Breaking Category Five SPHINCS$^+$ with SHA-256

Ray Perlner[1], John Kelsey[1,2], and David Cooper[1]

[1] National Institute of Standards and Technology, Gaithersburg, Maryland 20899, USA
[2] COSIC/KU Leuven

Abstract. SPHINCS$^+$ is a stateless hash-based signature scheme that has been selected for standardization as part of the NIST post-quantum cryptography (PQC) standardization process. Its security proof relies on the distinct-function multi-target second-preimage resistance (DM-SPR) of the underlying keyed hash function. The SPHINCS$^+$ submission offered several instantiations of this keyed hash function, including one based on SHA-256. A recent observation by Sydney Antonov on the PQC mailing list demonstrated that the construction based on SHA-256 did not have DM-SPR at NIST category five, for several of the parameter sets submitted to NIST; however, it remained an open question whether this observation leads to a forgery attack. We answer this question in the affirmative by giving a complete forgery attack that reduces the concrete classical security of these parameter sets by approximately 40 bits of security.
Our attack works by applying Antonov's technique to the WOTS$^+$ public keys in SPHINCS$^+$, leading to a new one-time key that can sign a very limited set of hash values. From that key, we construct a slightly altered version of the original hypertree with which we can sign arbitrary messages, yielding signatures that appear valid.

Keywords: hash-based signatures · post-quantum cryptography · SPHINCS$^+$.

1 Introduction

SPHINCS$^+$ [2] is a stateless hash-based signature scheme that has been selected for standardization as part of the NIST post-quantum cryptography standardization process [16]. Much of the underlying technology for SPHINCS$^+$ goes back to the very earliest days of academic cryptography [13–15]. Its security is based entirely on the security of symmetric cryptographic primitives.

Because of the age of the underlying technology and the lack of additional hardness assumptions (most other post-quantum algorithms depend on the difficulty of problems such as finding short vectors in a lattice or solving systems of multivariate quadratic equations), SPHINCS$^+$ appears to provide extremely reliable security, albeit at the cost of larger and slower signatures than most other post-quantum signature schemes.

Recently, however, Sydney Antonov described a failure of a particular property (the DM-SPR property[3]) claimed by the SPHINCS$^+$ designers when SHA-256 is the hash function used [1]. It was not clear, however, whether this observation led to an attack on the full SPHINCS$^+$ signature scheme.

In this paper, we describe such an attack. Specifically, we extend Antonov's observation to a forgery attack on both of the recommended parameter sets for SPHINCS$^+$ that claim category five [17] security (256 bits of classical security) and use the SHA-256 hash function. Our attack becomes even more powerful for other choices of SPHINCS$^+$ parameters that use SHA-256 while claiming category five security, specifically for smaller values of w than are used in the recommended parameter sets.

Our attack allows forgery of an unlimited number of signatures of the attacker's choice. While our attack is far too expensive to pose a real-world security threat, it demonstrates a failure of SPHINCS$^+$ to meet its claimed security goals for the category five parameter set. Table 1 gives the results of our attack for the two category five parameter sets in [2], assuming a SPHINCS$^+$ key that has been used to sign its maximum allowed number of signatures (2^{64}).

Table 1. Summary of Our Results on SPHINCS$^+$ Category Five Parameters

Parameter Set	Cost				Reference
	Herd	Link	Signable	Total	
SPHINCS$^+$-256f	$2^{214.8}$	$2^{216.4}$	$2^{215.7}$	$\approx 2^{217.4}$	Section 4.3
SPHINCS$^+$-256s	$2^{214.8}$	$2^{216.4}$	$2^{215.7}$	$\approx 2^{217.4}$	Section 4.3

Both Antonov's approach and our extension of it are partly based on properties of Merkle-Damgård hash functions first described in [9, 11, 12] (notably, these attacks would not work if the hash function were replaced with a random oracle), but also incorporate details of the internal structure of SPHINCS$^+$. Earlier [19], another security issue with SPHINCS$^+$ level five parameters was noted, again due to the use of SHA-256 to provide 256 bits of security.

These results do not seem to us to indicate any fundamental weakness in SPHINCS$^+$. Instead, they demonstrate that using a 256-bit Merkle-Damgård hash like SHA-256 to get more than 128 bits of security is quite difficult. If SHA-512 were used in place of SHA-256 for category five security in SPHINCS$^+$, all of these observations and attacks would be entirely blocked. Similarly, when SPHINCS$^+$ uses SHAKE256 to get category five security, none of these attacks are possible. Very recently [8], the SPHINCS$^+$ team has proposed a tweak which appears to block these attacks. A discussion of the proposed tweaks appears in Section 6.

The rest of the paper is organized as follows: We begin by describing SPHINCS$^+$ (Section 2). We then introduce some tools, concepts, and notation that will be

[3] For a formal definition of this property, see Section 3.2.

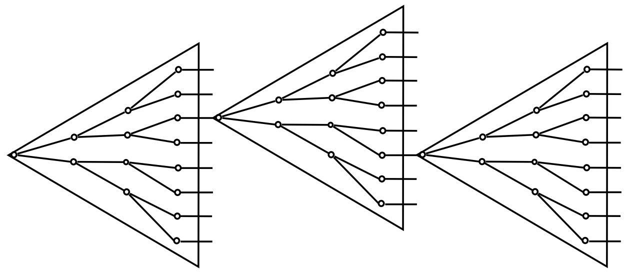

Fig. 1. The SPHINCS$^+$ hypertree.

used in the rest of the paper (Section 3). We then describe our attack (Section 4). Next, we justify the costs we assigned to each step of our attack, and some possible optimizations (Section 5). Finally, we conclude the paper with a discussion of what can be done to prevent this kind of attack, and where else the attack or variants of it may apply (Section 6).

2 The SPHINCS$^+$ Signature Scheme

The SPHINCS$^+$ signature scheme consists of a few components: a one-time signature scheme, WOTS$^+$ (a specific variant of Winternitz signatures defined in [7]); a few-time signature scheme, FORS (Forest of Random Subsets); and Merkle trees [14]. SPHINCS$^+$ forms the WOTS$^+$ public keys into a hypertree, or tree of trees (see Figure 1). Each tree is a Merkle tree in which the leaves are WOTS$^+$ public keys. The root of each tree is signed by a WOTS$^+$ key from a tree at the next level up, and the root of the top-level tree is the SPHINCS$^+$ public key. The WOTS$^+$ keys from the lowest-level tree are used to sign FORS public keys, and the FORS keys are in turn used to sign messages.

SPHINCS$^+$ uses randomized hashing. When a message is to be signed, a random bit string, R, is generated and is hashed along with the message. Some bits from the hash value are then used to select a FORS key (which selects a path through the hypertree); the rest are signed using that FORS key. A SPHINCS$^+$ signature then consists of R, the signature created using the selected FORS key, the sequence of WOTS$^+$ signatures in the hypertree leading from the top-level tree to the FORS public key used to sign the message, and the authentication paths corresponding to each WOTS$^+$ signature needed to compute the roots of each of the Merkle trees.

