

On Forging SPHINCS⁺-Haraka Signatures on a Fault-tolerant Quantum Computer*

Robin M. Berger and Marcel Tiepelt

KASTEL, Karlsruhe Institute of Technology, Germany,
{robin.berger,marcel.tiepelt}@kit.edu

Abstract. SPHINCS⁺ is a state-of-the-art hash based signature scheme, the security of which is either based on SHA-256, SHAKE-256 or on the Haraka hash function. In this work, we perform an in-depth analysis of how the hash functions are embedded into SPHINCS⁺ and how the quantum pre-image resistance impacts the security of the signature scheme. Subsequently, we evaluate the cost of implementing Grover’s quantum search algorithm to find a pre-image that admits a universal forgery.

In particular, we provide quantum implementations of the Haraka and SHAKE-256 hash functions in Q# and consider the efficiency of attacks in the context of fault-tolerant quantum computers. We restrict our findings to SPHINCS⁺-128 due to the limited security margin of Haraka. Nevertheless, we present an attack that performs better, to the best of our knowledge, than previously published attacks.

We can forge a SPHINCS⁺-128-Haraka signature in about $1.5 \cdot 2^{90}$ surface code cycles and $2.03 \cdot 10^6$ physical qubits, translating to about $1.55 \cdot 2^{101}$ logical-qubit-cycles. For SHAKE-256, the same attack requires $8.65 \cdot 10^6$ qubits and $1.6 \cdot 2^{84}$ cycles resulting in about $1.17 \cdot 2^{99}$ logical-qubit-cycles.

Keywords: Post-quantum cryptography · quantum implementation · resource estimation · cryptanalysis.

Table of Contents

1	Introduction	2
2	Preliminaries	5
3	Reversible Implementations	10
4	Attacking the SPHINCS ⁺ Signature Scheme	14
5	Fault-tolerant Cost	19
6	Conclusion	22
A	pre-image Attack on the Hash Functions using Grover’s Algorithm	25
B	Further Attacks	26

* A reduced version of this paper was originally published in Springer’s LNCS series.
https://doi.org/10.1007/978-3-030-88238-9_3

1 Introduction

Overview and Related Work Ongoing research in the area of quantum technologies has led to the belief that quantum computers will be able to break current public-key cryptosystems within the coming decades. On the contrary, symmetric-key primitives are believed to be somewhat resistant against quantum attacks, with the most promising generic attack being Grover’s search algorithm [15]; Its quadratic improvement over a classical brute force search can easily be countered by doubling the key length of the underlying primitives.

In order to prepare for the (public-key) quantum menace, the National Institute for Standards and Technology (NIST) started the post-quantum standardization competition in 2017. The NIST competition features 5 security levels [24,23]: The first level provides security equivalent to performing a key search on AES-128, the second a collision attack on SHA-256 and the fifth a key search on AES-256. Moreover they categorize attacks with quantum computers according to the maximal circuit depth, where each level resembles a number of gates that can be serially computed over a plausible time period. Specifically, NIST estimates that quantum circuits up to a depth of 2^{40} gates can be computed within a single year, up to a depth of 2^{64} in a single decade and up to 2^{96} in a millennium. Respectively, the number of quantum gates to break AES is estimated by NIST to be $2^{170}/\text{MAXDEPTH}$, i.e. 2^{130} , 2^{104} and 2^{74} . [17] gave precise estimates for attacking AES-128 for different values of maxdepth and respective parallelization. Equivalently, NIST estimates 2^{143} classical computational steps. However, we note the most promising attack on AES-128 can be performed in $2^{126.1}$ classical steps as shown by [4].

From initial 69 submissions to the competition, only 7 were selected as finalists [24]. Additionally, 8 schemes were chosen as alternate candidates based on a high confidence of their security, but with a drawback in performance compared to the 7 finalists. Briefly speaking, they may be considered as *backup* candidates for standardization. Among the alternate candidates is the stateless hash-based signature scheme SPHINCS⁺ [16]. SPHINCS⁺ builds on the hardness of inverting one-way functions, i.e., Haraka [20], SHAKE-256 [22] or SHA-256 [11], the first of which can be derived from block-ciphers and thus is believed to provide similar security guarantees against quantum adversaries.

An estimate of the security of SPHINCS⁺, based on cryptographic assumptions, was given within the scope of the NIST submission: The authors considered general attacks [16, Sec. 9.3.1] on the distinct-function multi-target second pre-image resistance of the underlying hash functions and estimated the success probability of such an attack as $\Theta\left(\frac{(q_{\text{hash}}+1)^2}{2^n}\right)$, where q_{hash} is the number of hash queries and n a security parameter. Generally, they quantify the security based on the number of required hash function invocations and thus on the probability of an successful adversary.

SPHINCS⁺ features parameters for each security levels, i.e. SPHINCS⁺-SHAKE-256 and SPHINCS⁺-SHA-256 both provide a sufficient amount of security for all 5 NIST security levels. On the other side, SPHINCS⁺-Haraka achieves security level 1 or 2 at most.

The security of SPHINCS⁺ is closely related to the security of the underlying hash functions. An analysis of the security of SHAKE-256 has been given by [1], whose result is the main motivation for our work. They present a quantum circuit to implement a Grover search and attack the 256-bit pre-image resistance of the SHA3-256 hash function and give concise and fault-tolerant estimates for the resources required to implement such a circuit: They claim that their circuit requires $2^{153.8}$ surface code cycles using $2^{12.6}$ logical qubits, resulting in an overall requirement of about $2^{166.4}$ logical-qubit-cycles using 2^{128} black box queries for a 256-bit pre-image search. Their results may be adapted to estimate the work required to break the hash function for the SPHINCS⁺ signature scheme. However, there is still considerable ambiguity on the specific construction to forge a signature.

