(w-1)/2 \rfloor}$ gives the size of $\mathcal{C}$.

We note that $D_{l,s}$ is also the s-th coefficient of $(1+x+x^2+\cdots+x^{w-1})^l$. Euler [15] has studied $w = 3, 4, 5$, known as trinomial, quadrinomial and quintinomial coefficients respectively. The generalized form was studied in the literature, e.g., [1,42,3]. Actually, we can use an inclusion-exclusion argument to express it as a function of binomial coefficients [43]

$$D_{l,s} = \sum_{i=0}^{\lfloor s/w \rfloor} (-1)^i \binom{l}{i} \binom{s+l-iw-1}{l-1} \ .$$

The encoding algorithm. Now we make the construction explicit by giving an efficient encoding algorithm[6], which maps a message $x \in [|\mathcal{C}|]$ into an element in $\mathcal{C}$. We give the pseudocode of the encoding procedure in Algorithm 1.

Let us explain the encoding algorithm. As previously stated, the problem can be divided into several sub-problems by considering the value of the first element

[6] We note that a similar algorithm was previously proposed by Perin et al. [34] and we stress that the encoding algorithm is included for the sole purpose of completeness and it is not considered as part of our contributions.

Algorithm 1: Encode:$[|\mathcal{C}|] \to \mathcal{C}$.

```
Function Encode(x)
    Let v be an array of size l;
    m ← ⌊l(w − 1)/2⌋;
    for i ← l ... 1 do
        for j ← 0 ... min(w − 1, m) do
            if x ≥ D_{i−1,s−j} then
                x ← x − D_{i−1,s−j};
            else
                v_{l−i} ← j;
                break;
        m ← m − v_{l−i};
    return v;
```

v_{l-i}. To encode a natural number $x \in [0, D_{i,m})$, we can simply determine $v_{l-i} = j$ by seeking which j satisfies $x \in [\sum_{k<j} D_{i-1,m-k}, \sum_{k\le j} D_{i-1,m-k})$. Once the value of v_{l-i} is determined, we proceed to the next terms until all elements are decided.

Now prove the encoding-rate optimality of the constant-sum scheme using order theory (recalled in Sect. 2.2), which has been shown to be closely related to the design of one-time signatures [6].

Theorem 4. *Let $S_l = ([w]^l, \le)$ be a finite poset and $\mathcal{C} := \{\boldsymbol{c} \in S_l | \sum_{i=1}^{l} c_i = \lfloor l(w-1)/2 \rfloor$. Then $\mathcal{C}$ is the maximum antichain in S_l.*

Proof: According to Dilworth's theorem, we can prove that $\mathcal{C}$ is the maximum antichain of S_l by arguing that (1) $\mathcal{C}$ is an antichain and (2) we can find a chain decomposition whose size equals to $|\mathcal{C}|$. We have proved that $\mathcal{C}$ is an antichain in Theorem 2. It remains to construct the chain decomposition of size $|\mathcal{C}|$ as follows. Our proof can be viewed as a generalization of the proof of Sperner's theorem [38], which considers the special case for $w = 2$.

Consider poset $S_l = ([w]^l, \le)$, and we denote its element by $\boldsymbol{c} := (c_1, ..., c_l) \in [w]^l$ and differentiate different elements using superscript. We slightly abuse the notation by $|(c_1, ..., c_n)| \stackrel{\text{def}}{=} c_1 + ... + c_n$.

We construct the chain decomposition for S_l by induction, where every chain $\boldsymbol{c}^1 \le \ldots \le \boldsymbol{c}^t$ satisfies the following two properties:

- $|\boldsymbol{c}^{i+1}| = |\boldsymbol{c}^i| + 1, \forall i \in \{1, 2, \ldots, t-1\}$,
- $|\boldsymbol{c}^1| + |\boldsymbol{c}^t| = l \cdot (w-1)$.

The case for $l = 1$ is trivial, i.e., $D_{1,\lfloor (w-1)/2 \rfloor} = 1$, which corresponds to the chain $(0) \le (1) \le \ldots \le (w-1)$.

Assume that we have a chain decomposition for S_{l-1} satisfying the above two properties, we proceed to the construction of a chain decomposition for S_l. By the inductive assumption we have the chain decomposition for S_{l-1} satisfying the two properties. For any chain $\boldsymbol{c}^1 \leq \boldsymbol{c}^2 \leq \ldots \leq \boldsymbol{c}^t$ from the aforementioned decomposition of S_{l-1}, we build $k+1$ chains for S_l as follows, where $k = \min(w-1, t-1)$. That is, for every $j \in \{0, ..., k\}$ the j-th chain consists of:

$$(\boldsymbol{c}^1, j) \leq \ldots \leq (\boldsymbol{c}^{t-j}, j) \leq (\boldsymbol{c}^{t-j}, j+1) \leq ... \leq (\boldsymbol{c}^{t-j}, w-1) \ .$$

This yields the $k+1$ chains as shown in Fig. 1.

$$\begin{array}{cccccccc}
(\boldsymbol{c}^1, 0) \leq \ldots & \ldots & \ldots & \leq (\boldsymbol{c}^t, 0) \leq \ldots & & \leq (\boldsymbol{c}^t, w-1) \\
\vdots \quad \ldots & \ldots & \iddots & \ldots & \ldots & \vdots \\
(\boldsymbol{c}^1, k) \leq \ldots \leq (\boldsymbol{c}^{t-k}, k) \leq \ldots & & \ldots & & \ldots & \leq (\boldsymbol{c}^{t-k}, w-1)
\end{array}$$

Fig. 1. A demonstration of how a chain from S_{l-1} is expanded into $k+1$ chains for S_l, where every row is an expanded chain. Note that it is not a rectangular matrix (every row has two less elements than the previous).

It is easy to verify that $|(\boldsymbol{c}^1, j)| + |(\boldsymbol{c}^{t-j}, w-1)| = |(\boldsymbol{c}^1, 0)| + j + |(\boldsymbol{c}^t, 0)| - j + (w-1) = l(w-1)$, and every subsequent element increase the sum value of its predecessor by one. Namely, the two properties are preserved for all the constructed chains of S_n.

It remains to argue that all the chains constructed (from the decomposed chains of S_{l-1}) constitute a partition of $S_l := [w]^l$. That is, for every $\boldsymbol{c}^i \in S_{l-1}$, each of its augmented elements $(\boldsymbol{c}^i, 0), ..., (\boldsymbol{c}^i, w-1)$ appears in the constructed chains exactly once. Note that every $\boldsymbol{c}^i$ belongs to exactly one of the decomposed chains of S_{l-1}, say $\boldsymbol{c}^1 \leq \ldots \leq \boldsymbol{c}^t$. We discuss the following cases.

Case $t \leq w$. We have $k = t-1 \leq w-1$. Viewing the elements in Fig. 1 as a matrix by filling the lower right corner with zeros, we have $[(\boldsymbol{c}^i, 0), \ldots, (\boldsymbol{c}^i, k+1-i)]$ appears as the first $(k+2-i)$ elements of the i-th column, and then $[(\boldsymbol{c}^i, k+1-i), \ldots, (\boldsymbol{c}^i, w-1)]^T$ as the last $(w+i-k-1)$ elements of the $(k+2-i)$-th row.
Case $t > w$. We have $k = w-1 < t-1$. If $1 \leq i \leq t-w+1$, then $[(\boldsymbol{c}^i, 0), ..., (\boldsymbol{c}^i, w-1)]$ appears as the i-th column in Fig. 1. Otherwise, it holds that $t-w+1 < i \leq t$. $[(\boldsymbol{c}^i, 0), \ldots, (\boldsymbol{c}^i, t-i)]$ and $[(\boldsymbol{c}^i, t-i), \ldots, (\boldsymbol{c}^i, w-1)]^T$ are the first $t-i+1$ elements of the i-th column, and the last $(w+i-t)$ elements of the $(t-i+1)$-th row respectively.

Therefore, we have shown that for every $\boldsymbol{c} \in [w]^{n-1}$, $(\boldsymbol{c}, 0)$, …,$(\boldsymbol{c}, w-1)$ appears exactly once in the newly constructed chains, namely, the chains constitutes as a chain decomposition for S_l. Finally, it remains to count the number of chains in the decomposition. The two properties guarantee that every chain contains exactly one element $\boldsymbol{c}^{\mathsf{mid}}$ with $|\boldsymbol{c}^{\mathsf{mid}}| = \lfloor l(w-1)/2 \rfloor$ (i.e., $\boldsymbol{c}^{\mathsf{mid}} \in \mathcal{C}$). Thus, the size of chain decomposition is $|\mathcal{C}| = D_{l, \lfloor l(w-1)/2 \rfloor}$. This completes the proof that $\mathcal{C}$ is the maximum antichain. ∎

3.2 Theoretical Performance

The constant-sum WOTS$^+$ has two advantages over the original WOTS$^+$.