In this paper, we apply a multi-target preimage attack (described in Section 3) to the WOTS$^+$ public keys that are used to sign the roots of Merkle trees and FORS public keys. The WOTS$^+$ public keys are computed as shown in Figure 2. For the parameter sets in [2] that target category five security, which use a hash function with a 256-bit output and a Winternitz parameter, w, of 16, the public key is the hash of a public seed, PK.seed, which is padded to 64 bytes, a 22-byte compressed address, and 67 public values. Computing the

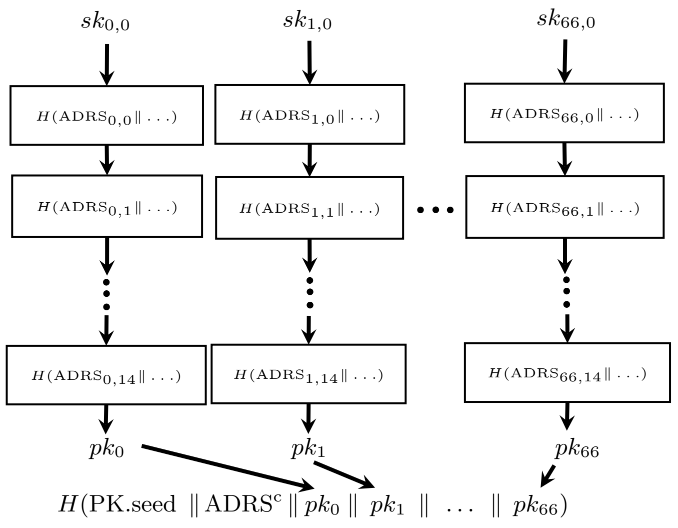
$sk_{0,0}$
$sk_{1,0}$
$sk_{66,0}$
$H(\text{ADRS}_{0,0} \| \ldots)$
$H(\text{ADRS}_{1,0} \| \ldots)$
$H(\text{ADRS}_{66,0} \| \ldots)$
$H(\text{ADRS}_{0,1} \| \ldots)$
$H(\text{ADRS}_{1,1} \| \ldots)$
$H(\text{ADRS}_{66,1} \| \ldots)$
$H(\text{ADRS}_{0,14} \| \ldots)$
$H(\text{ADRS}_{1,14} \| \ldots)$
$H(\text{ADRS}_{66,14} \| \ldots)$
pk_0
pk_1
pk_{66}
$H(\text{PK.seed} \,\|\, \text{ADRS}^c \,\|\, pk_0 \,\|\, pk_1 \,\|\, \ldots \,\|\, pk_{66})$

Fig. 2. A WOTS$^+$ public key.

public key hash requires 35 iterations of the SHA-256 compression function, as shown in Figure 7. Each public value is computed by generating a hash chain, which involves iterating a different secret value through the hash function 15 times. Each call to the hash function includes an unique address as input, which identifies the tree layer in which the WOTS$^+$ key appears along with the key's index within that layer. For the computations of the public values, the address also identifies which of the 67 public values is being computed as well as the iteration of the hash function. It is these addresses that were intended to prevent multi-target attacks.

A WOTS$^+$ signature consists of one entry from each of the 67 hash chains of the WOTS$^+$ one-time key.[4] The 256-bit hash of the value to be signed is written as 64 hexadecimal digits, and each is signed using a different hash chain. For example, if the first hexadecimal digit is `0`, then $sk_{0,0}$ is the signature for the first digit. If the second digit is `f`, then pk_1 is the signature for that digit. If the third digit is `3`, then the signature value for the third digit is $sk_{2,3}$ – the result of iterating $sk_{2,0}$ through the hash function three times.

A WOTS$^+$ signature is verified by completing the computation of each of the hash chains. In the example from the previous paragraph, the signature on the first digit (`0`) is checked by iterating the signature value through the hash function 15 times and comparing the result to pk_0. The second digit (`f`) is checked by simply comparing the signature value to pk_1. The third digit (`3`) is checked by iterating the signature value through the hash function 12 times

[4] This description is accurate for the recommended parameters for SPHINCS$^+$ at category five security; other choices of parameters would require the description to be slightly changed.

and comparing the result to pk_2. Note that when the digit signed is an `f`, the verifier does no additional hashing of the value from the signature. Unlike digits `0-e`, the verifier's calculation on an `f` digit does not incorporate the value of the one-time key's `ADRS`.

The final three hash chains are used to sign a three-hexadecimal-digit checksum value. The checksum value is computed by summing the digits of the 256-bit hash value and then subtracting the result from the maximum possible sum, 960. Including the checksum in the signature prevents an attacker from modifying the signature without performing a preimage attack on the hash function.

The SPHINCS$^+$ submission [2] defines two parameter sets that target category five security: SPHINCS$^+$-256s, which contains eight levels of trees, each with a height of eight; and SPHINCS$^+$-256f, which contains 17 levels of trees, each with a height of four. So, the SPHINCS$^+$-256s parameter set includes a little over 2^{64} WOTS$^+$ keys and the SPHINCS$^+$-256f parameter set includes a little over 2^{68} WOTS$^+$ keys.

3 Building Blocks

3.1 Merkle-Damgård Hash Functions

A Merkle-Damgård hash function is constructed from a fixed-length hash function called a *compression function*. In the case of SHA-256, the compression function takes a chaining value of 256 bits and a message block of 512 bits. In order to process a message, the message is unambiguously padded to an integer multiple of 512 bits (the padding incorporates the length of the unpadded message), broken into a sequence of 512-bit message blocks $M_{0,1,\ldots,L-1}$, and processed sequentially, starting from a constant initial chaining value. Thus, to hash the above sequence of message blocks, we compute

$$\begin{aligned} H_{-1} &= \text{initial chaining value} \\ H_j &= \texttt{COMPRESS}(H_{j-1}, M_j) \end{aligned}$$

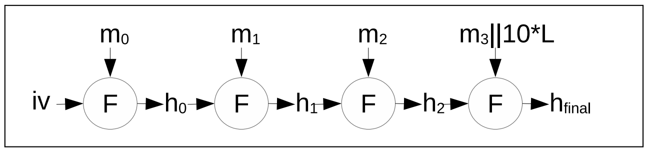

Fig. 3. Merkle-Damgård hashing

After each 512-bit message block, the state of the hash is reduced to a 256-bit chaining value, and this chaining value is the *only information* about the message processed so far that is carried forward into the hash computation. A

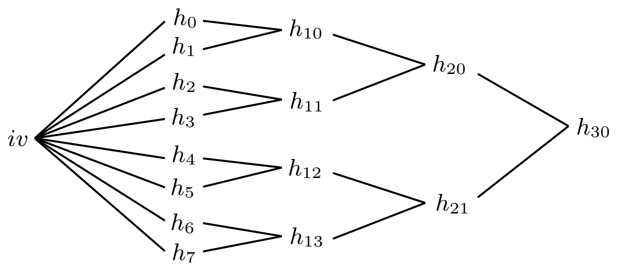

Fig. 4. A diamond structure – constructing eight messages with the same hash

consequence of this fact is we can choose the beginning of two different messages so that their chaining value h_1 collides, and then we can get a new collision by appending any sequence of message blocks to both messages. As noted in [9], this process can be repeated, constructing many different messages with the same hash value.

Diamond Structures Applying the techniques from [11], we can extend this attack, finding these internal hash chain collisions in a tree structure. Starting from 2^k different initial message blocks, we find subsequent message blocks that map them all to a single hash chaining value. This structure of collisions is called a *diamond structure*, and is illustrated in Figure 4. In the figure, lines represent message blocks, labels represent intermediate hash values; each path from iv to h_{30} gives a sequence of four message blocks, and all eight possible sequences of four message blocks yield the same hash value. Constructing the diamond structure requires a sequence of batched collision searches, resulting in a distinct message block for each line in the diagram.