The quantum security of Haraka has not been explicitly analyzed yet. However, due to the capacity of the sponge construction in SPHINCS⁺-Haraka using only 256 bits, attacking the second-pre-image-resistance as described in [3] only requires about $2^{129.5}$ classical hash function invocations, producing a collision on the internal state of the hash function in the process. The best known generic quantum collision attacks on hash functions is the BTH algorithm by [7], which finds a collision using $\mathcal{O}(2^{n/3})$ Grover iterations, (where n is a security parameter), however, also requiring $\mathcal{O}(2^{n/3})$ quantum RAM (QRAM). The concept of QRAM is highly controversial, as quantum states that interact with the environment eventually decay. [10, Thm 2] presented a trade-off using only $\tilde{\mathcal{O}}(n)$ ¹ QRAM but $\tilde{\mathcal{O}}(2^{2n/5})$ Grover iterations, resulting in a work effort of about 2^{102} iterations for a collision search with $n = 256$ on Haraka.

The (quantum) invocation of the hash function induces a significant overhead and has to be accounted for. Moreover the implementation on a fault-tolerant quantum computer requires additional overhead to compensate for error correction within the circuit. In our analysis we adapt the concept of logical-qubit-cycles as quantum cost metric, such that each cycle is roughly equivalent to a single (classical) hash function invocation [1]. Briefly speaking, a logical-qubit-cycle is the time-space product of the number of fault-tolerant quantum gates (time) and the number of qubits (space) that is used during the computation. Thus, we can consider the time-space product of Grover iterations and memory, which is $\tilde{\mathcal{O}}(2^{3n/5})$, resulting in a cost of about 2^{153} .

Contribution In this work we consider attacks on SPHINCS⁺ based on inverting the underlying hash functions at specific points, i.e. attacking the XMSS, WOTS or FORS structure or the message digest function. We chose particularly Haraka, because of its placement as a potential component within the NIST competition. Moreover, pre-image resistance of the Haraka [20] hash function has not, to the best of our knowledge, been explicitly evaluated in the quantum setting in any literature. We estimate the logical resources² required to imple-

¹ $\tilde{\mathcal{O}}(x)$ ignores logarithmic factors.

² The estimated resources are based on assumptions on the current (2021) state-of-the-art in quantum computing. We note that the results of our analysis may be subject to change with further advances.

Target	Component	Existential Forgery	Universal Forgery
$\mathcal{H}(m, \dots) := y$	\mathcal{H}	✓ oracle depth = 1	✗
σ_{FORS}^y	FORS	✓	✓ oracle depth = 2, or multiple pre-images
$\sigma_{\text{HT}}^{pk_{\text{FORS}}}$	WOTS	✓	✓ oracle depth = 5
$\text{Path}_{\text{XMSS}}$	XMSS Path	✓	✓ oracle depth = 1

Fig. 1: Overview of oracle depth for attacking different components in SPHINCS⁺.

ment our attacks on the Haraka as well as the SHAKE-256 hash-functions and further estimate the fault-tolerant cost to attack the SPHINCS⁺-128 scheme, using either of the hash functions. For the sake of completeness and comparability we also present the numbers to attack the SPHINCS⁺-256 scheme.

The attacks on the different components differ in (1) the overhead introduced by implementing the Grover oracle on the pre-image resistance, and (2) by a classical overhead introduced when placing a forged signature into a valid hypertree. In this work we focus mostly on (1), the results of which are summarized in Fig. 1.

In Section 2 we recall parts of the SPHINCS⁺, Haraka and SHAKE-256 scheme, and review the Grover algorithm with respective metrics for fault-tolerant quantum computing. In our work, we use the logical-qubit-cycles metric (introduced in [1]) which compares to classical hash function invocations.

Section 3 shows the results for our implementation³ of the hash functions in Q#. To construct a circuit for Haraka, we partially reused the work of [17] on AES functions, resulting in the first implementation of the Haraka hash function in the quantum setting. The implementation for SHAKE-256 was built from scratch. For both circuits, we consider the number of qubits as well as different metrics based on the gate count and T-Depth. As a result, our implementation of the Haraka512 permutation in the hash function consumes about $2.2 \cdot 10^6$ quantum gates on 1144 logical qubits. Our Keccak permutation in the SHAKE-256 hash function consumes about $3.3 \cdot 10^6$ quantum gates on 3200 logical qubits.

In Section 4, we analyze the most promising points of attack in the SPHINCS⁺ signature scheme. We propose that the weakest link is the XMSS authentication path for a given WOTS⁺ public key, as this allows a universal forgery attack. Our most promising attack on SPHINCS⁺-128-Haraka requires about $1.6 \cdot 2^{86}$ quantum gates. The same circuit to attack SPHINCS⁺-128-SHAKE-256 has about $1.2 \cdot 2^{86}$ gates.

³ <https://github.com/RobinBerger/Grover-Sphincs>

In Section 5, we partially follow the approach of [1] to estimate the resources for this attack in the context of fault-tolerant quantum computing. We compute the amount of error correction in terms of surface code cycles and the optimal scheme for magic state distillation.

For the Haraka hash function, our attack requires $3.91 \cdot 10^{30} \approx 1.55 \cdot 2^{101}$ logical-qubit-cycles on $2.03 \cdot 10^6$ physical qubits, which is better than the generic quantum collision attack on the hash function, which requires 2^{102} quantum hash function invocations (without considering the cost of implementing the hash function), or a time-space product of 2^{153} , which appears to be the more realistic comparison to the cost of logical-qubit-cycles. Performing our attack with the SHAKE-256 hash function instead requires $7.44 \cdot 10^{29} \approx 1.17 \cdot 2^{99}$ logical-qubit-cycles on $8.65 \cdot 10^6$ physical qubits.

2 Preliminaries

2.1 The SPHINCS⁺ Signature Scheme

In this section we partially review the SPHINCS⁺ signature scheme as proposed and submitted by [16] to the second and third round of NIST’s post-quantum cryptography competition. The structure of the SPHINCS⁺-scheme combines a hypertree (HT) of eXtended Merkle Signature Schemes (XMSS) and Winternitz One-Time Signature schemes (WOTS) with a Forest Of Random Subsets (FORS) as represented in Fig. 2.