- Constant computing time. The number of hash function calls is fixed, in contrast to possibly variable numbers for the signing and verification algorithm of WOTS$^+$. While no timing attacks are identified against the implementations of WOTS$^+$, stable computing time is always preferable (especially for signing algorithms whose computation involves a private key).
- Reduced signature size and hash calls. For instance, the SPHINCS$^+$-256s parameter set suggests $w = 16$ and $l = 67$. In constant-sum WOTS$^+$, for $w = 16$ we require $l = 66$, which reduces 1.5% in both running time (in terms of the expected number of hash function calls) and size. We refer to Table 1 for more details.

Table 1. Comparison of length l between WOTS$^+$ and constant-sum WOTS$^+$ for different values of Winternitz parameters w and security parameter λ ("CS" denotes constant-sum).

	128-bit		192-bit		256-bit	
w	WOTS$^+$	CS	WOTS$^+$	CS	WOTS$^+$	CS
8	46	45	67	66	90	88
16	35	34	51	50	67	66
24	31	30	45	44	59	58
32	28	27	42	40	55	53
40	27	26	39	38	52	50
48	25	25	37	36	48	48

Although the encoding algorithm of constant-sum WOTS$^+$ costs slightly more than the checksum method, it is less dominant compared to the number of hash function calls used in the signature scheme, which will be confirmed in the experiments.

4 Graph-Based One-Time Signature

In this section, we prove that the constant-sum WOTS$^+$ scheme achieves the maximum message space among all tree-based OTS with the same graph size. Moreover, we point out a flaw in the graph-based design previously considered optimal [7], which leaves the constant-sum WOTS$^+$ the most size-optimal among all existing schemes to the best of our knowledge. We begin by recalling the graph-based OTS notations and then present our proof.

4.1 DAG-based One-time Signature

Since Lamport introduced the construction of one-time signature based on one-way function [28], there has been various works improving the efficiency of such construction. The state-of-the-art analysis framework is by modelling the internal computation structure as a directed acyclic graph [6,21,13]. In this subsection, We recall the notations and definitions which mainly come from [6,13].

Without loss of generality we consider DAGs with only one sink vertex r (i.e., with out-degree zero). Given a DAG $G = (V, E)$, the secret key vertices $SK \subseteq V$ is defined as the sets of vertices with in-degree zero and the public key vertex $PK \subseteq V$ is $\{r\}$ (i.e. the sink vertex). Let X be a subset of V. A vertex w is defined recursively to be computable from X if either $w \in X$ or all predecessors of w are computable from X. A set $Y \subseteq V$ is computable from X if any $y \in Y$ is computable from X.

A set $X \subseteq V$ is called verifiable if r is computable from X. A verifiable set X is minimal if no proper subset of X is verifiable.

We define the set of all minimal verifiable sets (MVSs) of a DAG G as G^*, and additionally define the following binary relation on G^*.

Definition 4. *Given a DAG $G = (V, E)$ and G^*, we define the relation $U \leq V$ for two verifiable sets $U, V \in G^*$ if U is computable from V.*

With the binary relation the set G^* becomes a partially ordered set (poset). We additionally call the two verifiable sets $U, V \in G^*$ incomparable if neither $U \leq V$ nor $V \leq U$. The following lemma shows that any DAG with only one sink vertex implies a one-time signature scheme, which was proved in [21,13].

Lemma 1. *Given a DAG $G = (V, E)$ with only one sink vertex r and n source nodes $s_1, ..., s_n$, we can define the following one-time signature scheme. The secret key is a length-n vector of λ-bit blocks $\mathsf{sk}_1, ..., \mathsf{sk}_n$, each one corresponding to a source vertex. We recursively define $\mathsf{label}(u)$ of each vertex $u \in G$ as follows:*

- *If $u = s_i$ then $\mathsf{label}(u) = \mathsf{sk}_i$*
- *Otherwise, $\mathsf{label}(u) = \mathsf{H}(\mathsf{label}(u_1), ..., \mathsf{label}(u_k))$ where $u_1, ..., u_k$ are the predecessors of vertex u.*

The public key is $\mathsf{label}(r)$. Fix an antichain $\mathcal{A}$ of G^*, the message m in the message space $\mathcal{M} := \{0, \ldots, |\mathcal{A}|-1\}$ is mapped to the m-th MVS in the antichain $\mathcal{A}$ (which is also referred to as the signature scheme) . The properties of MVS guarantee that one can generate the labels of any verifiable sets in $\mathcal{A}$ from the source vertices (signing keys) and derive the label of the sink node (public key) from the labels of any verifiable sets.

We list a table below to show the relationship between a directed acyclic graph and its corresponding hash-based one-time signature scheme.

Table 2. The correspondence between concepts in DAG and those in OTS.

Concept in DAG	Concept in OTS
Sink Vertex r	Public Key
Source Vertices $s_1, \ldots, s_n$	Private Key
Antichain $\mathcal{A}$ of G^*	Message Space / Signature Scheme
MVS $\boldsymbol{c} \in \mathcal{A}$	Signature Pattern
Max Size in $\mathcal{A}$	Maximum Signature Size
Graph Size $\|G\|$	Computational Cost

4.2 From Trees to Chains

In this section, we prove that with regard to the same tree size, the chain structure has the same performance as any tree structure. We prove this result by adapting the technique of Bleichenbacher and Maurer [8] in the binary tree setting to the arbitrary tree structure.

Theorem 5. *Let C_s denote a chain with size s. Let x be the root of a tree T, and $T_1, \ldots, T_n$ be the subtrees of T where $n \geq 1$. Then*

$$C_s^* \cong C_s$$

and

$$T^* \cong T_1^* \times \cdots \times T_n^* \cup \{x\}$$

Proof: It is easy to verify that $C_s^* \cong C_s$. For any $p \in T^*$, if $p \neq \{x\}$ then p can be splited by each subtrees, thus $p \in T_1^* \times \cdots \times T_n^* \cup \{x\}$. If $p = \{x\}$ then $p \in T_1^* \times \cdots \times T_n^* \cup \{x\}$. p can not be $\{x\} \cup S$ for a non-empty set S because p is minimal. The arguments for the other direction is similar. Therefore $T^* \cong T_1^* \times \cdots \times T_n^* \cup \{x\}$.

Theorem 6. *Given a tree $T = (V, E)$ with associated signature scheme $\mathcal{S}$, we can construct another tree T' with associated signature scheme $\mathcal{S}'$, where the root of T' is the only node with indegree greater than 1, and $|\mathcal{S}'| \geq |\mathcal{S}|$.*

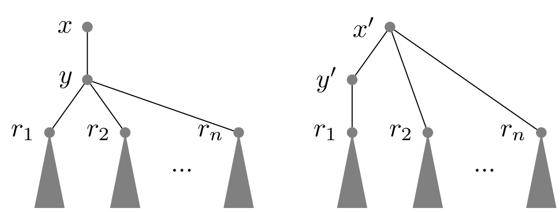

Fig. 2. The conversion process that moves the splitting point further to the top, where triangles denote subtrees.

Proof: Let y be a non-root node in T with indegree greater than 1. Denote x as its parent and $r_1, \ldots, r_n$ as its children for $n > 1$. And $z_1, \ldots, z_m$ be the children of x other than y for $m \geq 0$. We replace the tree $T[x]$ with $T[x']$, where we set the parent of $r_2, \ldots, r_n$ from y to x'. To simplify the expression, let T_r^* be $T^*[r_1] \times \cdots \times T^*[r_n]$ and T_z^* be $T^*[z_1] \times \cdots \times T^*[z_m]$. According to Theorem 5, we have

$$\begin{aligned}
T^*[x] &= T^*[y] \times (T^*[z_1] \times \ldots T^*[z_m]) \cup \{x\} \\
&= (T^*[r_1] \times \cdots \times T^*[r_n] \cup \{y\}) \times (T^*[z_1] \times \cdots \times T^*[z_m]) \cup \{x\} \\
&= (T_r^* \cup \{y\}) \times T_z^* \cup \{x\}
\end{aligned}$$

and

$$\begin{aligned}
T^*[x'] &= (T^*[y'] \times T^*[r_2] \times \cdots \times T^*[r_n]) \times (T^*[z_1] \times \ldots T^*[z_m]) \cup \{x'\} \\
&= ((T^*[r_1] \cup \{y'\}) \times T^*[r_2] \times \cdots \times T^*[r_n]) \times T_z^* \cup \{x'\} \\
&= (T_r^* \cup \{y'\} \times T^*[r_2] \times \cdots \times T^*[r_n]) \times T_z^* \cup \{x'\}
\end{aligned}$$

For any signature pattern $p \in T^*[x]$, if $y \in p$ then we replace y with $y', r_2 \ldots, r_n$ and map p to the resulting $p' \in T^*[x']$; If $p = \{x\}$ then we map p to the $\{x'\} \in T^*[x']$; Otherwise we map p directly to $T[x']$. According to the formulas above, the mapping is injective. Therefore the size of its associated signature scheme $\mathcal{S}'$ is always not less than the size of $\mathcal{S}$.