3.2 Multi-target Preimage Attacks and SPHINCS$^+$

Consider an attacker asked to find a preimage – that is, a message that hashes to a single target value, T. If SHA-256 behaves randomly, this should require about 2^{256} trial hashes to accomplish – the attacker can simply hash random messages until one gives the right result. Now consider an attacker asked to find a message that hashes to any one of 2^{64} different hash values, $T_{0,1,\ldots,2^{64}-1}$. Again, if SHA-256 behaves randomly, this should require only about 2^{192} trial hashes – the attacker hashes random messages until one matches *any one* of the values in the target list. Intuitively, the attacker has many targets, so is more likely to hit one. The situation where the attacker tries to find a message that hashes to any one of many targets is called a *multi-target preimage attack.*

Consider the same attacker, given 2^{64} different target hash values, but each target hash value is associated with a different prefix and the preimage is only valid if it starts with the correct prefix. The straightforward multi-target preimage attack no longer works. The attacker must start each message with a particular prefix to get a valid preimage, and if the message hashes to a target associated with a different prefix, it isn't valid.

The SPHINCS$^+$ specification formalizes the above defense against multi-target preimage attacks by (in the "simple" SHA-256 parameter sets) treating the SHA-256 hash of a message, prepended with each prefix, as a separate member in a hash function family. SPHINCS$^+$ also includes "robust" parameter sets, where the hash function members process the input not just by prefixing a constant, but also by XORing the input message with a constant deterministically generated pseudorandom keystream. Our attack applies to both schemes, but for simplicity, we will describe the attack only in terms of the "simple" parameter sets.

Definition 1. ***PQ-DM-SPR*** *(definition 8 from [5]) Let* $H : \mathcal{K} \times \{0,1\}^\alpha \rightarrow \{0,1\}^n$ *be a keyed hash function. We define the advantage of any adversary* $\mathcal{A} = (\mathcal{A}_1), \mathcal{A}_2$ *against distinct-function, multi-target second-preimage resistance (DM-SPR). This definition is parameterized by the number of targets* p.

$$\begin{aligned} Succ^{DM\text{-}SPR}(\mathcal{A}) = \Big[&\{K_i\}_{i=1}^p \leftarrow \mathcal{A}_1(), \{M_i\}_1^p \leftarrow_R (\{0,1\}^\alpha)^p; \\ &(j, M') \leftarrow_R \mathcal{A}_2(\{(K_i, M_i)\}_{i-1}^p) : M' \neq M_j \\ &\wedge H(K_j, M_j) = H(K_j, M') \wedge \boldsymbol{DIST}(\{K_i\}_{i=1}^p) \Big]. \end{aligned}$$

where we assume that $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *share state and define the predicate* $\boldsymbol{DIST}(\{K_i\}_{i=1}^p) = (\forall i, k \in [1,p], i \neq k) : K_i \neq K_k$.

3.3 Antonov's Attack on DM-SPR

In 2022, Sydney Antonov described an attack against the DM-SPR property of the SHA-256-based keyed hash functions used in SPHINCS$^+$ [1]. The attack takes advantage of SHA-256's Merkle-Damgård construction, using a series of collision attacks against the underlying compression function to transform a distinct-function multi-target second-preimage attack into a single-function multi-target second-preimage attack, using several of the techniques described in Section 3.

Figure 5 shows an example of how the attack works.

Suppose there are six target messages, $\mathrm{M}_0 \ldots \mathrm{M}_5$, each hashed using a different key, $\mathrm{ADRS}_0 \ldots \mathrm{ADRS}_5$. The attack begins building a diamond structure, as shown in Figure 4, above. The SHA-256 compression function is applied to the addresses for each of the targets using the SHA-256 initialization vector.[5] This results in a set of intermediate hash values, $H_0^{(1)} \ldots H_5^{(1)}$. Then collision attacks are performed on pairs of intermediate hash values. For example, the attacker searches for random values x_0 and x_1 such that $\mathrm{C}(\mathrm{H}_0^{(1)}, x_0) = \mathrm{C}(\mathrm{H}_1^{(1)}, x_1)$. The

[5] For simplicity, the example assumes that ADRS_i is exactly 512-bits in length, but this is not a requirement for the attack to work.

$$\begin{tikzcd}
H_0^{(1)} = \mathrm{C}(\mathrm{IV}, \mathrm{ADRS}_0) \arrow[r] & H_{0,1}^{(2)} = \mathrm{C}(H_0^{(1)}, x_0) = \mathrm{C}(H_1^{(1)}, x_1) \arrow[ddddddl, bend left] \\
H_1^{(1)} = \mathrm{C}(\mathrm{IV}, \mathrm{ADRS}_1) \arrow[ur] & \\
H_2^{(1)} = \mathrm{C}(\mathrm{IV}, \mathrm{ADRS}_2) \arrow[r] & H_{2,3}^{(2)} = \mathrm{C}(H_2^{(1)}, x_2) = \mathrm{C}(H_3^{(1)}, x_3) \arrow[ddddl, bend left] \\
H_3^{(1)} = \mathrm{C}(\mathrm{IV}, \mathrm{ADRS}_3) \arrow[ur] & \\
H_4^{(1)} = \mathrm{C}(\mathrm{IV}, \mathrm{ADRS}_4) \arrow[r] & H_{4,5}^{(2)} = \mathrm{C}(H_4^{(1)}, x_4) = \mathrm{C}(H_5^{(1)}, x_5) \arrow[ddl, bend left] \\
H_5^{(1)} = \mathrm{C}(\mathrm{IV}, \mathrm{ADRS}_5) \arrow[ur] & \\
H_{0\ldots5}^{(3)} = \mathrm{C}(H_{0,1}^{(2)}, x_{0,1}) = \mathrm{C}(H_{2,3}^{(2)}, x_{2,3}) = \mathrm{C}(H_{4,5}^{(2)}, x_{4,5}) \arrow[d] & \\
\mathrm{C}(H_{0\ldots5}^{(3)}, z \parallel \text{padding}) = \text{SHA-256}(\mathrm{ADRS}_i \parallel \mathrm{M}_i) &
\end{tikzcd}$$

Fig. 5. A multi-target second-preimage attack on a SHA-256-based keyed hash function.

search yields message blocks x_0, x_1 and the intermediate hash, $H_{0,1}^{(2)}$. A second iteration of collision attacks is then performed using the intermediate hash values generated from the first iteration. In this case, there are three intermediate hash values, $H_{0,1}^{(2)}$, $H_{2,3}^{(2)}$, and $H_{4,5}^{(2)}$, and a three-way collision can be found such that $H_{0\ldots5}^{(3)} = \mathrm{C}(H_{0,1}^{(2)}, x_{0,1}) = \mathrm{C}(H_{2,3}^{(2)}, x_{2,3}) = \mathrm{C}(H_{4,5}^{(2)}, x_{4,5})$, for some $H_{0\ldots5}^{(3)}$, $x_{0,1}$, $x_{2,3}$, and $x_{4,5}$. (If there were more targets, then more iterations of collision attacks could be performed until all of the targets had been herded[6] to a single intermediate hash value.)

Next, the attack carries out a multi-target preimage attack, as discussed in Section 3. The attacker searches for some message block z such that

$$\mathrm{C}(H_{0\ldots5}^{(3)}, z \| \text{padding}) = \text{SHA-256}(\mathrm{ADRS}_i, \mathrm{M}_i)$$

for some $i \in \{0, \ldots, 5\}$. If, for example, the final step finds a second preimage for M_3, then $\text{SHA-256}(\mathrm{ADRS}_3 \| x_3 \| x_{2,3} \| z) = \text{SHA-256}(\mathrm{ADRS}_3 \| \mathrm{M}_3)$.

Given t target messages, the expected cost for the final step in the attack, finding a preimage, is $2^{256}/t$ calls to the compression function, C. The preceding steps require performing $\tilde{O}(t)$ collision attacks. The expected cost of each collision attack will depend on whether a 2-way, 3-way, 4-way, etc. collision is sought. In general, the expected cost of finding an n-way collision is $\tilde{O}(2^{256(n-1)/n})$ calls to the compression function.