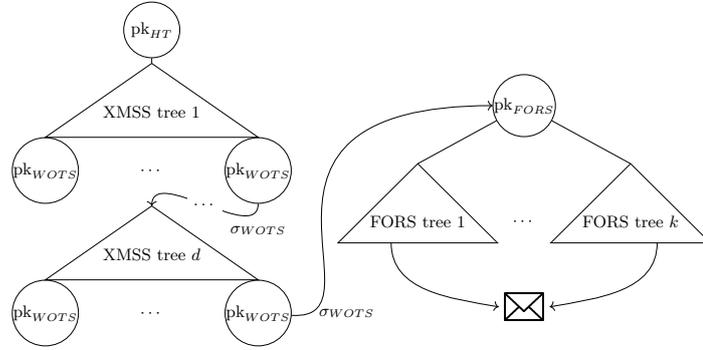


Fig. 2: Overview of a SPHINCS⁺ hypertree.

In the following, we consider a signature σ_y^x using the scheme y to sign the message x and a hash function $\mathcal{H} \in \{\text{SHAKE-256}, \text{Haraka-512}, \text{Haraka-sponge}\}$ for all the subsequent hashes. Moreover, each scheme is associated with a $\text{KeyGen}(\cdot)$, $\text{Sign}(\cdot)$ and $\text{Verify}(\cdot)$ functionality. Let $\text{pk}_{\text{SPHINCS}^+}, \text{sk}_{\text{SPHINCS}^+}$ be a SPHINCS⁺ key pair associated with seeds to deterministically generate the

Algorithm 1: SPHINCS⁺ – KeyGen()

<ol style="list-style-type: none"> 1 $sk_{\text{seed}} \xleftarrow{\\$} \{0, 1\}^n, sk_{\text{prf}} \xleftarrow{\\$} \{0, 1\}^n, pk_{\text{seed}} \xleftarrow{\\$} \{0, 1\}^n$ 2 $pk_{\text{root}} \leftarrow \text{KeyGen}_{\text{HT}}(sk_{\text{seed}}, pk_{\text{seed}})$ 3 return $(pk_{\text{SPHINCS}^+} := (pk_{\text{seed}}, pk_{\text{root}}), sk_{\text{SPHINCS}^+} := (sk_{\text{seed}}, sk_{\text{prf}}))$
--

subsequent keys of the scheme. Then a signature of a message m is a tuple of value the σ_r and signatures from the hypertree and signatures instances:

$$\sigma_{\text{SPHINCS}^+}^m := \left(\sigma_r \parallel \sigma_{\text{FORS}}^m \parallel \sigma_{\text{HT}}^{\text{pk}_{\text{FORS}}} \right). \quad (1)$$

The sub-signature $\sigma_{\text{HT}}^{\text{pk}_{\text{FORS}}}$ itself consists of one XMSS signature for each layer in the hypertree, e.g. $(\sigma_{\text{XMSS},1}^{\text{pk}_{\text{XMSS},2}}, \sigma_{\text{XMSS},2}^{\text{pk}_{\text{XMSS},3}}, \dots, \sigma_{\text{XMSS},i}^{\text{pk}_{\text{FORS}}})$ for a hypertree height of i . σ_r will be mostly ignored in the remaining paper. During signing, one generates a FORS instance, signs a message digest with the FORS key, and signs the FORS pk with the hypertree.

The hypertree consists⁴ of several layers of XMSS instances. Each XMSS instance is a binary hash tree with WOTS schemes at the leaves, where the value of each node is the output of hashing its child nodes. Each XMSS tree is associated with a root node pk_{XMSS} and a set of WOTS keys $pk_{\text{WOTS}}, sk_{\text{WOTS}}$.

An XMSS signature consists of a WOTS signature and an authentication path $\sigma_{\text{XMSS}}^x := (\sigma_{\text{WOTS}}^x, \text{path}_{\text{XMSS}})$, where $\text{path}_{\text{XMSS}}$ consists of all sibling nodes on the path from a leaf to the root of the tree. The WOTS instances at the leaf nodes are then used to sign the root node of the next layer, resulting in a hypertree. The root node of the top tree is the public key of the hypertree. The bottom WOTS instances represent the respective secret key of the hypertree that is used to create the signature $\sigma_{\text{HT}}^{\text{pk}_{\text{FORS}}}$.

To validate a signature $\sigma_{\text{SPHINCS}^+}^m$, one first computes a FORS public key from σ_{FORS}^m and then verifies the hypertree signature $\sigma_{\text{HT}}^{\text{pk}_{\text{FORS}}}$. For the latter, one has to compute the authentication path through the hypertree and finally compare the resulting public key pk'_{HT} to the key associated with the SPHINCS⁺ signature scheme. The Algorithms 1, 2, 3 review these procedures using the respective signature schemes and a function `prf.msg`, that generates a pseudo-random value as part of the signature. We note that the description is not complete (as in [16]), i.e. it is restricted to a level appropriate to follow the remaining paper.

2.2 Quantum Computing

We assume the reader to be familiar with the basics of quantum information theory (e.g. see [25]). In the following we first describe the general attack strategy

⁴ The hypertree exists implicitly only, and a actual path in the tree is generated during signing/ verification only.

Algorithm 2: SPHINCS⁺ – Sign($m := \{0, 1\}^*$, sk_{SPHINCS^+})

```

1  $r \xleftarrow{\$} \{0, 1\}^n$ 
2  $\sigma_r \leftarrow \text{prf\_msg}(sk_{\text{prf}}, r, m)$ 
3  $md \leftarrow \mathcal{H}(\sigma_r, pk_{\text{seed}}, pk_{\text{root}}, m)$ 
4  $\sigma_{\text{FORS}}^{md} \leftarrow \text{Sign}_{\text{FORS}}(md, sk_{\text{seed}}, pk_{\text{seed}})$ 
5  $pk_{\text{FORS}} \leftarrow \text{pkFromSig}_{\text{FORS}}(\sigma_{\text{FORS}}, m, pk_{\text{seed}})$ 
6  $\sigma_{\text{HT}}^{pk_{\text{FORS}}} \leftarrow \text{Sign}_{\text{HT}}(pk_{\text{FORS}}, sk_{\text{seed}}, pk_{\text{seed}}, md)$ 
7 return ( $\sigma := (\sigma_r || \sigma_{\text{FORS}}^{md} || \sigma_{\text{HT}}^{pk_{\text{FORS}}})$ )
    
```