We repeat this transformation until there is only one node with indegree greater than one (i.e., the root), which completes the proof. ∎

Optimal tree with bounded signature size. The conversion above shows that the chain structure is never worse than any other tree structures of the same tree size. However, this conversion may increase the signature size. Table 3 lists the optimal tree for fixed tree size and signature size, found by brute-force search. All optimal trees listed in the table have the chain structure.

4.3 The Flaw of "The Best Known Graph" Construction

Bleichenbacher and Maurer [7] first proposed "The best known graph" but didn't give an explicit encoding algorithm. Dods et al. [13] presented this construction in detail. We describe the construction, and show that it is not a valid scheme.

The scheme is parameterized by an integer w and an integer B. The scheme consists of a set of B blocks, each block is a matrix of width w and height $w+1$. There is also an additional 0-th block which consists of a single row of w entries. We use the term $z_{b,r,c}$ to refer to the entry in the r-th row and c-th column of the b-th block, where rows and columns are numbered from zero. The entries are assumed to hold values, and they are inferred from the following computational rule:

$$z_{b,r,c} = \begin{cases} \mathsf{H}(z_{b,r-1,c} || z_{b-1,w,(c+r) \bmod w}) & r > 0 \text{ and } b > 1, \\ \mathsf{H}(z_{b,r-1,c} || z_{b-1,0,(c+r) \bmod w}) & r > 0 \text{ and } b = 1, \\ x_{bw+c} & r = 0 \end{cases}$$

Table 3. Optimal trees of small sizes, where the notations follows the conventions in [6]. Here C_s denotes a chain with size s, $[T_1, \dots, T_n]$ denotes a tree constructed by connecting the roots of subtree $T_1, \dots, T_n$ to a new root node. In this case all subtrees are chains.

Tree Size	Upper Bound of Signature Size		
	2	3	4
6	$[C_2, C_3]$	$[C_2, C_3]$	$[C_2, C_3]$
7	$[C_3, C_3]$	$[C_3, C_3]$	$[C_3, C_3]$
8	$[C_3, C_4]$	$[C_2, C_2, C_3]$	$[C_2, C_2, C_3]$
9	$[C_4, C_4]$	$[C_2, C_3, C_3]$	$[C_2, C_3, C_3]$
10	$[C_4, C_5]$	$[C_3, C_3, C_3]$	$[C_3, C_3, C_3]$
11	$[C_5, C_5]$	$[C_3, C_3, C_4]$	$[C_2, C_2, C_3, C_3]$
12	$[C_5, C_6]$	$[C_3, C_4, C_4]$	$[C_2, C_3, C_3, C_3]$
13	$[C_6, C_6]$	$[C_4, C_4, C_4]$	$[C_3, C_3, C_3, C_3]$

To define a signature we first need to define a signature pattern. This is an ordered list of w numbers $p = (r_0, \dots, r_{w-1})$, each $r_i \in \{0, \dots, w\}$, i.e., one row per column. We select the set of patterns $\mathcal{S}$ such that

$$\bigcup_{i \in \{0, \dots, w-1\}} \{i + j \bmod w : r_i \leq j < w\} = \{0, \dots, w-1\}$$

As a toy example, when $w = 2$ the signature space consists 6 choices: $(0,0)$, $(1,0)$, $(2,0)$, $(0,2)$, $(0,1)$, $(1,1)$. We use $\mathcal{S}_i$ to denote the i-th element of $\mathcal{S}$ (e.g. $\mathcal{S}_0 = (0,0), \mathcal{S}_3 = (0,2)$ when $w = 2$), which is also a mapping from $\{0, \dots, |\mathcal{S}|-1\}$ to $\mathcal{S}$. We further define $wt(p) \stackrel{\text{def}}{=} \sum_{i=0}^{w-1} (w - r_i)$ for $p \in \mathcal{S}$. Note that $wt(p) < |\mathcal{S}|$.

The secret key consists of $N = (B+1)w$ values $x_0, \dots, x_{N-1}$ which are placed in the bottom row of each block. The public key is $\mathsf{H}(z_{B,w,0} || \dots || z_{B,w,w-1})$, i.e. the hash of the values in the top row of the last block.

To sign a λ-bit message m, we first represent m in base-$|\mathcal{S}|$ $(m_1, \dots, m_l)$ where $l = \lceil \lambda / \log_2 |\mathcal{S}| \rceil$. Then we compute the checksum in base-$|\mathcal{S}|$: i.e., $c = \sum_{i=1}^{l} wt(\mathcal{S}_{m_i}) = (c_1, \dots, c_{l'})$ where $l' = \lceil 1 + \log_{|\mathcal{S}|} l \rceil$. Let $B = l + l'$, finally we encode $M = (m_1, \dots, m_l, c_1, \dots, c_{l'})$ to this graph which consists of B blocks.

The flaw is that the checksum works on Winternitz type structure but it does not generally work on every structure. We present two message with their checksums respectively $(m, c), (m', c')$ that they are comparable.

Consider the simplest case: $w = 2$ and $l = l' = 1$. For $m = 0, c = 4$ we have $M = (0,0,0,1)$. For $m' = 1, c' = 3$ we have $M' = (1,0,0,2)$. They are comparable. In other words, if the signer signs the first message m, an adversary can easily forge a signature for message m'. We refer to Appendix B for more discussions.

5 Experiments

We replace the OTS component in the SPHINCS$^+$ and XMSS signature schemes with the constant-sum WOTS$^+$, and report the performance improvement.

5.1 Implementation

We adapt the respective official implementations on github [40,37] to ours [44,19], where we reuse most of its basic modules such as hash functions, and implement from scratch only the newly added, i.e., the encoding algorithm. Notice that the latest implementation of SPHINCS$^+$ takes into consideration the flaw in the security reduction [24], and thus the comparison is fair and up-to-date.

We use the SPHINCS$^+$ implementation optimized with architecture-specific instructions such as AESNI or AVX2 [40]. The optimized SPHINCS$^+$ signature is called SPHINCS-α and its details are available in [45]. Since the XMSS team does not provide an official high-performance implementation, we resort to the reference code in [37]. In other words, we choose the best available implementation of the baseline schemes and plug in the constant-sum encoding, without additional engineering optimization.

Instantiation For SPHINCS$^+$, we provide 12 combinations of parameter choices and instantiations. The classic security level includes 128, 192 or 256 bits. The hash functions can be `shake256` [14] or `sha2` [33] (we also use `sha512` to avoid the attack on `sha256` [35]). Following the decisions made by NIST [32], we remove `haraka` [25] and robust version from tweakable hash function. We also offer a small or fast option towards either small signatures or fast signature generation.

For XMSS, we select 41 sets of parameter choices[7] among which the security level can be 192, 256 or 512 bits. The hash function can be either `sha2` series or `shake` series.

Parameter Sets. The parameter sets for SPHINCS$^+$ are re-tuned and listed in Table 4. Note "bitsec" represents classic security level. Readers can also find the parameter estimation code in our open source implementation. Please open `para.ipynb` in Jupyter Notebook with SageMath. The parameters for XMSS are chosen according the the original configuration and we refer the readers to the original publication [23] for the details.

We note that unlike SPHINCS$^+$ which comes with fast and short variants, the IETF documentation of XMSS [23] does not explicitly specify the optimization direction for the XMSS scheme. Instead, it only lists out the parameters for different combinations of tree/hyper-tree depths (which determines the message space), hash functions (SHA2 or SHAKE), and security levels (256 or 512). Therefore, we simply replace WOTS$^+$ with the constant-sum variant and benchmark the performance. In general, we believe that it is possible to re-tune the parameters of XMSS in order to achieve a specific design goal (e.g., to achieve

[7] We omit the parameter sets that lead to extremely high runtime to facilitate fast experiment.

Table 4. Parameter sets for the SPHINCS-α scheme.

Parameter Set	n	h	d	$\log t$	k	w	l	bitsec	sec level	sig bytes
sphincs-a-128s	16	63	9	13	12	73	22	128	I	6880
sphincs-a-128f	16	63	21	8	25	14	36	128	I	16720
sphincs-a-192s	24	63	9	14	17	77	32	192	III	14568
sphincs-a-192f	24	64	16	8	37	8	66	192	III	34896
sphincs-a-256s	32	66	11	13	23	79	42	255	V	27232
sphincs-a-256f	32	68	17	9	35	16	66	255	V	49312

the smallest signature possible while keeping the verification and signing time below a certain threshold) for application-specific scenarios.