For a generic second-preimage attack, the attack cost is optimized by using $(t-1)$ 2-way collisions to herd t targets down to 1 intermediate hash value. However, when attacking SPHINCS$^+$, messages have fixed lengths, which limits

[6] The combination of building a diamond structure and finding a linking message is referred to as a herding attack in [11].

the number of targets that may be used in the attack. Using some 3- and 4-way collisions increases the cost of the collisions-finding step, but increases the number of targets that may be used, which reduces the overall cost of the attack.

4 Creating Forgeries for SPHINCS$^+$ Category Five Parameters

4.1 Turning Antonov's Attack Into a Forgery Attack

Suppose the target of Antonov's attack is a set of WOTS$^+$ public keys within the same SPHINCS$^+$ hypertree. After the owner of the key has signed many messages, the attacker has many choices of one-time public key to choose from. Each one-time public key that was used to produce a signature can be computed from the corresponding message and signature, and so is known to the attacker. Further, the attacker can reconstruct the exact hash computation (every 512-bit message block and 256-bit intermediate chaining value) that appeared in the hash computation for each one-time public key used. That hash is computed by hashing some values that are constant for a given SPHINCS$^+$ key, followed by a unique $\texttt{ADRS}^C$ that is guaranteed to be different for every one-time key used, followed by a sequence of 67 hash values. In turn, each of those hash values is the last entry in a hash chain of length 16, and each of the hash computations in that chain is also done with a unique $\texttt{ADRS}^C$ value linked to the $\texttt{ADRS}^C$ of the public key, as shown in Figure 2.

SPHINCS$^+$ uses a huge number of these one-time public keys – after 2^{64} messages are signed, we expect to be able to find more than 2^{62} such hashes, each a plausible target.[7]

In this case, we can apply Antonov's attack to construct a set of many WOTS$^+$ keys with different $\texttt{ADRS}^C$ values, herd them down to a single chaining value, and then carry out a multi-target preimage attack against the original keys' hashes. The result is a *chimera* – a new one-time public key with the $\texttt{ADRS}^C$ value of one of the target messages, many hash chains at the beginning which the attacker has generated anew, followed by some hash chains at the end from an existing key.

Original:	PK.seed $\texttt{ADRS}^C$	$PK_0\ PK_1\ PK_2\ \ldots\ PK_{62}\ PK_{63}$	$PK_{64}\ PK_{65}\ PK_{66}$
Chimera:	PK.seed $\texttt{ADRS}^C$	$PK^*_0\ PK^*_1\ PK^*_2\ \ldots\ PK^*_{62}\ PK^*_{63}$	$PK_{64}\ PK_{65}\ PK_{66}$
	same for both keys	*new values*	*same for both keys*

Fig. 6. Original key and chimera key

[7] In this paper we assume that the attacker only chooses target WOTS$^+$ keys from a single SPHINCS$^+$ key. However, an attacker may choose target WOTS$^+$ keys from multiple SPHINCS$^+$ keys, in which case a successful attack would result in the ability to forge messages for one of the targeted SPHINCS$^+$ keys.

$$\begin{aligned}
&H^{(1)} = C(\mathrm{IV}_{\text{SHA-256}}, \text{PK.seed}\|\text{toByte}(0, 64-n))\\
&H^{(2)} = C(H^{(1)}, \mathrm{ADRS}^{c}_{i,j}\|pk_{i,j,0}\|pk_{i,j,1}[0\dots 9])\\
&H^{(3)} = C(H^{(2)}, pk_{i,j,1}[10\dots 31]\|pk_{i,j,2}\|pk_{i,j,3}[0\dots 9])\\
&H^{(4)} = C(H^{(3)}, pk_{i,j,3}[10\dots 31]\|pk_{i,j,4}\|pk_{i,j,5}[0\dots 9])\\
&H^{(5)} = C(H^{(4)}, pk_{i,j,5}[10\dots 31]\|pk_{i,j,6}\|pk_{i,j,7}[0\dots 9])\\
&H^{(6)} = C(H^{(5)}, pk_{i,j,7}[10\dots 31]\|pk_{i,j,8}\|pk_{i,j,9}[0\dots 9])\\
&\dots\\
&H^{(32)} = C(H^{(31)}, pk_{i,j,59}[10\dots 31]\|pk_{i,j,60}\|pk_{i,j,61}[0\dots 9])\\
&H^{(33)} = C(H^{(32)}, pk_{i,j,61}[10\dots 31]\|pk_{i,j,62}\|pk_{i,j,63}[0\dots 9])\\
&H^{(34)} = C(H^{(33)}, pk_{i,j,63}[10\dots 31]\|pk_{i,j,64}\|pk_{i,j,65}[0\dots 9])\\
&H^{(35)} = C(H^{(34)}, pk_{i,j,65}[10\dots 31]\|pk_{i,j,66}\|\text{padding})\\
&\text{SHA-256}(\text{PK.seed}, \mathrm{ADRS}_{i,j}, pk) = H^{(35)}
\end{aligned}$$

Fig. 7. Computing the SHA-256 hash for a WOTS$^+$ public key.

The chimera key contains the same beginning (PK.seed and `ADRS`^{C}) as the original key, and also the same final three hash values. But the rest of the hash values in the key are newly produced by the attacker, and critically, the chimera key has the same SHA-256 hash as the original key. To be more specific, the values of $PK^*_{0\dots 62}$, along with the first nine bytes of PK^*_{63}, must be chosen so that the intermediate hash of the chimera key after processing those bytes is identical to that of the original key after processing $PK_{0\dots 62}$ and the first nine bytes of PK_{63}. (In Figure 7, this intermediate hash value is $H^{(33)}$.)

Constructing such a chimera key whose hash matches one of the WOTS$^+$ keys in the SPHINCS$^+$ hypertree is a major part of our attack, but several more steps are needed to get a complete forgery attack. Unfortunately, the chimera key we get from Antonov's attack cannot yet be used to create valid WOTS$^+$ signatures.

The problem is as follows: in order to use the diamond structure, many different starting `ADRS`^{C} values might be associated with the same chimera key. We will not know which `ADRS`^{C} value should be used until we have found the linking message, that is, found a choice of `ADRS`^{C}, $pk_{0\dots 62}, pk_{63}[0\dots 9]$ whose intermediate hash is the same as the value of $H^{(33)}$ for one of the target WOTS$^+$ keys. Because the verifier will re-derive $pk_{0\dots 66}$ by iterated hashing of the elements of the signature, and will incorporate `ADRS`^{C} into those hash computations, the value of `ADRS`^{C} is bound to the hash chains. With very high probability, the linking message will be to a different `ADRS`^{C} than the one used to compute the hash chains used for the herding step, and so the resulting chimera key won't work.

This leads to a key insight of our attack: Recall that signing a message with WOTS$^+$ starts by writing the hash of the message as a hexadecimal number. Consider the ith digit of the hash. If the digit is any value *except* `f`, the verifier must use the correct `ADRS`^{C} to derive the value of pk_i. The value of pk_i is bound to a single value of `ADRS`^{C} in this case. But when the ith digit of the hash is `f`, the verifier does not have to do any hashing operation the corresponding element of the signature, and so `ADRS`^{C} is not incorporated. Thus, when we construct the chimera key, the first X hash chain values can be chosen to allow signing of any digit; the next $62 - X$ chains can be used only to sign the digit `f`, and the last three chains will encode the checksum. This allows us to construct a chimera key that can be used to sign at least some hashes.

4.2 Summary of Our Attack

The full attack thus happens in multiple phases.