Algorithm 3: SPHINCS⁺ – Verify

($\sigma := (\sigma_r || \sigma_{\text{FORS}}^{md} || \sigma_{\text{HT}}^{pk_{\text{FORS}}})$, m , pk_{SPHINCS^+})

```

1  $md \leftarrow \mathcal{H}(\sigma_r, pk_{\text{seed}}, pk_{\text{root}}, m)$ 
2  $pk_{\text{FORS}} \leftarrow \text{pkFromSig}_{\text{FORS}}(\sigma_{\text{FORS}}, m, pk_{\text{seed}})$ 
3 return  $\text{Verify}_{\text{HT}}(pk_{\text{FORS}}, \sigma_{\text{HT}}, pk_{\text{seed}}, md, pk_{\text{root}})$ 
    
```

using Grover’s algorithm [15]. Then, we recall the setup to estimate quantum resources on a fault-tolerant quantum computing architecture based on the excellent description of [19] using surface codes [13] and magic state distillation [8].

2.3 Grover’s Algorithm on Pre-image Resistance

For a fixed n , given a predicate $p : \{0, 1\}^n \rightarrow \{0, 1\}$ marking M elements $x \in \{0, 1\}^n$, Grover’s algorithm finds an element x , for which $p(x) = 1$. Let the initial superposition be $|\phi\rangle = \sqrt{(N-M)/N} |\{x|p(x) = 0\}\rangle + \sqrt{M/N} |\{x|p(x) = 1\}\rangle$. Then the algorithm of Grover operates in the space spanned by $|\phi\rangle$ and $|\{x|p(x) = 1\}\rangle$, where $\langle\phi| |\{x|p(x) = 1\}\rangle = \sin(\theta)$. The initial value is $\theta = \arcsin(\sqrt{M/N})$, and is increased in every iteration by roughly $\sqrt{M/N}$, where the advance diminishes during the last few iterations. Thus the probability to measure a marked element is the largest after $R = \lceil \pi/4 \sqrt{N/M} \rceil$ Grover iterations. Our implementation of the Grover iteration ⁵ follows the principle construction for oracle invocations.

If the number of matches M is not (exactly) known, and one performs too many iterations, the value of θ decreases. Instead one can run Grover’s algorithm multiple times with different values for M . [6, Theorem 3] have shown that the expected number of iterations remains in $\mathcal{O}(\sqrt{N/M})$.

In the context of hash functions and the random oracle model, we assume the number of matches to be $M = 1$, i.e. we are given a single value y and we are looking for a value x , so that $y = \mathcal{H}(x)$. Whereas there is no guarantee that there are no collisions (i.e. $M > 1$), $M = 1$ is to be expected, since the input and output domain of the hash functions are of equal size in our case.

⁵ A circuit describing the implementation Grover’s Algorithm is given in Appendix A.

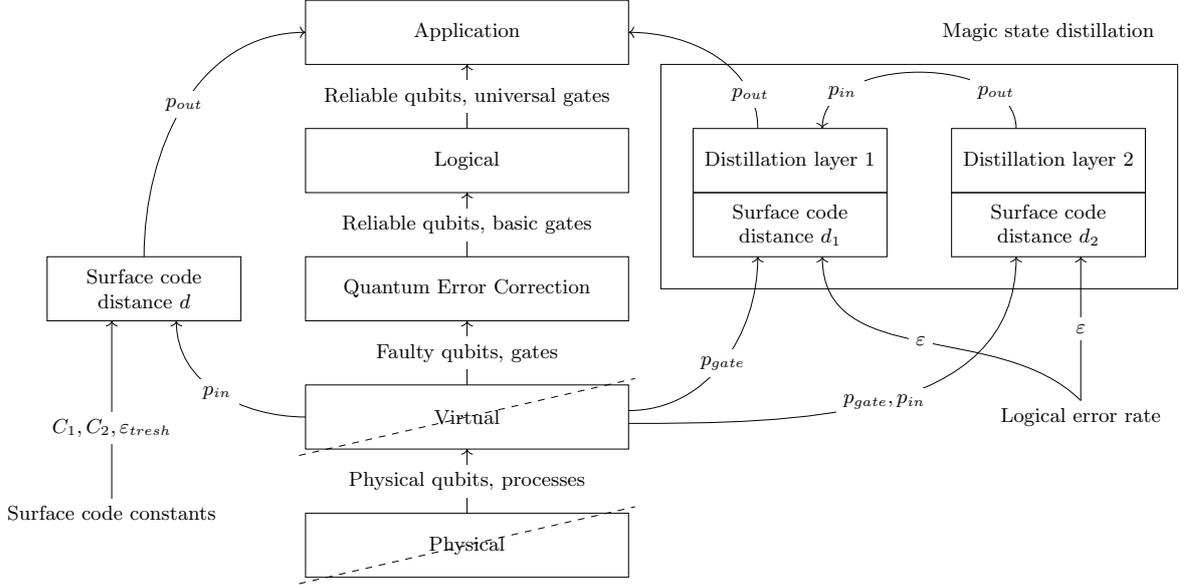


Fig. 3: Layered architecture for quantum computers including parameters for the error correction layer (left) and exemplary magic state distillation (right).

2.4 Fault-tolerant Resource Estimation

The layered architecture in [19] describes the physical design of a fault-tolerant quantum computer. The first and second layer cover the physical processes and the virtual interfaces of the hardware and are not considered in the analysis. The third layer provides reliable QubitClifford-gates, but not T-gates, by performing a series of measurements and faulty gate applications on physical qubits to correct errors. Each of these intervals is called a surface code cycle. Then, the logical layer provides a universal gate set. The final layer consists of the application of Grover's algorithm.

In the following we describe the top three layers in more detail, review the cost metrics of [1] in our setting and explicitly mention the assumptions (since quantum benchmarks are not available) required for the analysis. Our description of the different layers, which are pictured in Fig. 3, is tailored to our resource estimate. We combine these with the cost metrics used by [1] for comparability.

Assumption 1 The cost for a computation of a large-scale fault-tolerant quantum computer is well approximated using surface codes [1,26,18].

The following parameters approximate today's state of the art [1,12]. We use these for comparability, but note that other values have also been suggested [19].