Table 5. Performance comparison between SPHINCS$^+$ and SPHINCS-α, with simple tweakable hash function instantiated with shake. Key generation, signing and verification time are in terms of CPU cycles; public key, secret key and signature size are in bytes. All cycle counts are the median of 100 runs.

	SPHINCS$^+$				SPHINCS-α				Relative Change			
Param.	KeyGen	Sign	Verify	Size	KeyGen	Sign	Verify	Size	KeyGen	Sign	Verify	Size
128f	1143558	26872236	2204802	17088	1036602	26635716	2028186	16720	−9.35%	−0.88%	−8.01%	−2.15%
192f	1662498	45405504	3003534	35664	2199276	45218790	1744038	34896	32.29%	−0.41%	−41.93%	−2.15%
256f	4327632	92059542	2967642	49856	4286574	91335474	3175290	49312	−0.95%	−0.79%	7.00%	−1.09%
128s	72597852	551233638	846486	7856	51421086	537033762	2689650	6880	−29.17%	−2.58%	217.74%	−12.42%
192s	105310692	1022229270	1201230	16224	78050718	988899534	3845970	14568	−25.89%	−3.26%	220.17%	−10.21%
256s	69033492	918473904	1701324	29792	52048332	764352612	6005448	27232	−24.60%	−16.78%	252.99%	−8.59%

Environment. We conduct our benchmarks on a Ubuntu 20.04 machine with Ryzen™ 5 3600 CPU and 16GB RAM, compiled with gcc-9.3.0 -O3 -march=native -fomit-frame-pointer -flto.

5.2 Performance

We report the performance of the improved schemes in this subsection. Instances which are optimized using architecture-specific instructions such as AVX2 are marked as avx2 otherwise they are marked as ref.

For SPHINCS$^+$, we show in Table 5 a tiny performance comparison. Table 7 and Table 8 give comprehensive performance summaries for all the parameter sets. As summarized in Table 9, the improved scheme reduces both signing time and signature size for most parameter sets. On the downside, we experience an up to 253% increase in verification time.

In general, we re-tune the parameters towards minimizing signature size (the short variant) or signing time (the fast variant), which showcases advantages over SPHINCS$^+$ of the same security strength. Otherwise said, verification time is not the main factor taken into consideration as it is typically one order of magnitude smaller than the signing time. As a result, the verification time is increased for certain parameter choices. Nevertheless, we argue that for specific scenarios where verification time is critical, we can re-tune the parameters towards fast verification. This is also the reason behind the fluctuation of key generation time in Table 9.

For XMSS, we refer to Table 10 and Table 11 for comprehensive summaries of all parameter sets and to Table 12 for the summarized comparison. The improvement over XMSS is less significant (up to 1.78% saving in signature size) compared to that over SPHINCS$^+$, which may attribute to that we only replaced the encoding scheme without re-tuning the parameters.

References

1. André, D.: Mémoire sur les combinaisons régulières et leurs applications. In: Annales scientifiques de l'École Normale Supérieure. vol. 5, pp. 155–198 (1876)
2. Aumasson, J.P., Endignoux, G.: Improving stateless hash-based signatures. In: Smart, N.P. (ed.) Topics in Cryptology – CT-RSA 2018. Lecture Notes in Computer Science, vol. 10808, pp. 219–242. Springer, Heidelberg, Germany, San Francisco, CA, USA (Apr 16–20, 2018). https://doi.org/10.1007/978-3-319-76953-0_12
3. Belbachir, H., Igueroufa, O.: Congruence properties for Bi^s nomial coefficients and like extended ram and kummer theorems under suitable hypothesis. Mediterranean Journal of Mathematics **17**(1), 1–14 (2020)
4. 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, Part I. Lecture Notes in Computer Science, vol. 9056, pp. 368–397. Springer, Heidelberg, Germany, Sofia, Bulgaria (Apr 26–30, 2015). https://doi.org/10.1007/978-3-662-46800-5_15
5. 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.) ACM CCS 2019: 26th Conference on Computer and Communications Security. pp. 2129–2146. ACM Press (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3363229
6. Bleichenbacher, D., Maurer, U.M.: Directed acyclic graphs, one-way functions and digital signatures. In: Desmedt, Y. (ed.) Advances in Cryptology – CRYPTO'94. Lecture Notes in Computer Science, vol. 839, pp. 75–82. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 21–25, 1994). https://doi.org/10.1007/3-540-48658-5_9
7. Bleichenbacher, D., Maurer, U.M.: On the efficiency of one-time digital signatures. In: Kim, K., Matsumoto, T. (eds.) Advances in Cryptology – ASIACRYPT'96. Lecture Notes in Computer Science, vol. 1163, pp. 145–158. Springer, Heidelberg, Germany, Kyongju, Korea (Nov 3–7, 1996). https://doi.org/10.1007/BFb0034843
8. Bleichenbacher, D., Maurer, U.M.: Optimal tree-based one-time digital signature schemes. In: Annual Symposium on Theoretical Aspects of Computer Science. pp. 361–374. Springer (1996)
9. Bos, J.N., Chaum, D.: Provably unforgeable signatures. In: Brickell, E.F. (ed.) Advances in Cryptology – CRYPTO'92. Lecture Notes in Computer Science, vol. 740,

pp. 1–14. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 1993). https://doi.org/10.1007/3-540-48071-4_1
10. Cooper, D.A., Apon, D.C., Dang, Q.H., Davidson, M.S., Dworkin, M.J., Miller, C.A., et al.: Recommendation for stateful hash-based signature schemes. NIST Special Publication **800**, 208 (2020)
11. Cruz, J.P., Yatani, Y., Kaji, Y.: Constant-sum fingerprinting for winternitz one-time signature. In: 2016 International Symposium on Information Theory and Its Applications (ISITA). pp. 703–707. IEEE (2016)
12. Dilworth, R.P.: A decomposition theorem for partially ordered sets. In: Classic Papers in Combinatorics, pp. 139–144. Springer (2009)
13. Dods, C., Smart, N.P., Stam, M.: Hash based digital signature schemes. In: Smart, N.P. (ed.) 10th IMA International Conference on Cryptography and Coding. Lecture Notes in Computer Science, vol. 3796, pp. 96–115. Springer, Heidelberg, Germany, Cirencester, UK (Dec 19–21, 2005)
14. Dworkin, M.: SHA-3 standard: Permutation-based hash and extendable-output functions (Aug 2015). https://doi.org/https://doi.org/10.6028/NIST.FIPS.202
15. Euler, L.: De evolutione potestatis polynomialis cuiuscunque $(1 + x + x^2 + x^3 + x^4 + \text{etc.})^n$. Nova Acta Academiae Scientiarum Imperialis Petropolitanae pp. 47–57 (1801)
16. Goldreich, O.: Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In: Odlyzko, A.M. (ed.) Advances in Cryptology – CRYPTO'86. Lecture Notes in Computer Science, vol. 263, pp. 104–110. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 1987). https://doi.org/10.1007/3-540-47721-7_8
17. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, Cambridge, UK (2004)
18. Gueron, S., Mouha, N.: SPHINCS-simpira: Fast stateless hash-based signatures with post-quantum security. Cryptology ePrint Archive, Report 2017/645 (2017), `http://eprint.iacr.org/2017/645`
19. Hashbasedsignature: XMSS-i. `https://github.com/hashbasedsignature/xmss-i` (2022)
20. Hashbasedsignature: cswots. `https://github.com/hashbasedsignature/cswots` (2023)
21. Hevia, A., Micciancio, D.: The provable security of graph-based one-time signatures and extensions to algebraic signature schemes. In: Zheng, Y. (ed.) Advances in Cryptology – ASIACRYPT 2002. Lecture Notes in Computer Science, vol. 2501, pp. 379–396. Springer, Heidelberg, Germany, Queenstown, New Zealand (Dec 1–5, 2002). https://doi.org/10.1007/3-540-36178-2_24
22. Hülsing, A.: W-OTS+–shorter signatures for hash-based signature schemes. In: International Conference on Cryptology in Africa. pp. 173–188. Springer (2013)
23. Hülsing, A., Butin, D., Gazdag, S.L., Rijneveld, J., Mohaisen, A.: XMSS: extended merkle signature scheme. In: RFC 8391. IRTF (2018)
24. Hülsing, A., Kudinov, M.: Recovering the tight security proof of SPHINCS$^+$. Cryptology ePrint Archive, Paper 2022/346 (2022), `https://eprint.iacr.org/2022/346`, `https://eprint.iacr.org/2022/346`
25. Kölbl, S., Lauridsen, M.M., Mendel, F., Rechberger, C.: Haraka v2 - Efficient short-input hashing for post-quantum applications. IACR Transactions on Symmetric Cryptology **2016**(2), 1–29 (2016). https://doi.org/10.13154/tosc.v2016.i2.1-29, `http://tosc.iacr.org/index.php/ToSC/article/view/563`
26. Kudinov, M., Hülsing, A., Ronen, E., Yogev, E.: SPHINCS+C: Compressing SPHINCS+ with (almost) no cost. Cryptology ePrint Archive, Paper 2022/778 (2022), `https://eprint.iacr.org/2022/778`