1. **Choose 2^k target keys.** Select a set of WOTS$^+$ keys from the SPHINCS$^+$ hypertree to target in the attack. We do this by examining the set of one-time signature keys that have appeared in at least one SPHINCS$^+$ signature. Each such one-time key will have been used to sign either a Merkle tree root or a FORS public key. We select a set of 2^k target keys whose signatures had acceptably low checksum values, for reasons explained below.
2. **Generate 2^k candidate keys.** Each candidate key starts from the $\texttt{ADRS}^C$ of one of the 2^k target keys; one of these 2^k $\texttt{ADRS}^C$ values will appear in our final chimera key. For each candidate key, we generate X new hash chains using the new key's $\texttt{ADRS}^C$, ensuring that we can sign any possible hex digit in the first X digits of the hash with this key.
3. **Build a diamond structure mapping all the new one-time keys to the same hash chaining value.** To do this, we start with 2^k distinct chaining values (one for each candidate key), and carry out a set of collision attacks to reduce the number to 2^{k-2}, then 2^{k-4}, and so on until we get down to a single hash chaining value.[8] Each 512-bit message block reduces our number of hash chains by a factor of four. In order to do the herding, we select random values for the ends of the next Y hash chains. Note that we only know the end value of these hash chains, and so when this key is used, the corresponding digits of the hash can only be signed if they are `f`, but they can be used in this way with *any* $\texttt{ADRS}^C$. The result of this step is a single hash chaining value (internal to SHA-256) that is reached by all 2^k of our new candidate keys.
4. **Find a linking message.** From that single hash chaining value, select Z random values for the end of hash chains (filling in the values of one 512-bit message block for SHA-256), so that the resulting hash chaining value collides with a hash chaining value at the right position in one of our original target messages. This is a multitarget preimage attack, and requires about 2^{256-k} hash operations. Steps 1-4 are illustrated in Figure 8. In the diagram, $P1 \ldots P8$ are used as shorthand for the PK.seed and $\texttt{ADRS}^C$ of eight different target keys; $t_{0\ldots7}$ are the intermediate hash values of the target keys just before the checksum chains.
5. **Construct the chimera key.** We now have a *chimera* key–a WOTS$^+$ key whose first X hash chains are newly generated (and can be used to sign any

[8] This assumes we search for 4-collisions at each step; optimizing the attack can vary this–see Section 5.

hash digit), while its next $Y+Z$ chains are random (and can be used to sign only an `f` hash digit), and the last three (used to encode the checksum) are original to the key whose $\texttt{ADRS}^C$ our chimera key has taken. The chimera key has the same $\texttt{ADRS}^C$ and the same hash as that original key. The chimera key produced consists of the following components:

(a) PK.seed
(b) The $\texttt{ADRS}^C$ of the target key – the one that will be replaced by the chimera.
(c) X hash values from newly-generated hash chains with the correct $\texttt{ADRS}^C$. These will allow us to sign any value in the first X digits of the hash.
(d) $Y+Z$ randomly-generated hash values. These will allow us to sign the next $Y+Z$ digits of the hash only if those digits are all `f`.
(e) Three hash values from the key that will be replaced by the chimera. These encode the checksum from the signature produced by the original key. Because of the properties of WOTS$^+$, we can *increase* any digit of the checksum and get a valid signature, but we cannot *decrease* any digit of the checksum.

6. **Sign a Merkle tree root or FORS key with the chimera key.** Given the chimera key, we can sign with it. While an ordinary WOTS$^+$ key can sign *any* hash, the chimera key can only sign *a small subset of hashes*–ones with `f` digits in each of the $Y+Z$ random chains' positions, with the sum of the free digits small enough to yield either the same checksum as the one that appeared in the key's original signature, or a checksum that can be reached by incrementing the original checksum's digits. We must do a large brute-force search to find a Merkle tree root full of one-time keys or a FORS key whose hash this chimera key can sign. However, note that this need only be done once, to allow an arbitrary number of forged messages to be produced.
7. **Forge a signature.** With the chimera key and its signature computed, we now brute-force search for randomized messages until we find one whose hypertree path (determined by the idx value) includes the location of the original key, whose hash is the same as that of our chimera key. (This will take less than 2^{68} work.) Once we find such a message, we can use the new one-time key or FORS key we signed with our chimera key to construct a valid SPHINCS$^+$ signature on the message.

Steps 3, 4, and 6 are each computationally very expensive. However, they are done sequentially. The total cost of the attack is the sum of the costs of building the diamond structure (step 3), finding the linking message (step 4), and constructing a Merkle tree root or FORS key whose hash the chimera key can sign (step 6).

Step 6 can only succeed for a very limited set of hash values. In this step, we must create a Merkle tree root or FORS key whose hash value, written as a hexadecimal string, follows the pattern:

```
xxxxxxxx xxxxxxxx xxxxxxxf ffffffff
ffffffff ffffffff ffffffff ffffffff
```

where an `x` may be any digit, but an `f` must be a hex digit `f`. Let C be the checksum of the original key that was replaced by the chimera key, and S be the sum of the digits of the hash. Along with following the above pattern, we must find a hash value for which we can sign a valid checksum. Since 41 of the hex digits must be `f`, the lowest possible value of the checksum is 0x159.

Each digit in the checksum can be increased but not decreased, to reach our goal. If $C > 960 - S$, then the chimera key cannot be used to construct *any* valid signature. For this reason, we choose candidate keys based on the checksum they produced when they signed. In general, we want the lowest checksums possible. The probability of a random WOTS$^+$ signature having an acceptable checksum (for example, 0x140-143) is about 2^{-18}, so with many WOTS$^+$ keys that have been used to create signatures to choose from (around 2^{63} after all possible signatures have been made with a given SPHINCS$^+$ public key), we can always find a large set ($> 2^{40}$) of target messages.

Finding a hash that can be signed by the chimera key is accomplished by a brute force attack–we simply try many inputs to the hash until we get one that can be signed. Depending on the location in the hypertree of the original key that is to be replaced by the chimera key, we will either have to sign a root of a Merkle tree or a FORS key. In either case, we can generate many candidate values relatively efficiently by keeping most of the tree or key fixed and only altering a single leaf in the tree (or leaf in the last tree of the FORS key).

Once we have successfully signed a single Merkle tree or FORS key with the chimera key, we can use the Merkle tree's WOTS$^+$ keys or the FORS key to sign arbitrary messages, as many as we like. Since the chimera key has the same hash as the target WOTS$^+$ key it has replaced, new SPHINCS$^+$ signatures can be constructed, substituting the chimera key and the new Merkle tree or FORS key, but otherwise just like previous signatures with the same key.

4.3 Overview of the Forgery Attack on SPHINCS$^+$-SHA-256 with Category Five Parameters

In this section, we describe the full forgery attack against the SPHINCS$^+$-SHA-256-256f-simple parameter set from [2]. The basic idea behind the attack also applies to the other category five SHA-256-based parameter sets from [2] (including the 'robust' parameter sets), and would also apply to category five SHA-256-based parameter sets that used a Winternitz parameter, w, other than 16. The attacks follow the same basic outline, but some of the details differ.

In [1], Sydney Antonov's goal was simply to find a message of the same length as a WOTS$^+$ public key ($32 \cdot 67 = 2144$ bytes) that would hash to the same value as the WOTS$^+$ public key when using the same prefix (PK.seed and $\texttt{ADRS}^C$). In order to extend this into an forgery attack against SPHINCS$^+$, we must construct a chimera key which can be used to generate a valid signature for at least some hash values.