Assumption 2 p_{in} is the initial error probability of a quantum state, i.e. before any layer of error correction $p_{\text{in}} \approx 10^{-4}$. $p_{\text{gate}} \approx p_{\text{in}}/10$ is the gate error rate. $T_{sc} = 200ns$ is the approximate time for a single surface code cycle.

Assumption 3 All quantum gates are distributed uniformly across all layers.

While Assumption 3 does not hold for our oracle implementation per se, it does so for the Grover algorithm over multiple Grover iterations.

Quantum Error Correction Let $C_1, C_2, \varepsilon_{\text{thresh}}$ be parameters determined by the implementation of the surface code with distance d . Given an initial error rate of p_{in} (provided by the underlying layer, or as the resulting error probability after *some* error correction) one can calculate the distance d for a targeted error rate p_{out} as per $p_{\text{out}} \approx C_1 (C_2 p_{\text{in}} / \varepsilon_{\text{thresh}})^{\lfloor d+1/2 \rfloor}$ [19, Sec. IV.B]. We follow the suggestion in [14, Fig. 8] and estimate that each logical qubit requires $2 \cdot (d+1)^2$ physical qubits to be implemented in a surface code with distance d .

Logical layer We deploy the Reed-Muller-15-to-1 distillation introduced by [8], each layer uses 15 magic states with an input error rate of p_{in} and produces one magic state with lower error rate $p_{\text{dist}} \approx 35p_{\text{in}}^3$. We follow the work of [1] and assume that the amount of logical errors introduced during distillation is already covered in the process resulting in $p_{\text{out}} = (1 + \varepsilon)p_{\text{dist}}$, hence $p_{\text{in}} \approx \sqrt[3]{p_{\text{out}}/35(1+\varepsilon)}$. The distillation is repeated in multiple layers i until p_{out} reaches a target value.

Let d_i be a surface code distance for layer i with $i = 1$ being the top (= output) layer of distillation, where each distillation requires $10 \cdot d_i$ cycles. For this, [12, Sec. II] gives an example calculation, [1, Alg. 4] gives an explicit algorithm that takes an initial gate error p_{gate} and calculates the number of layers of magic state distillation as well as their respective surface code distance. Each layer i requires $16 \cdot 15^{(i-1)}$ logical qubits. The number of physical qubits in the code is calculated based on the respective surface code.

Application Layer

For our implemented circuits we consider the total count for T-, CNOT- and QubitClifford gates, along with the T-depth and T-width, motivated in [17,1]. For a circuit implementing our attacks, i.e. using Grover's algorithm, let G^d be the total depth (i.e. number of layers) of a circuit and let SCC be the number of surface code cycles for each layer. First, we consider the total number of surface code cycles as

$$\text{COST}_{\text{SCC}} = \text{SCC} \cdot G^d.$$

Then, we consider the number of logical qubits Q_G^{log} required to implement the Grover algorithm and the number of logical qubits Q_{MD}^{log} to perform the magic state distillation. Finally, we consider the metric of logical-qubit-cycles from [1, As. 4 and Cost Metric 1], where each cycle is comparable to one (classical) hash function invocation. The number of logical-qubit-cycles is considered to be the total cost of the attack:

$$\text{COST}_{\text{lqc}} = \text{COST}_{\text{SCC}} \cdot (Q_G^{\text{log}} + Q_{MD}^{\text{log}}).$$

Algorithm 4: Haraka512Permutation($A[x] : \{0,1\}^{128}, 0 \leq x < 4$)

```

1 for  $0 \leq i < 5$  do
2   for  $0 \leq j < 4$  do
3      $A[j] := aesEnc(aesEnc(A[j], key_{i,j,1}), key_{i,j,0})$ 
4    $A := mix(A)$ 
5 return  $A$ 

```

We consider this metric to be the most fitting in comparison to the time-space product given for the best generic attack in [10].

3 Reversible Implementations

We implemented⁶ the Haraka and SHAKE-256 hash functions in Q#. We briefly review the schemes and describe our reversible implementations. To the best of our knowledge, this is the first reversible implementation of Haraka. A resource estimate for carrying out a pre-image attack on these hash function is in Appendix A.

3.1 Haraka

Haraka, as specified in [20], consists of AES encryptions (*aesEnc*) and a mixing step (*mix*) for the permutation, which is used in turn to instantiate a sponge construction with a capacity of 256 bits, resulting in the Haraka-Sponge hash function. The Haraka512 hash function is defined as the truncated XOR of the input value and the output of the Haraka512 permutation on said input. Algorithm 4 describes the Haraka512 permutation. We partially reuse the AES implementation from [17] and adjust it to our use case.

For the AES encryption[21], we implement each of its four steps. The *SubBytes* step consists of applying the AES S-Box on each 8-bit block of the input. We use the implementation of [17] for the S-Box and additionally implement its inverse based on the proposed circuit in [5] using 120 ancillary qubits. This allows us to compute the output of the operation into new qubits and then using the adjoint inverse S-Box to reset the input qubits. In contrast to the implementation in [17], this allows us to recursively apply AES multiple times without needing additional qubits for every application, at the cost of additional quantum gates required. The *ShiftRows* step swaps qubits, thus we simply apply all following gates to different qubits (resulting in no additional cost). The *MixColumns* operation is the same implementation as the one by [17]. The *AddKey* operation is implemented using classically controlled NOT gates, as we use classical AES round keys, whereas [17] use quantum round keys. Fig. 4a shows the complete circuit for the AES encryption.

⁶ <https://github.com/RobinBerger/Grover-Sphincs>

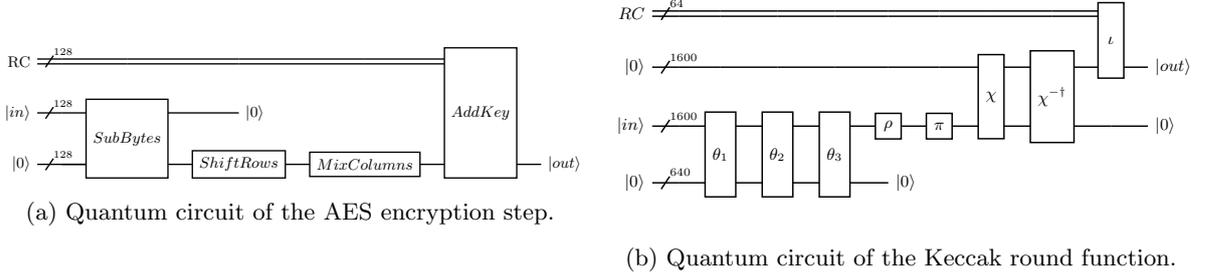


Fig. 4: Implementation of the round function components of Haraka and SHAKE-256.