27. Kudinov, M.A., Kiktenko, E.O., Fedorov, A.K.: Security analysis of the W-OTS$^+$ signature scheme: Updating security bounds. Matematicheskie Voprosy Kriptografii [Mathematical Aspects of Cryptography] **12**(2), 129–145 (Jun 2021). https://doi.org/10.4213/mvk362
28. Lamport, L.: Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory (Oct 1979)
29. McGrew, D., Curcio, M., Fluhrer, S.: Leighton-micali hash-based signatures. In: RFC 8554. IRTF (2019)
30. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology – CRYPTO'87. Lecture Notes in Computer Science, vol. 293, pp. 369–378. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 1988). https://doi.org/10.1007/3-540-48184-2_32
31. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology – CRYPTO'89. Lecture Notes in Computer Science, vol. 435, pp. 218–238. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 1990). https://doi.org/10.1007/0-387-34805-0_21
32. Moody, D.: Parameter selection for the selected algorithms (2022), https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/4MBurXr58Rs/m/lj4VRfAnFwAJ
33. National Institute of Standards and Technology: Secure hash standard (2015-08-04 2015). https://doi.org/https://doi.org/10.6028/NIST.FIPS.180-4
34. Perin, L.P., Zambonin, G., Custódio, R., Moura, L., Panario, D.: Improved constant-sum encodings for hash-based signatures. Journal of Cryptographic Engineering **11**, 329–351 (2021)
35. Perlner, R., Kelsey, J., Cooper, D.: Breaking category five SPHINCS+ with SHA-256. Cryptology ePrint Archive, Paper 2022/1061 (2022), https://eprint.iacr.org/2022/1061
36. Rabin, M.O.: Digitalized signatures. Foundations of secure computation pp. 155–168 (1978)
37. Rijneveld, J., Hülsing, A., Cooper, D., Westerbaan, B.: The XMSS reference code. https://github.com/XMSS/xmss-reference (2022)
38. Sperner, E.: Ein satz über untermengen einer endlichen menge. Mathematische Zeitschrift **27**(1), 544–548 (1928)
39. Team, T.N.P.: PQC standardization process: Announcing four candidates to be standardized, plus fourth round candidates. NIST (2022), https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4
40. The SPHINCS+ Team: The SPHINCS+ reference code, accompanying the submission to NIST's post-quantum cryptography project. https://github.com/sphincs/sphincsplus (2021)
41. Vaudenay, S.: One-time identification with low memory. In: Eurocode'92, pp. 217–228. Springer (1993)
42. Warnaar, S.O.: The Andrews-Gordon identities and q-multinomial coefficients. Communications in mathematical physics **184**(1), 203–232 (1997)
43. Zare, D.: How to express $(1+x+x^2+\cdots+x^m)^n$ as a power series? Mathematics Stack Exchange, https://math.stackexchange.com/q/28861, (version: 2011-11-15)
44. Zhang, K.: sphincs-a. https://github.com/kzoacn/sphincs-a (2023)
45. Zhang, K., Cui, H., Yu, Y.: SPHINCS-α: A compact stateless hash-based signature scheme. Cryptology ePrint Archive, Paper 2022/059 (2022), https://eprint.iacr.org/2022/059, https://eprint.iacr.org/2022/059

A An Example of Constant-sum WOTS$^+$

In this section, we present a concrete example of constant-sum WOTS$^+$, including counting, encoding algorithm and the optimality proof. In this example, we choose parameter $l = 3$ and $w = 4$, therefore the size of $\mathcal{C}$ is maximum when the constant-sum is $\lfloor l(w-1)/2 \rfloor = 4$. This example can be also generated from a python code, which is open-sourced at [20].

Counting the size. Recall that

$$D_{l,s} = |\{\boldsymbol{c} \in [w]^l : \sum_{i=1}^{l} c_i = s\}| \ ,$$

with their initial values

$$\begin{aligned} D_{0,0} &= 1, \\ D_{0,s} &= 0, \text{ for } s \in \{1, 2, \ldots, w-1\} \\ D_{l,s} &= 0, \text{ for } 1 \leq l \in \mathbb{Z}, s \in \mathbb{Z}^- \ , \end{aligned}$$

and recurrence relation

$$D_{l,s} = \sum_{i=0}^{w-1} D_{l-1,s-i}, 2 \leq l \in \mathbb{Z}, s \in \{0, 1, \ldots, l(w-1)\} \ .$$

We can compute the table of D, Table 6.

Table 6. The table of D

l \ s	0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0
2	1	2	3	4	3	2	1	0	0	0
3	1	3	6	10	12	12	10	6	3	1

The value of $D_{3,4}$ tells us that we have 12 different vectors such that the length of each is 3 and the sum of each is 4.

The encoding algorithm. Since we $D_{3,4} = 12$, we can encode at most 12 different messages, represented by $\{0, 1, \ldots, 11\}$. We show how to encode $x = 4$ to a constant-sum vector. Recall that for each loop, we determine $v_{l-i} = j$ by seeking which j satisfies $x \in [\sum_{k<j} D_{l-1,s-k}, \sum_{k \leq j} D_{l-1,s-k})$.

– Initialization. Initially, we have $x = 4$ and $s = 4$.

- Loop 1. $D_{l,s} = D_{3,4} = 12 = 2+3+4+3 = D_{2,1} + D_{2,2} + D_{2,3} + D_{2,4}$ and $D_{2,4} \leq x = 4 < D_{2,3} + D_{2,4}$. So we have $j = 1$. Update the variables to $v_1 = 1, s = 3, x = 1$.
- Loop 2. $D_{l,s} = D_{2,3} = 4 = 1+1+1+1 = D_{1,0} + D_{1,1} + D_{1,2} + D_{1,3}$ and $D_{1,3} \leq x = 1 < D_{1,2} + D_{1,3}$. So we have $j = 1$. Update the variables to $v_2 = 1, s = 2, x = 0$.
- Loop 3. $D_{l,s} = D_{1,2} = 1 = 1+0+0 = D_{0,0} + D_{0,1} + D_{0,2}$ and $D_{0,1} + D_{0,2} + \leq x = 0 < D_{0,0} + D_{0,1} + D_{0,2}$. So we have $j = 2$. Update the variables to $v_3 = 2, s = 0, x = 0$.
- Finally. We get $v = (1, 1, 2)$.

By executing the encoding algorithm, We can list those 12 vectors (in order): $\{(0,1,3),(0,2,2),(0,3,1),(1,0,3),(1,1,2),(1,2,1),(1,3,0),(2,0,2),(2,1,1),(2,2,0),(3,0,1),(3,1,0)\}$.

Optimality proof.

By Dilworth's theorem, the proof of optimality is also a construction of chain decomposition. We present an example here, which is computed in the way of the proof of Theorem 4.

- $l = 1$. This is a trivial case. We have only one chain $(0) \leq (1) \leq (2) \leq (3)$.
- $l = 2$. We have 4 chains, they are:
  1. $(0,0) \leq (1,0) \leq (2,0) \leq (3,0) \leq (3,1) \leq (3,2) \leq (3,3)$.
  2. $(0,1) \leq (1,1) \leq (2,1) \leq (2,2) \leq (2,3)$.
  3. $(0,2) \leq (1,2) \leq (1,3)$.
  4. $(0,3)$.
- $l = 3$. We have 12 chains, they are:
  1. $(0,0,0) \leq (1,0,0) \leq (2,0,0) \leq (3,0,0) \leq (3,1,0) \leq (3,2,0) \leq (3,3,0) \leq (3,3,1) \leq (3,3,2) \leq (3,3,3)$
  2. $(0,0,1) \leq (1,0,1) \leq (2,0,1) \leq (3,0,1) \leq (3,1,1) \leq (3,2,1) \leq (3,2,2) \leq (3,2,3)$
  3. $(0,0,2) \leq (1,0,2) \leq (2,0,2) \leq (3,0,2) \leq (3,1,2) \leq (3,1,3)$
  4. $(0,0,3) \leq (1,0,3) \leq (2,0,3) \leq (3,0,3)$
  5. $(0,1,0) \leq (1,1,0) \leq (2,1,0) \leq (2,2,0) \leq (2,3,0) \leq (2,3,1) \leq (2,3,2) \leq (2,3,3)$
  6. $(0,1,1) \leq (1,1,1) \leq (2,1,1) \leq (2,2,1) \leq (2,2,2) \leq (2,2,3)$
  7. $(0,1,2) \leq (1,1,2) \leq (2,1,2) \leq (2,1,3)$
  8. $(0,1,3) \leq (1,1,3)$
  9. $(0,2,0) \leq (1,2,0) \leq (1,3,0) \leq (1,3,1) \leq (1,3,2) \leq (1,3,3)$
  10. $(0,2,1) \leq (1,2,1) \leq (1,2,2) \leq (1,2,3)$
  11. $(0,2,2) \leq (0,2,3)$
  12. $(0,3,0) \leq (0,3,1) \leq (0,3,2) \leq (0,3,3)$

The size of this chain decomposition meets the size of antichain $\mathcal{C}$. According to Dilworth's theorem, the antichain $\mathcal{C}$ is maximum.