In order to do this, our attack takes advantage of a detail of WOTS$^+$ signatures: The one-time public key is computed by hashing together the final value in each of the 67 hash chains used in the signature. The signature contains 67

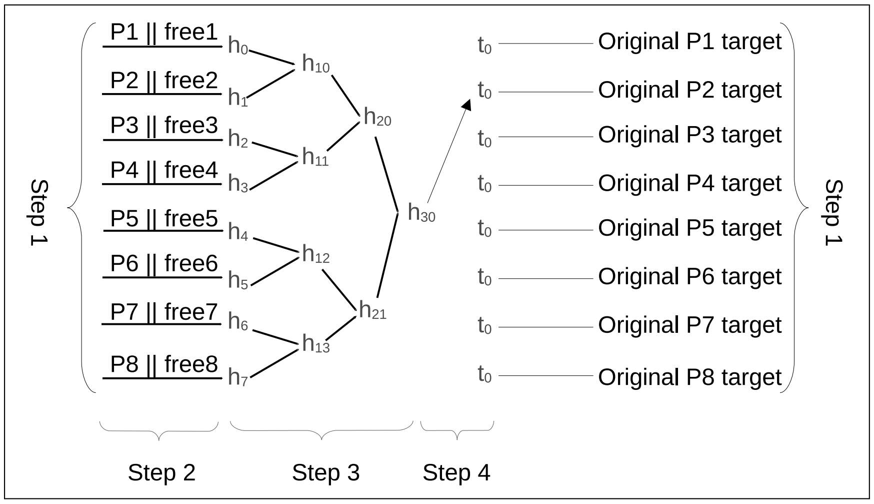

Fig. 8. Steps 1-4 of the attack

elements, each an entry in one of the hash chains, and the verifier must derive the final entry in each hash chain from these.

When a given signature element is signing a hex digit of $0 \dots e$, computing the final element in that chain requires hashing the signature element, and that hash incorporates the correct value of ADRS. But when the signature element signs a hex digit of f, the verifier simply uses the provided element as the final entry in that chain.

This means that the verifier's processing of that signature element will be identical, regardless of the ADRS.

Step 1 in the attack is to choose a set of $t = 3 \cdot 2^{38} \approx 2^{39.58}$ targets. In order to be able to sign a Merkle tree root, each digit in the checksum for that root must be at least as large as in the original signature. So, targets with small checksums need to be chosen. For this attack, $t = 3 \cdot 2^{38}$ targets are chosen that have a checksum of 319 or less. An attacker that has collected about 2^{58} WOTS$^+$ signatures should have access to $3 \cdot 2^{38}$ signatures with checksums of this form.

Step 2 is to create the starting points for the multi-target second-preimage attack. For each of the t target keys ($0 \leq i < 3 \cdot 2^{38}$), we create a new string which starts with the PK.seed and ADRS^C value from the target key, and then construct 22 hash chains using the ADRS^C value and other metadata, so that we will be able to sign arbitrary digits with these hash chains. We compute 22 secret values $(sk_i[0], \dots, sk_i[21])$, and iterate through the hash chains to compute the corresponding public values $(pk_i[0], \dots, pk_i[21])$. We then compute the twelfth intermediate hash value, $H_i^{(12)}$.

Step 3 is to herd these $3 \cdot 2^{38}$ targets down to a single intermediate hash chaining value. The process begins by creating 2^{38} 3-way collisions for the thirteenth intermediate hash value, $H^{(13)}$. When creating an input value for the function,

f, values are chosen for $sk[22]$ and the first ten bytes of $pk[23]$. Performing 2^{38} 3-way collision searches would require about 2^{214} calls to the compression function, but this cost can be dramatically reduced (to the approximate equivalent of 2^{196} compression function calls accounting for memory costs) by performing a batched multi-target multi-collision search as described in Section 5.4.

The herding process is completed by performing 19 rounds of 4-way collision searches, resulting in all $3 \cdot 2^{38}$ targets having the same value for $H_i^{(32)}$. For each round, the input value for the function, f, is an arbitrary 512-bit value. For the corresponding portions of the WOTS$^+$ key $(pk[23], \ldots, pk[61])$ only the public value will be known, and so only a digit with a value of 0xf may be signed. The 19 rounds would require about 2^{230} calls to the compression function, if $2^{36.42}$ separate 4-way collision searches were performed, but the cost may be significantly reduced by performing batched multi-target multi-collision searches. We estimate the complexity of these searches in Equation 1 in Section 5.4. Accounting for memory costs as well as computation, the combined cost is approximately equivalent to $2^{214.8}$ compression function calls.

Step 4 is to find a message block that links with an intermediate chaining value in the right position in one of the target messages. Finding a message block that collides with one of these $3 \cdot 2^{38}$ target vales for $H_i^{(33)}$ should require approximately $2^{256-39.58} = 2^{216.42}$ calls to the compression function.

Step 5 is to construct our chimera key, as discussed above.

Step 6 is to construct a Merkle tree or FORS key that can be signed by the chimera key.

An initial Merkle tree is generated by creating 16 WOTS$^+$ keys and then computing the root of the Merkle tree for those keys. Additional Merkle tree roots may be created by changing just the final public value for one of the 16 WOTS$^+$ keys. Computing the new WOTS$^+$ key and updating the Merkle tree would require 20 calls to the compression function.[9][10] New Merkle tree roots are created until one is generated that has the form

```
xxxxxxxx xxxxxxxx xxxxxxxf ffffffff
ffffffff ffffffff ffffffff ffffffff
```

and that has a checksum that can be signed using the chimera key. More than $\binom{26+23-1}{26} \approx 2^{44.64}$ Merkle tree roots that have this form, so finding one should require less than $20 \cdot 2^{256-44.64} = 2^{215.68}$ calls to the compression function.

The final step is to create a message whose signature makes use of the forged Merkle tree or FORS key. Since SPHINCS$^+$ uses randomized hashing, any message can be signed by finding a randomizer that results in the forged Merkle tree being used in the signing process. In the worst-case scenario, the forged Merkle tree or FORS key will appear at the bottom the SPHINCS$^+$ hypertree.

[9] This can be optimized so that only one call to the compression function is required to generate each additional Merkle tree root.

[10] If the original key was used to sign a FORS key, then new FORS keys could similarly be created at a cost for 17 calls to the compression function for each new key.

As there are 2^{68} WOTS$^+$ keys at the lowest level, approximately 2^{68} randomizers will need to be tried in order to find one that results in the correct path through the hypertree being used. For each randomizer, two hashes will need to be computed. One of the messages to be hashed will be short. The length of the second message to be hashed will correspond to the length of the message to be signed. Note that once the above steps are completed, we can forge arbitrarily many new messages, each one costing about 2^{68} work to create.

5 Optimizations and Attack Cost Calculations

Our attack uses subroutines that find multi-collisions and multi-target preimages in generic functions. Here we review, and where necessary, adapt the best-known techniques for doing this. We discuss how to estimate costs both in models that ignore memory costs and in models that try to take them into account. As this turns out to only make about 2 bits of security difference in the complexity of our attack, we present our results only in terms of a fairly pessimistic 2-dimensional model of classical memory. We also briefly discuss known quantum speedups. While we don't explicitly analyze how to optimize our attack to take advantage of quantum computation, the existence of known quantum speedups for multi-target preimage search, even in models of computation where memory access is expensive, combined with the ability to drastically reduce the cost of herding (e.g., by targeting 2-collisions instead of 4-collisions) likely means that SPHINCS$^+$'s claimed category five parameters using SHA-256, not only fail to meet category five, but category four as well.