Table 1: Resources for our implementation of the Haraka512 permutation and hash function. The width of the circuit includes the input and output qubits.

	T	CNOT	QubitClifford	T-Depth	Width
Permutation	609 289	1 383 040	189 440	69 125	1144
Hash Function	1 218 560	2 767 616	378 880	138 250	1912

Similarly to the *ShiftRows* operation, we implemented the mixing step for the Haraka permutation by redirecting the quantum wires.

The AES encryption operation computes the output into a new set of qubits, freeing up the input qubits. We apply this twice on each input block, alternating the input and output qubits, followed by the mixing step. This completes the round function that is repeated a total of 6 times for the Haraka512 permutation.

We implement the Haraka512 hash function by *copying* the input into ancillary qubits using CNOT gates, then applying the Haraka512 permutation on these qubits. Next, the relevant qubits from the output of the permutation and the input of the hash function are XORed into the output qubits using CNOT gates. Finally, the ancilla qubits are freed up again by applying the adjoint Haraka512 permutation. The Haraka-based sponge construction is implemented by instantiating a sponge construction with the Haraka512 permutation.

The quantum gate count for our implementations of the Haraka permutation and hash function can be seen in Table 1. Note that Q# optimizes the width of the quantum circuit, reusing ancillary qubits whenever possible, even if this results in a significantly higher depth of the quantum circuit. As the exact amount of quantum gates required depends on the SPHINCS⁺ instance, all round constants for determining the gate count here and in the rest of this work are assumed to be zero. When using the default round constants, 2582 additional NOT gates are required for every application of the Haraka512 permutation, which is negligible compared to the gates required for the rest of the implementation.

<p>Algorithm 5: KeccakPermutation($A[x][y][z] : \{0,1\}, 0 \leq x, y < 5, 0 \leq z < 64$)</p>
<pre> 1 for $0 \leq i < 24$ do 2 $A := \iota((\chi \circ \pi \circ \rho \circ \theta)(A), i)$ 3 return A </pre>

3.2 SHAKE-256

The SHAKE-256 hash function, as specified in [22], consists of the Keccak permutation, which is used to instantiate a sponge construction. The Keccak permutation consists of iterating the steps θ , ρ , π , χ and ι 24 times. The complete permutation is described in Algorithm 5, where the five steps are defined as

$$\begin{aligned}
\theta : \quad C[x][z] &:= \bigoplus_{0 \leq j < 5} A[x][j][z] \\
D[x][z] &:= C[x-1][z] \oplus C[x+1][z-1] \\
A'[x][y][z] &:= A[x][y][z] \oplus D[x][z] \\
\rho : \quad A'[x][y][z] &:= A[x][y][z + c[x][y]] \\
\pi : \quad A'[x][y][z] &:= A[x + 3y][x][z] \\
\chi : \quad A'[x][y][z] &:= A[x][y][z] \oplus ((A[x+1][y][z] \oplus 1) \cdot A[x+2][y][z]) \\
\iota : \quad A'[x][y][z] &:= \begin{cases} A[x][y][z] \oplus RC_i[z] & x = 0 \wedge y = 0 \\ A[x][y][z] & \text{otherwise} \end{cases} .
\end{aligned}$$

We note that our implementation follows closely the definition in [22] and thus has a similar structure to the one used by [1]. The operation θ is split into three parts $\theta_{1,2,3}$. θ_1 and θ_2 are a straight forward implementation of the SHA-3 specification, where we compute intermediate values in step θ_1 which are used in θ_2 to compute the output of the θ step. θ_3 implements θ^{-1} to uncompute intermediate values and is based on the KeccakTools reference implementation[2]. All XOR operations are implemented using CNOT gates. ρ and π are permuting the input and output bits by adjusting the subsequent quantum wires. The χ step of the Keccak permutation is a straight forward implementation of the specification with binary addition and multiplication based on CNOT and Toffoli gates, χ^{-1} is the respective inverse, where the adjoint χ^{-1} uncomputes the input qubits. This is the design also used by [1]. The ι step XORs a round constant on the state, which is implemented using classically controlled NOT gates.

The padding for the sponge construction is implemented using classically controlled NOT gates on the state.

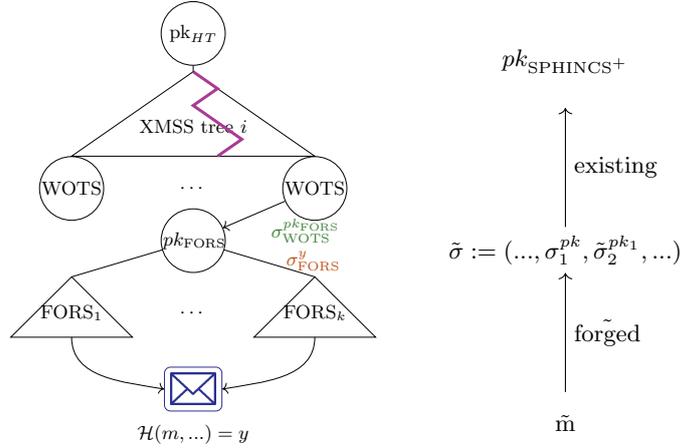
The quantum circuit for the round function is represented in Fig. 4b. The Keccak permutation consists of applying this implementation 24 times while alternating input and output qubits.