B On The Best Known Graph

We first correct a minor fault in the design of the weight function of [13]. The old weight function was $wt(p) = \sum_{i=0}^{w-1}(w + 1 - r_i)$. Since we know $r_i \in \{0, \ldots, w\}$, the range of the old weight function is $[w, w(w+1)]$. When $w = 2$, this range can not be fitted into $\{0, \ldots, |\mathcal{S}|-1\}$. Thus we make it into $wt(p) = \sum_{i=0}^{w-1}(w - r_i) \in [0, w^2]$, which is a more suitable choice.

For $l = l' = 1, w = 2$, we can have a correct construction if we reorder the mapping $\{\mathcal{S}_i\}$ to $(0, 0), (1, 0), (0, 1), (1, 1), (2, 0), (0, 2)$. ([13] does not specific the order that mapping integer to signature pattern.) This does not mean we fixed this construction. The key problem is we can not prove the pairs of the message and checksum (m, c) form an antichain in this graph. There may exist forgery attacks for larger l, l', w parameters.

There is a way to fix it by using "separate representation function encoding", purposed in [7], which can be viewed as a generalized checksum method. However, even if we use this new encoding, the performance of this graph-based is clearly worst than WOTS$^+$ (with checksum). Both two constructions require encoding checksum separately. For $w = 3$, the WOTS$^+$ fully utilized $(w+1)^w = 64$ message space while the graph-based has only $|\mathcal{S}| = 51$ choices.

C More Detailed Comparisons

C.1 Comparison Between Original and Improved SPHINCS$^+$

We benchmarked the performance of the improved SPHINCS$^+$ under 24 parameter settings ($\{\text{shake256}, \text{sha256}\} \times \{128, 192, 256\} \times \{\text{fast}, \text{small}\} \times \{\text{ref}, \text{avx2}\}$). To facilitate a fair comparison, we tested our implementation (adapted from the SPHINCS$^+$ codes) along with the original SPHINCS$^+$. The test results are reported in Table 7 and Table 8 with a comparison in Table 9.

Table 7. Runtime benchmarks for SPHINCS$^+$. Key generation, signing and verification time are in the number of cpu cycles; public key, secret key and signature size are in bytes. All cycle counts are the median of 100 runs.

Parameter Set	Impl.	KeyGen	Sign	Verify	Pk	Sk	Sig
sphincs-shake-128f	ref	7622514	178188408	10775124	32	64	17088
sphincs-shake-192f	ref	11240172	290022120	15972588	48	96	35664
sphincs-shake-256f	ref	29488050	593083386	15949980	64	128	49856
sphincs-shake-128s	ref	493648758	3747092580	3602178	32	64	7856
sphincs-shake-192s	ref	717515010	6427813662	5332932	48	96	16224
sphincs-shake-256s	ref	470748762	5584718124	7709508	64	128	29792
sphincs-sha2-128f	ref	4600566	107749800	6402438	32	64	17088
sphincs-sha2-192f	ref	6705198	181354752	9365400	48	96	35664
sphincs-sha2-256f	ref	17695728	362443014	9947394	64	128	49856
sphincs-sha2-128s	ref	294665274	2237140404	2282346	32	64	7856
sphincs-sha2-192s	ref	428811300	3954195432	3266478	48	96	16224
sphincs-sha2-256s	ref	283132530	3503794590	4785678	64	128	29792
sphincs-shake-128f	avx2	2494854	58500990	4063716	32	64	17088
sphincs-shake-192f	avx2	3541392	91863954	5919426	48	96	35664
sphincs-shake-256f	avx2	9676188	193273884	6019830	64	128	49856
sphincs-shake-128s	avx2	159844320	1210947264	1491894	32	64	7856
sphincs-shake-192s	avx2	233254134	2093058036	2165706	48	96	16224
sphincs-shake-256s	avx2	153274212	1799699922	3085776	64	128	29792
sphincs-sha2-128f	avx2	1143558	26872236	2204802	32	64	17088
sphincs-sha2-192f	avx2	1662498	45405504	3003534	48	96	35664
sphincs-sha2-256f	avx2	4327632	92059542	2967642	64	128	49856
sphincs-sha2-128s	avx2	72597852	551233638	846486	32	64	7856
sphincs-sha2-192s	avx2	105310692	1022229270	1201230	48	96	16224
sphincs-sha2-256s	avx2	69033492	918473904	1701324	64	128	29792

Table 8. Runtime benchmarks for SPHINCS-α. Key generation, signing and verification time are in the number of cpu cycles; public key, secret key and signature size are in bytes. All cycle counts are the median of 100 runs.

Parameter Set	Impl.	KeyGen	Sign	Verify	Pk	Sk	Sig
sphincs-a-shake-128f	ref	6861114	176440590	9035874	32	64	16720
sphincs-a-shake-192f	ref	14555628	281397294	7353450	48	96	34896
sphincs-a-shake-256f	ref	29112588	586596492	15740802	64	128	49312
sphincs-a-shake-128s	ref	347407200	3628303722	12591234	32	64	6880
sphincs-a-shake-192s	ref	533064942	6209945190	19292382	48	96	14568
sphincs-a-shake-256s	ref	362125278	4942646316	31932360	64	128	27232
sphincs-a-sha2-128f	ref	4157334	106933014	5569452	32	64	16720
sphincs-a-sha2-192f	ref	8837622	173603070	4654278	48	96	34896
sphincs-a-sha2-256f	ref	17439858	357693966	9646776	64	128	49312
sphincs-a-sha2-128s	ref	208068264	2172320100	7584102	32	64	6880
sphincs-a-sha2-192s	ref	319426722	3827118258	11723508	48	96	14568
sphincs-a-sha2-256s	ref	215034228	3011033142	19147662	64	128	27232
sphincs-a-shake-128f	avx2	2218014	57069090	3558492	32	64	16720
sphincs-a-shake-192f	avx2	4614804	92073114	3028500	48	96	34896
sphincs-a-shake-256f	avx2	9563742	191187306	5983920	64	128	49312
sphincs-a-shake-128s	avx2	108983646	1139743980	4891482	32	64	6880
sphincs-a-shake-192s	avx2	171004500	1996754616	7254738	48	96	14568
sphincs-a-shake-256s	avx2	115604604	1582371720	11677806	64	128	27232
sphincs-a-sha2-128f	avx2	1036602	26635716	2028186	32	64	16720
sphincs-a-sha2-192f	avx2	2199276	45218790	1744038	48	96	34896
sphincs-a-sha2-256f	avx2	4286574	91335474	3175290	64	128	49312
sphincs-a-sha2-128s	avx2	51421086	537033762	2689650	32	64	6880
sphincs-a-sha2-192s	avx2	78050718	988899534	3845970	48	96	14568
sphincs-a-sha2-256s	avx2	52048332	764352612	6005448	64	128	27232

Table 9. Performance comparison between the original and improved SPHINCS$^+$ in terms of relative changes.