5.1 Collision Search and General Framework

All these techniques follow the paradigm of the Van-Oorschot, Wiener parallel collision search [18]. In each case the computation is divided up among p parallel threads that repeatedly compose the function, f, which is being attacked, starting with a seed, and stopping when a "distinguished point" is reached. The distinguished point is defined by an output that meets a rare, easily identifiable condition. For example, if it is desirable that the expected number of iterations to reach a distinguished point is m, the distinguished point may be an output value which is 0 modulo m. If any output value occurring in one chain appears anywhere in a second chain (i.e., a collision occurs in f), then all subsequent values in both chains will be the same, and both chains will reach the same distinguished point. The collision can be recovered by sorting the p distinguished points to find any duplicates. Once a duplicate resulting from a collision has been found, the actual colliding inputs can be recovered for an expected cost of m additional computations of f; assuming a total iteration count and seed value have been saved for each thread, the collision is recovered by recomputing the output values in both chains and comparing each pair of output values with an iteration count offset by the difference in the number of iterations required to reach the shared distinguished point.

If the function f can be modeled as random and has an n bit output size, then it is expected that approximately $\frac{(mp)^2}{2 \cdot 2^n}$ collisions will be found (This approximation holds as long as $p \gg \frac{(mp)^2}{2 \cdot 2^n}$. When $p < \frac{(mp)^2}{2 \cdot 2^n}$ most of the computations of f are duplicated across multiple chains and are therefore less useful.)

Depending on the parameters of the attack, (m and p), and assumptions on the relative cost of computation, memory, and memory access, the dominant cost of the attack may be any of the following:

1. Building and maintaining $M = p(n + p + \log_2(m))$ bits of memory.
2. Approximately mp computations of f required to compute the distinguished points.
3. Sorting the list of p distinguished point values.

The last cost depends on assumptions regarding the cost of random access queries to memory. If one, unrealistically, assumes memory access costs are independent of the size of memory, the cost of sorting the distinguished points could be assumed to be as small as $np \log_2(p)$). However, it is perhaps more reasonable to assume the cost of random access to a memory of size M follows a power law where, for a d-dimensional memory architecture, the cost per bit to read a register from a memory of size M would be $C \cdot M^{1/d}$. A popular choice of C and d is given by [4], with $C = 2^{-5}$ bit operation equivalents and $d = 2$. This latter estimate likely overestimates the cost of memory access for very large memories, so we will take it is an upper bound for memory costs in order to demonstrate that our attack does not lose much efficacy when memory costs are taken into consideration. For comparison, we will follow [17] in estimating the cost of the SHA-256 compression function as requiring 2^{18} bit operations.

5.2 Multi-target Preimage Search

The problem of finding a preimage of one of t targets with respect to a function f with an n-bit output is discussed by [3], which considers classical and quantum models of computation, with either $O(1)$ or $O(M^{1/2})$ memory access cost.

Classically, the non-memory cost of multi-target preimage search is $\frac{2^n}{t}$ evaluations of f. Section 1.2 of [3] describes how to modify the techniques of [18] to minimize memory costs without significantly increasing computation costs. By our calculations, with respect to SHA-256, this technique makes memory access costs negligible for $t \ll 2^{106}$, which is the case in all examples we consider.

In the quantum case [3] gives a cost per thread for t-target preimage search, using p-way parallelism of $O(\sqrt{\frac{2^n}{pt}})$ in the $O(1)$ memory access cost case, and $O(\sqrt{\frac{2^n}{pt^{1/2}}})$ in the $O(M^{1/2})$ case (the main result of the paper). If we assume, following the NIST PQC Call For Proposals [17] that the quantum circuit is depth-limited, then these costs represent a cost savings factor compared to single target preimage search of $O(t)$ in the $O(1)$ memory access cost case and $O(t^{1/2})$ in the $O(M^{1/2})$ case.

5.3 Multi-collision Search

The use of parallel collision search techniques for finding k-way collisions is described in [10]. As with 2-way parallel collision search, the computation is divided among p threads, each computing f iteratively on a seed until a distinguished point is reached after an average of m steps. In order for a k-way collision to be found with reasonable probability, the total number of computations of f, mp must satisfy:

$$\frac{(mp)^k}{k! \cdot 2^{n(k-1)}} \geq 1,$$

and to ensure that most of the computations of f act on distinct inputs, p must be set to be comparable to the expected number of 2-way collisions, i.e., so that:

$$p \geq \frac{(mp)^2}{2 \cdot 2^n}.$$

5.4 Batched Multi-target Multi-collision Search

A key step in our attack requires us to herd together t/k groups of k hash inputs where each hash input must have a prefix chosen without repetition from a list of t targets $S_0, ..., S_{t-1}$. We can relate this procedure to finding collisions in a single function f by defining a function $f(x)$ with an n-bit input and output that hashes an input, injectively derived from x with the prefix S_i indexed by $i = x \bmod t$. Then our goal is to find t/k k-collisions in f meeting the constraint that each input x has a different remainder mod t.

While it is possible to compute each k-collision individually, it is generally more efficient to compute the k-collisions in large batches, taking advantage of the fact that we don't care which prefixes collide. This is because finding t/k k-collisions in f only requires $(t/k)^{1/k}$ times as many queries to f as finding one collision. This situation is somewhat complicated by the requirement that each input has a different prefix, but nonetheless we find that efficiency is optimized when k-collisions are computed in batches of size $\alpha(t/k)$, where α is a constant of the same order of magnitude as 1, whose exact value depends on the cost of f, the value of k, and assumptions about memory costs.

We expect a batch of $\alpha(t/k)$ k-collisions to contain approximately $\beta(t/k)$ k-collisions with different prefixes, where β is given by the differential equation $\frac{d\beta}{d\alpha} = (1-\beta)^k$ and the initial condition $\beta = 0$ when $\alpha = 0$. Once this batch of k-collisions is computed, the search for k-collisions continues recursively with t reduced by a factor of $1 - \beta$.

Optimal values of α and β depend upon whether the complexity of batched collision search is dominated by queries to f or memory access while sorting the list of distinguished points. The number of queries required is proportional to $\alpha^{1/k}$, while the size of the list of distinguished points is proportional to the square of the number of queries, i.e., proportional to $\alpha^{2/k}$. If we assume, according to a 2-dimensional memory model, that the cost of sorting the list scales with the

3/2 power of the list size, then rather than scaling as $\alpha^{1/k}$, the cost of computing a batch of k collisions will scale like $\alpha^{3/k}$.

For $k = 4$ the values of α and β that minimize the cost per k-collision are $\alpha = 3.27$; $\beta = 0.548$ ignoring memory costs and $\alpha = 0.185$; $\beta = 0.137$ assuming costs are dominated by square root memory access costs while sorting the list of distinguished points. Interestingly, the situation relevant to analyzing our attack in a 2-dimensional memory model will turn out to be intermediate between these two cases for reasons we will discuss shortly, so we will use $\alpha = 1$; $\beta = 0.37$. For $k = 3$ the cost of sorting is dominated by the cost of queries for all values of t that are of interest to us, so whether memory is included or not, the optimal values for α and β are $\alpha = 1.89$ $\beta = 0.543$.

We will now go on to give a concrete estimate of the cost of batched 4-collision search for $t > 2^{27}$ assuming the cost to access a bit of memory in a memory of size M is equivalent to $2^{-5} \cdot \sqrt{M}$ bit operations, and the cost of the SHA-256 compression function is equivalent to 2^{18} bit operations.