The quantum gates for our implementation of the Keccak permutation and a comparison with [1] can be seen in Table 2. The most notable differences are

Table 2: Quantum gate count for our implementation of the Keccak permutation and for the work by [1]. Gate counts for θ and χ are given for one round. Gate counts for ι , and the complete Keccak permutation are given for all 24 rounds.

	step	T	CNOT	QubitClifford	T-Depth	Width
Our implementation	$\theta_{1,2,3}$	0	63 040	0	0	2240
	χ	11 200	19 200	3200	25	3200
	χ^{-1}	13 440	23 360	3840	30	3200
	ι	0	0	86	0	1600
	Keccak	591 360	2 534 400	169 046	1176	3200
Implementation in [1]	Keccak	591 360	33 269 760	169 045	792	3200
	optimized	499 200	34 260 480	169 045	432	3200

that we use more than an order of magnitude fewer CNOT gates, because we use ancilla qubits for the θ operation and that we use the T-depth 5 Toffoli gate provided by Q# while [1] use a T-depth 3 Toffoli gate. More details on applying Grover for a pre-image attack are presented in Appendix A.



(a) Simplified hypertree structure showing the positions of the distinct signatures but excluding the vast majority of the XMSS hypertree. (b) Generic structure of a forged SPHINCS⁺ signature evaluating to a valid public-key.

Fig. 5: Simplified SPHINCS⁺ structure for forging a signature.

4 Attacking the SPHINCS⁺ Signature Scheme

We analyzed the WOTS, FORS and XMSS components of the SPHINCS⁺ scheme to identify weak points and then compared the required resources to mount an attack. Our attacks all admit the same structure as in Fig. 5b, where a set of forged sub-signatures evaluate to an *existing, valid* public key: One or multiple sub-signatures $\tilde{\sigma}_i^{pk_i}$ are generated from scratch using the canonical *Sign* algorithm (Algorithm 2). Then a single signature is forged to connect the generated sub-signatures with the existing public key. The generated and forged signature will be marked with the $\tilde{\sigma}$ accordingly.

Briefly speaking, we determined that forging an XMSS signature path requires the fewest logical resources to forge a complete SPHINCS⁺ signature when considering Grover like pre-image attacks. In the following we describe attacks on the XMSS and WOTS components in more detail. The procedure to forge signatures based on the message digest (B.1) and the FORS component (B.2) are covered in the appendix.

4.1 Forging a SPHINCS⁺ Signature on the XMSS-component

To compute a universal forgery for a signature of a message \tilde{m} , we create a new SPHINCS⁺ instance associated with a secret key $\tilde{sk}_{\text{SPHINCS}^+}$. The root node of the topmost XMSS instance of our new hypertree evaluates to the original public key pk_{SPHINCS^+} as in Fig. 6.

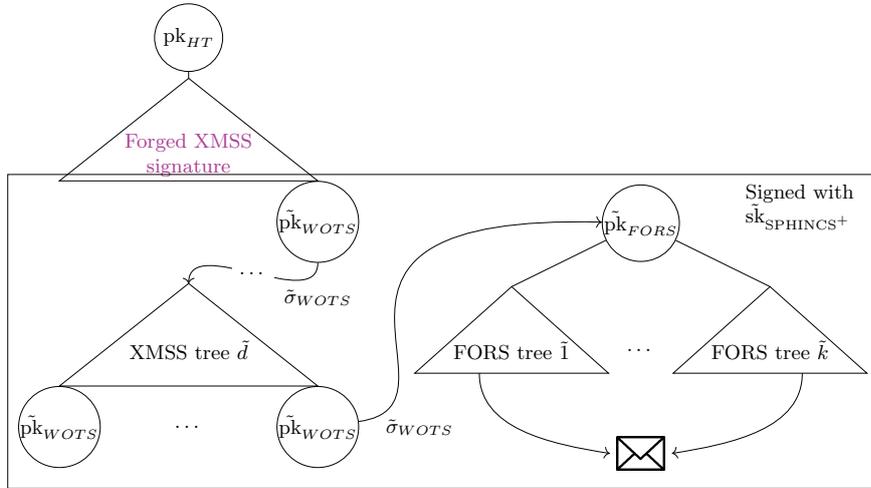


Fig. 6: Forged SPHINCS⁺ signature using a forged XMSS signature.

Let $\tilde{\sigma}_{\text{SPHINCS}^+}^{\tilde{m}} := (\tilde{\sigma}_{\text{FORs}}^{\tilde{m}}, \tilde{\sigma}_{\text{HT}}^{\text{pk}_{\text{FORs}}})$ be this new signature. The FORS signature is a freshly generated signature, the validation of which depends only

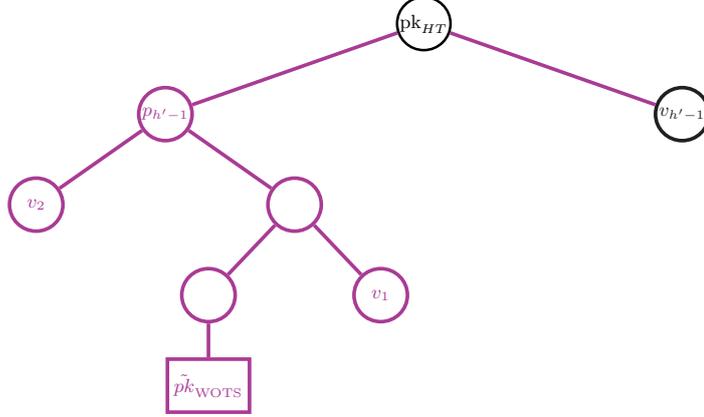


Fig. 7: Structure of a forged XMSS signature.

on the new key pair. To make this a valid signature, we modify the $\tilde{\sigma}_{XMSS}$ in the topmost layer of the hypertree signature $\tilde{\sigma}_{HT}^{pk_{HT}}$. In that layer, we use the public key of the WOTS instance generated from $\tilde{sk}_{SPHINCS^+}$ and replace the respective XMSS authentication path with a forged one path_{XMSS} obtaining a new XMSS signature $\tilde{\sigma}_{XMSS}$. Therefore, we need to find an authentication path $\tilde{\text{path}}_{XMSS}$ for $\tilde{\sigma}_{XMSS}$, so that computing the root node of the tree along the path with the respective WOTS public key results in the given XMSS public key as in Fig. 7.