Parameter Set			Runtime			
SPHINCS$^+$	SPHINCS-α	Impl.	KeyGen	Sign	Verify	Sig Size
sphincs-shake-128f	sphincs-a-shake-128f	ref	−9.99%	−0.98%	−16.14%	−2.15%
sphincs-shake-192f	sphincs-a-shake-192f	ref	29.50%	−2.97%	−53.96%	−2.15%
sphincs-shake-256f	sphincs-a-shake-256f	ref	−1.27%	−1.09%	−1.31%	−1.09%
sphincs-shake-128s	sphincs-a-shake-128s	ref	−29.62%	−3.17%	249.55%	−12.42%
sphincs-shake-192s	sphincs-a-shake-192s	ref	−25.71%	−3.39%	261.76%	−10.21%
sphincs-shake-256s	sphincs-a-shake-256s	ref	−23.07%	−11.50%	314.19%	−8.59%
sphincs-sha2-128f	sphincs-a-sha2-128f	ref	−9.63%	−0.76%	−13.01%	−2.15%
sphincs-sha2-192f	sphincs-a-sha2-192f	ref	31.80%	−4.27%	−50.30%	−2.15%
sphincs-sha2-256f	sphincs-a-sha2-256f	ref	−1.45%	−1.31%	−3.02%	−1.09%
sphincs-sha2-128s	sphincs-a-sha2-128s	ref	−29.39%	−2.90%	232.29%	−12.42%
sphincs-sha2-192s	sphincs-a-sha2-192s	ref	−25.51%	−3.21%	258.90%	−10.21%
sphincs-sha2-256s	sphincs-a-sha2-256s	ref	−24.05%	−14.06%	300.10%	−8.59%
sphincs-shake-128f	sphincs-a-shake-128f	avx2	−11.10%	−2.45%	−12.43%	−2.15%
sphincs-shake-192f	sphincs-a-shake-192f	avx2	30.31%	0.23%	−48.84%	−2.15%
sphincs-shake-256f	sphincs-a-shake-256f	avx2	−1.16%	−1.08%	−0.60%	−1.09%
sphincs-shake-128s	sphincs-a-shake-128s	avx2	−31.82%	−5.88%	227.87%	−12.42%
sphincs-shake-192s	sphincs-a-shake-192s	avx2	−26.69%	−4.60%	234.98%	−10.21%
sphincs-shake-256s	sphincs-a-shake-256s	avx2	−24.58%	−12.08%	278.44%	−8.59%
sphincs-sha2-128f	sphincs-a-sha2-128f	avx2	−9.35%	−0.88%	−8.01%	−2.15%
sphincs-sha2-192f	sphincs-a-sha2-192f	avx2	32.29%	−0.41%	−41.93%	−2.15%
sphincs-sha2-256f	sphincs-a-sha2-256f	avx2	−0.95%	−0.79%	7.00%	−1.09%
sphincs-sha2-128s	sphincs-a-sha2-128s	avx2	−29.17%	−2.58%	217.74%	−12.42%
sphincs-sha2-192s	sphincs-a-sha2-192s	avx2	−25.89%	−3.26%	220.17%	−10.21%
sphincs-sha2-256s	sphincs-a-sha2-256s	avx2	−24.60%	−16.78%	252.99%	−8.59%

C.2 Comparison Between Original and Improved XMSS

We benchmarked the performance of the improved XMSS under selected parameter settings. To facilitate a fair comparison, we tested our implementation (adapted from the official repository) along with the original XMSS. The test results are reported in Table 10 and Table 11 with a comparison in Table 12.

Table 10. Runtime benchmarks for XMSSMT. Key generation, signing and verification time are in the number of cpu cycles; public key, secret key and signature size are in bytes. All cycle counts are the median of 16 runs.

	Runtime			Size		
Parameter Set	KeyGen	Sign	Verify	Pk	Sk	Sig
XMSSMT-SHA2-20/2-256	3127413888	3771702	1617156	64	5998	4963
XMSSMT-SHA2-20/4-256	222559560	6706008	3110238	64	10938	9251
XMSSMT-SHA2-40/4-256	6285411684	6830658	3247326	64	15252	9893
XMSSMT-SHA2-40/8-256	401766552	6904440	6711552	64	24516	18469
XMSSMT-SHA2-60/6-256	9494855496	10171836	5168160	64	24507	14824
XMSSMT-SHA2-60/12-256	631228644	6877116	9580554	64	38095	27688
XMSSMT-SHA2-20/2-512	21988477260	27152100	11814210	128	15822	18115
XMSSMT-SHA2-20/4-512	1398460644	47020626	22096512	128	33818	34883
XMSSMT-SHA2-40/4-512	43629570864	47908404	23528916	128	42164	36165
XMSSMT-SHA2-40/8-512	2805197148	47601774	45550674	128	76964	69701
XMSSMT-SHA2-60/6-512	65939229972	69142698	33890688	128	68507	54216
XMSSMT-SHA2-60/12-512	4155322500	47779668	68537412	128	120111	104520
XMSSMT-SHA2-20/2-192	2020361292	2466378	1176246	48	4182	2955
XMSSMT-SHA2-20/4-192	127543572	4318398	2133288	48	7138	5403
XMSSMT-SHA2-40/4-192	4053119040	4406868	2126394	48	10444	5885
XMSSMT-SHA2-40/8-192	256662108	4352274	4225824	48	15884	10781
XMSSMT-SHA2-60/6-192	6067543248	6375420	3298608	48	16707	8816
XMSSMT-SHA2-60/12-192	378984168	4349250	6459930	48	24631	16160
XMSSMT-SHAKE-20/2-256	11248052760	13700142	6175242	64	5998	4963
XMSSMT-SHAKE-20/4-256	710060580	24423426	11387034	64	10938	9251
XMSSMT-SHAKE-40/4-256	22458447756	24651684	11682468	64	15252	9893
XMSSMT-SHAKE-40/8-256	1439101980	24385518	23653764	64	24516	18469
XMSSMT-SHAKE-60/6-256	33696560700	35529246	16996320	64	24507	14824
XMSSMT-SHAKE-60/12-256	2150833356	24420024	36412686	64	38095	27688
XMSSMT-SHAKE-20/4-512	2455477200	83672478	40536054	128	33818	34883
XMSSMT-SHAKE-40/4-512	76825886112	84051144	40027680	128	42164	36165
XMSSMT-SHAKE-40/8-512	4869152712	83529522	78300342	128	76964	69701
XMSSMT-SHAKE-60/6-512	115072275744	121304394	60009516	128	68507	54216
XMSSMT-SHAKE-60/12-512	7279843032	83360394	120177378	128	120111	104520
XMSSMT-SHAKE256-20/2-256	10871025984	13265334	5549670	64	5998	4963
XMSSMT-SHAKE256-20/4-256	717916932	23495616	11077614	64	10938	9251
XMSSMT-SHAKE256-40/4-256	21817454832	23909310	12181104	64	15252	9893
XMSSMT-SHAKE256-40/8-256	1402673508	23652882	22820472	64	24516	18469
XMSSMT-SHAKE256-60/6-256	32735511720	34534476	18080442	64	24507	14824
XMSSMT-SHAKE256-60/12-256	2086527672	23456430	34476750	64	38095	27688
XMSSMT-SHAKE256-20/2-192	8045426736	9687330	4610826	48	4182	2955
XMSSMT-SHAKE256-20/4-192	527327388	17429292	8847306	48	7138	5403
XMSSMT-SHAKE256-40/4-192	16079381892	17519976	8559144	48	10444	5885
XMSSMT-SHAKE256-40/8-192	1035980136	17371080	17699076	48	15884	10781
XMSSMT-SHAKE256-60/6-192	24215925168	25526448	13124520	48	16707	8816
XMSSMT-SHAKE256-60/12-192	1525123404	17327790	25423218	48	24631	16160

Table 11. Runtime benchmarks for improved XMSSMT. Key generation, signing and verification time are in the number of cpu cycles; public key, secret key and signature size are in bytes. All cycle counts are the median of 16 runs.