First we analyze the cost of computing f: Since the function f must retrieve a chaining value corresponding to one of t target prefixes, as well as the address string, computing f will require looking up approximately 2^9 bits in a memory of size $2^9 t$. (We will assume this memory is shared among many threads so as not to inflate the size of the memory in which the distinguished points are stored.) The cost of this memory access is equivalent to $2^{8.5} t^{1/2}$ bit operations, and this is the dominant cost of f for $t > 2^{27}$, even if computing f requires computing a hash chain of length $w - 1 = 15$

The number of queries required by the first batch is $q = 2^{192}(4!\alpha t/4)^{1/4}$, and since the cost of a query is $O(t^{1/2})$, the cost of a batch is $O(t^{3/4})$, which means each subsequent batch is cheaper than the previous batch by a factor of $(1 - \beta)^{3/4}$. If we use the summation for a geometric series to estimate the total cost of computing all batches required to find the full set of $t/4$ 4-collisions, we find that the total cost of queries to f is $\frac{1}{1-(1-\beta)^{3/4}} \cdot \alpha^{1/4} t^{3/4} \cdot 2^{200.8}$.

Now we consider the cost of sorting the list of distinguished points. In order for p to be at least comparable to the number of 2-collisions existing in q queries, we need $p = \frac{q^2}{2 \cdot 2^{256}}$. The cost of sorting distinguished points associated with the first batch is then given by $2^{-5}(2^9 p)^{3/2}$, which is $O(t^{3/4})$. We can therefore use the same rule as before to sum the costs of all the batches. The resulting cost is $\frac{1}{1-(1-\beta)^{3/4}} \cdot \alpha^{3/4} t^{3/4} \cdot 2^{200.9}$.

Summing these costs with $\alpha = 1$ ($\beta = 0.37$) gives a total cost of approximately $2^{203.7} t^{3/4}$ bit operations, or the equivalent of approximately

$$\text{MemoryCost}(k = 4, t > 2^{27}) \approx 2^{185.7} t^{3/4} \tag{1}$$

SHA-256 compression function calls.

If we instead ignore memory costs , then the cost of batched 4-collision search is dominated by approximately $2^{195.5} t^{1/4}$ queries to f. The computational cost of the queries will either be 1 SHA-256 compression function computation (if we can freely choose 256 input bits without needing to know a preimage) or 16 compression functions (if we need to construct a hash chain of length $w-1 = 15$).

We can do similar calculations for $k = 3$. In both the free memory cost model and the square-root memory cost model, the cost is dominated by approximately $2^{173.4}t^{1/3}$ queries to f. In the free memory cost model, the cost of each query is the equivalent of either 1 or 16 compression functions (similar to the 4-collision case). In the square root memory cost model, the total cost of all the queries is the equivalent of $2^{162.8} \cdot t^{5/6}$ compression function computations.

6 Conclusions

In this paper, we have shown how to extend Antonov's attack on the PQ-DM-SPR property in SPHINCS$^+$ into a full signature forgery attack, allowing an attacker to forge signatures on arbitrary messages. This attack requires access to a large number of legitimate signatures formed by the key, and an enormous computation which, while practically infeasible, is substantially below the claimed security strength for the category five parameter sets of SPHINCS$^+$.

We do not believe this attack calls the general soundness of the SPHINCS$^+$ design into question. Combined with the earlier observations on the PQC forum regarding weaknesses in category five security in the message hashing [19], it seems clear that both their attack and ours are made possible by the SPHINCS$^+$ designers' attempt to use a 256-bit Merkle-Damgård hash like SHA-256 to generically get 256 bits of security.[11]

Very recently in [8], Hülsing has described a set of tweaks to SPHINCS$^+$ to address a number of observations and proposed attacks, including Antonov's and ours. The relevant tweak for our attack is to the T_ℓ tweakable hash function–for category three or five security, the function now uses SHA-512 instead of SHA-256. This change means that building the diamond structure and finding the linking message (see Section 3) for the hash function requires at least 2^{256} hash function computations, effectively blocking both Antonov's attack and our own.

Our work leaves many questions open. Among them:

1. Are there still places within SPHINCS$^+$ in which the internal properties of SHA-256 can be used to carry out some attack with less than 2^{256} work, despite the tweak? SHA-256 is still used in the definition of F and PRF, even for category five security.
2. Can these or similar techniques be used to attack the category three (192 bit classical security) parameters?
3. Is there a technique to construct inputs to the hash which can be shown to prevent all such attacks, despite using SHA-256 to achieve 256-bit security? This would allow the use of the slightly more efficent SHA-256 instead of SHA-512 for category three or five security.

[11] Modeling a Merkle-Damgård hash function as a random oracle can easily give misleading results for more than $2^{n/2}$ queries. Indeed, even modeling the *compression function* as a random oracle may not work, since the SHA-256 compression function is constructed from a large block cipher in Davies-Meyer mode. Note that this means that fixed points for the hash function are easy to find, as exploited in [6, 12].

References

1. Antonov, S.: Round 3 official comment: SPHINCS+ (2022), https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/FVItvyRea28/m/mGaRi5iZBwAJ
2. Aumasson, J.P., Daniel J. Bernstein, W.B., Dobraunig, C., Eichlseder, M., Fluhrer, S., Stefan-Lukas Gazdag, A.H., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Westerbaan, B.: SPHINCS$^+$ – submission to the NIST post-quantum project, v.3 (2020)
3. Banegas, G., Bernstein, D.J.: Low-communication parallel quantum multi-target preimage search. In: Adams, C., Camenisch, J. (eds.) Selected Areas in Cryptography – SAC 2017. pp. 325–335. Springer International Publishing, Cham (2018)
4. Bernstein, D.J., Brumley, B.B., Chen, M.S., Chuengsatiansup, C., Lange, T., Marotzke, A., Peng, B.Y., Tuveri, N., van Vredendaal, C., Yang, B.Y.: NTRU Prime: round 3. Submission to the NIST's post-quantum cryptography standardization process (2020)
5. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS$^+$ signature framework. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 2129–2146. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363229
6. Dean, R.D.: Formal Aspects of Mobile Code Security. Ph.D. thesis, Princeton University, USA (1999)
7. Hülsing, A.: W-OTS+ – shorter signatures for hash-based signature schemes. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) Progress in Cryptology – AFRICACRYPT 2013. pp. 173–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. Hülsing, A.: Round 3 official comment: SPHINCS+ (2022), https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/Ca4zQeyObOY
9. Joux, A.: Multicollisions in iterated hash functions. application to cascaded constructions. In: Franklin, M. (ed.) Advances in Cryptology – CRYPTO 2004. pp. 306–316. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
10. Joux, A., Lucks, S.: Improved generic algorithms for 3-collisions. In: Matsui, M. (ed.) Advances in Cryptology – ASIACRYPT 2009. pp. 347–363. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
11. Kelsey, J., Kohno, T.: Herding hash functions and the Nostradamus attack. In: Vaudenay, S. (ed.) Advances in Cryptology - EUROCRYPT 2006. pp. 183–200. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
12. Kelsey, J., Schneier, B.: Second preimages on n-bit hash functions for much less than 2^n work. In: Cramer, R. (ed.) Advances in Cryptology – EUROCRYPT 2005. pp. 474–490. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
13. Lamport, L.: Constructing Digital Signatures from a One Way Function. Tech. rep., SRI (Oct 1979), https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Constructing-Digital-Signatures-from-a-One-Way-Function.pdf
14. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology — CRYPTO' 89 Proceedings. pp. 218–238. Springer New York, New York, NY (1990)

15. Merkle, R.C.: Secrecy, authentication, and public key systems. Ph.D. thesis, Stanford university (1979)
16. National Institute of Standards and Technology: NIST post-quantum cryptography standardization (2016), https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization
17. National Institute of Standards and Technology: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2016), https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf
18. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with application to hash functions and discrete logarithms. In: Proceedings of the 2nd ACM Conference on Computer and Communications Security. p. 210–218. CCS '94, Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/191177.191231, https://doi.org/10.1145/191177.191231
19. Stern, M.: Re: Diversity of signature schemes (2021), https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/2LEoSpskELs/m/LkUdQ5mKAwAJ