	Runtime			Size		
Parameter Set	KeyGen	Sign	Verify	Pk	Sk	Sig
XMSSMT-i-SHA2-20/2-256	3118500216	3768984	1676682	64	5966	4899
XMSSMT-i-SHA2-20/4-256	205928460	6701778	3302856	64	10842	9123
XMSSMT-i-SHA2-40/4-256	6111330408	6775578	3367674	64	15156	9765
XMSSMT-i-SHA2-40/8-256	421704720	6636150	6578640	64	24292	18213
XMSSMT-i-SHA2-60/6-256	9212217048	9834372	5043978	64	24347	14632
XMSSMT-i-SHA2-60/12-256	590852376	6655554	9848970	64	37743	27304
XMSSMT-i-SHA2-20/2-512	21771015228	26223552	11456892	128	15758	17987
XMSSMT-i-SHA2-20/4-512	1444733100	47859624	22947750	128	33626	34627
XMSSMT-i-SHA2-40/4-512	43337753064	47542536	23114718	128	41972	35909
XMSSMT-i-SHA2-40/8-512	2891694348	47495754	46440648	128	76516	69189
XMSSMT-i-SHA2-60/6-512	65241573480	68704668	34608996	128	68187	53832
XMSSMT-i-SHA2-60/12-512	4197559644	47451924	69522930	128	119407	103752
XMSSMT-i-SHA2-20/2-192	1997951040	2467044	1099980	48	4158	2907
XMSSMT-i-SHA2-20/4-192	133595676	4259340	2137932	48	7066	5307
XMSSMT-i-SHA2-40/4-192	3993020136	4302990	2186424	48	10372	5789
XMSSMT-i-SHA2-40/8-192	270959364	4279050	4293126	48	15716	10589
XMSSMT-i-SHA2-60/6-192	5922093888	6206382	3251826	48	16587	8672
XMSSMT-i-SHA2-60/12-192	388450260	4318524	6467760	48	24367	15872
XMSSMT-i-SHAKE-20/2-256	10926481032	13278870	5770854	64	5966	4899
XMSSMT-i-SHAKE-20/4-256	722327868	23823738	11518020	64	10842	9123
XMSSMT-i-SHAKE-40/4-256	21800594664	23788656	11308122	64	15156	9765
XMSSMT-i-SHAKE-40/8-256	1422925776	23784876	22552668	64	24292	18213
XMSSMT-i-SHAKE-60/6-256	32771966400	34440912	17060670	64	24347	14632
XMSSMT-i-SHAKE-60/12-256	2132075484	23932764	33975306	64	37743	27304
XMSSMT-i-SHAKE-20/4-512	2481508404	83815794	39808368	128	33626	34627
XMSSMT-i-SHAKE-40/4-512	75490496244	82866564	39879108	128	41972	35909
XMSSMT-i-SHAKE-40/8-512	4944876840	83403126	79286958	128	76516	69189
XMSSMT-i-SHAKE-60/6-512	113727752028	119970360	59860440	128	68187	53832
XMSSMT-i-SHAKE-60/12-512	7349705244	82872720	119115792	128	119407	103752
XMSSMT-i-SHAKE256-20/2-256	10433094588	12529494	5531094	64	5966	4899
XMSSMT-i-SHAKE256-20/4-256	698620392	22806234	11090754	64	10842	9123
XMSSMT-i-SHAKE256-40/4-256	20830958172	22584744	11067732	64	15156	9765
XMSSMT-i-SHAKE256-40/8-256	1357966188	22758804	22020138	64	24292	18213
XMSSMT-i-SHAKE256-60/6-256	31048091712	32707962	16657614	64	24347	14632
XMSSMT-i-SHAKE256-60/12-256	2019728664	22604778	32743098	64	37743	27304
XMSSMT-i-SHAKE256-20/2-192	7701230628	9301482	4125564	48	4158	2907
XMSSMT-i-SHAKE256-20/4-192	513443304	16590348	8076150	48	7066	5307
XMSSMT-i-SHAKE256-40/4-192	15321653784	16670448	8214084	48	10372	5789
XMSSMT-i-SHAKE256-40/8-192	979483644	16741926	16256556	48	15716	10589
XMSSMT-i-SHAKE256-60/6-192	22849024932	24038622	12288024	48	16587	8672
XMSSMT-i-SHAKE256-60/12-192	1470316644	16589088	24196410	48	24367	15872

Table 12. Performance comparison between the original and improved XMSS in terms of relative changes.

Parameter Set		Runtime			
Original	Improved	KeyGen	Sign	Verify	Sig Size
XMSSMT-SHA2-20/2-256	XMSSMT-i-SHA2-20/2-256	−0.29%	−1.29%	3.68%	−1.29%
XMSSMT-SHA2-20/4-256	XMSSMT-i-SHA2-20/4-256	−7.47%	−1.38%	6.19%	−1.38%
XMSSMT-SHA2-40/4-256	XMSSMT-i-SHA2-40/4-256	−2.77%	−1.29%	3.71%	−1.29%
XMSSMT-SHA2-40/8-256	XMSSMT-i-SHA2-40/8-256	4.96%	−1.39%	−1.98%	−1.39%
XMSSMT-SHA2-60/6-256	XMSSMT-i-SHA2-60/6-256	−2.98%	−1.30%	−2.40%	−1.30%
XMSSMT-SHA2-60/12-256	XMSSMT-i-SHA2-60/12-256	−6.40%	−1.39%	2.80%	−1.39%
XMSSMT-SHA2-20/2-512	XMSSMT-i-SHA2-20/2-512	−0.99%	−0.71%	−3.02%	−0.71%
XMSSMT-SHA2-20/4-512	XMSSMT-i-SHA2-20/4-512	3.31%	−0.73%	3.85%	−0.73%
XMSSMT-SHA2-40/4-512	XMSSMT-i-SHA2-40/4-512	−0.67%	−0.71%	−1.76%	−0.71%
XMSSMT-SHA2-40/8-512	XMSSMT-i-SHA2-40/8-512	3.08%	−0.73%	1.95%	−0.73%
XMSSMT-SHA2-60/6-512	XMSSMT-i-SHA2-60/6-512	−1.06%	−0.71%	2.12%	−0.71%
XMSSMT-SHA2-60/12-512	XMSSMT-i-SHA2-60/12-512	1.02%	−0.73%	1.44%	−0.73%
XMSSMT-SHA2-20/2-192	XMSSMT-i-SHA2-20/2-192	−1.11%	−1.62%	−6.48%	−1.62%
XMSSMT-SHA2-20/4-192	XMSSMT-i-SHA2-20/4-192	4.75%	−1.78%	0.22%	−1.78%
XMSSMT-SHA2-40/4-192	XMSSMT-i-SHA2-40/4-192	−1.48%	−1.63%	2.82%	−1.63%
XMSSMT-SHA2-40/8-192	XMSSMT-i-SHA2-40/8-192	5.57%	−1.78%	1.59%	−1.78%
XMSSMT-SHA2-60/6-192	XMSSMT-i-SHA2-60/6-192	−2.40%	−1.63%	−1.42%	−1.63%
XMSSMT-SHA2-60/12-192	XMSSMT-i-SHA2-60/12-192	2.50%	−1.78%	0.12%	−1.78%
XMSSMT-SHAKE-20/2-256	XMSSMT-i-SHAKE-20/2-256	−2.86%	−1.29%	−6.55%	−1.29%
XMSSMT-SHAKE-20/4-256	XMSSMT-i-SHAKE-20/4-256	1.73%	−1.38%	1.15%	−1.38%
XMSSMT-SHAKE-40/4-256	XMSSMT-i-SHAKE-40/4-256	−2.93%	−1.29%	−3.20%	−1.29%
XMSSMT-SHAKE-40/8-256	XMSSMT-i-SHAKE-40/8-256	−1.12%	−1.39%	−4.66%	−1.39%
XMSSMT-SHAKE-60/6-256	XMSSMT-i-SHAKE-60/6-256	−2.74%	−1.30%	0.38%	−1.30%
XMSSMT-SHAKE-60/12-256	XMSSMT-i-SHAKE-60/12-256	−0.87%	−1.39%	−6.69%	−1.39%
XMSSMT-SHAKE-20/4-512	XMSSMT-i-SHAKE-20/4-512	1.06%	−0.73%	−1.80%	−0.73%
XMSSMT-SHAKE-40/4-512	XMSSMT-i-SHAKE-40/4-512	−1.74%	−0.71%	−0.37%	−0.71%
XMSSMT-SHAKE-40/8-512	XMSSMT-i-SHAKE-40/8-512	1.56%	−0.73%	1.26%	−0.73%
XMSSMT-SHAKE-60/6-512	XMSSMT-i-SHAKE-60/6-512	−1.17%	−0.71%	−0.25%	−0.71%
XMSSMT-SHAKE-60/12-512	XMSSMT-i-SHAKE-60/12-512	0.96%	−0.73%	−0.88%	−0.73%
XMSSMT-SHAKE256-20/2-256	XMSSMT-i-SHAKE256-20/2-256	−4.03%	−1.29%	−0.33%	−1.29%
XMSSMT-SHAKE256-20/4-256	XMSSMT-i-SHAKE256-20/4-256	−2.69%	−1.38%	0.12%	−1.38%
XMSSMT-SHAKE256-40/4-256	XMSSMT-i-SHAKE256-40/4-256	−4.52%	−1.29%	−9.14%	−1.29%
XMSSMT-SHAKE256-40/8-256	XMSSMT-i-SHAKE256-40/8-256	−3.19%	−1.39%	−3.51%	−1.39%
XMSSMT-SHAKE256-60/6-256	XMSSMT-i-SHAKE256-60/6-256	−5.15%	−1.30%	−7.87%	−1.30%
XMSSMT-SHAKE256-60/12-256	XMSSMT-i-SHAKE256-60/12-256	−3.20%	−1.39%	−5.03%	−1.39%
XMSSMT-SHAKE256-20/2-192	XMSSMT-i-SHAKE256-20/2-192	−4.28%	−1.62%	−10.52%	−1.62%
XMSSMT-SHAKE256-20/4-192	XMSSMT-i-SHAKE256-20/4-192	−2.63%	−1.78%	−8.72%	−1.78%
XMSSMT-SHAKE256-40/4-192	XMSSMT-i-SHAKE256-40/4-192	−4.71%	−1.63%	−4.03%	−1.63%
XMSSMT-SHAKE256-40/8-192	XMSSMT-i-SHAKE256-40/8-192	−5.45%	−1.78%	−8.15%	−1.78%
XMSSMT-SHAKE256-60/6-192	XMSSMT-i-SHAKE256-60/6-192	−5.64%	−1.63%	−6.37%	−1.63%
XMSSMT-SHAKE256-60/12-192	XMSSMT-i-SHAKE256-60/12-192	−3.59%	−1.78%	−4.83%	−1.78%