Let $p_1 \dots p_{h'}$ be the nodes on the path from the given WOTS public key node to the root of the XMSS tree with p_1 being the leaf and $p_{h'}$ being the root node. Also, let $v_1 \dots v_{h'-1}$ be the respective sibling nodes. p_1 is the WOTS public key and p_i is computed from p_{i-1} and v_{i-1} for $i > 0$.

To find values v_i for an authentication path, we select the first $h' - 2$ values $v_1 \dots v_{h'-2}$ at random from $\{0, 1\}^n$. This results in fixed values $p_1 \dots p_{h'-1}$. Then we can forge the authentication path $\tilde{\text{path}}_{XMSS}$ if we can find a value $v_{h'-1}$ to complete the path. This can be seen in Fig. 7. We can estimate the probability of such a pre-image $v_{h'-1}$ existing for a fixed $v_1 \dots v_{h'-2}$ and a given public key, if we assume that the deployed hash function behaves like a random oracle, i.e. with each value $\mathcal{H}(x)$ being chosen uniformly at random independently from each other:

$$\begin{aligned} \mathbb{P}(\exists x \in \{0, 1\}^n : \mathcal{H}(x) = pk) &= 1 - \mathbb{P}(\forall x \in \{0, 1\}^n : \mathcal{H}(x) \neq pk) \\ &\geq 1 - \frac{1}{e} \end{aligned} \quad (2)$$

This means that a pre-image $v_{h'-1}$ exists with probability $\geq 1 - 1/e$. Therefore, forging a valid signature for a message depends only on finding the value $v_{h'-1}$.

Table 3: Gate count for our implementation of the Grover components in one Grover iteration.

	T	CNOT	QubitClifford	T-Depth	Width
SPHINCS ⁺ -128-Haraka	2 438 891	5 535 202	758 282	275 713	1400
SPHINCS ⁺ -128-SHAKE-256	1 184 491	5 071 842	338 614	3635	3456
SPHINCS ⁺ -256-Haraka	2 440 683	5 538 274	758 794	276 865	1656
SPHINCS ⁺ -256-SHAKE-256	1 186 283	5 076 450	339 126	4787	3712
Grover Diffusion (128 bit)	1771	2530	1022	1139	–
Grover Diffusion (256 bit)	3563	5090	2046	2291	–

In the remaining paper we are concerned with estimating the resources to find this value using Grover’s algorithm on a fault-tolerant quantum computer.

While this attack can be modified by generating WOTS instances for one half of the attacked XMSS instance, allowing to easily forge signatures for multiple messages if they fall on that side of the XMSS tree, the setup and the cost for the pre-image search is the same, so we will not go into more detail with this.

Resource Estimate To forge the XMSS signature, we need to find a pre-image of the Haraka-based sponge or the SHAKE-256 hash function using Grover’s algorithm. In the following estimate, let n be the security parameter in bits.

For the Haraka instantiation, the input to the hash function consists of a 256 bit address and two n -bit values, one of which is the hash value of a node in the XMSS tree, the other one is the value searched for by Grover’s algorithm to forge the signature. For the SHAKE-256 instantiation, the input to the hash function consists of a n -bit public key seed and the same inputs as with Haraka.

Using $n = 128$ for the Haraka instantiation, we can save resources by precomputing one iteration of the Haraka512 permutation. As the rate of the sponge instantiation is 256 bits, the first iteration absorbing the address can always be precomputed, so the quantum circuit is implemented using a different initial state, skipping this iteration. Using the same security parameter for the SHAKE-256 instantiation, none of the iterations can be precomputed. The gate count for the implementation of these Grover oracles as well as for the Grover diffusion operator for the SPHINCS⁺-128 and SPHINCS⁺-256 parameter sets as determined by Q# are shown in Table 3. While we include the 256-bit parameter sets for comparison, we want to note that for the Haraka hash function, more efficient attacks exist for that parameter set.

For $n = 128$, Grover’s algorithm requires roughly $1.6 \cdot 2^{63}$ iterations. Combining these gate counts with the amount of Grover iterations, we can evaluate two cost metrics for this attack. These results are shown in Table 4.

We can see that the attack using the SHAKE-256 hash function performs better on both cost metrics than the attack using Haraka. This results from the additional iterations of the Haraka permutation compared to SHAKE-256.

Table 4: Resource estimate for a pre-image search to forge an XMSS signature, where the target column indicates if the left or right node of the hash tree is attacked.

SPHINCS ⁺ instantiation	Gate count	T-Depth	T-Depth-Times-Width
SPHINCS ⁺ -128-Haraka	$1.6 \cdot 2^{86}$	$1.7 \cdot 2^{81}$	$1.1 \cdot 2^{92}$
SPHINCS ⁺ -128-SHAKE-256	$1.2 \cdot 2^{86}$	$1.8 \cdot 2^{75}$	$1.5 \cdot 2^{87}$
SPHINCS ⁺ -256-Haraka	$1.6 \cdot 2^{150}$	$1.7 \cdot 2^{145}$	$1.4 \cdot 2^{156}$
SPHINCS ⁺ -256-SHAKE-256	$1.2 \cdot 2^{150}$	$1.4 \cdot 2^{140}$	$1.2 \cdot 2^{152}$

4.2 Forging a SPHINCS⁺ Signature on the WOTS component

An alternate approach to forging SPHINCS⁺ signatures is to attack the WOTS component. Similarly to the previous attack, this is a universal forgery attack, however we also require a message m , that already has a valid signature $\sigma_{\text{SPHINCS}^+}$.

The general attack strategy is similar to [9], i.e. the selection of the WOTS instance and the construction of the SPHINCS⁺ signature from the other components: We generate a SPHINCS⁺ signature for a new message using a new secret key, making sure that this signature uses the first-layer WOTS instance at the same position as the one in $\sigma_{\text{SPHINCS}^+}$. We then forge a WOTS signature, that authenticates our second-layer XMSS public key for the first-layer WOTS public key in the original structure. In comparison to [9], who use a fault injection attack, we forge the WOTS signature using a quantum pre-image attack.

Custom Selection of WOTS instances Similarly to [9], when creating a SPHINCS⁺ signature $\tilde{\sigma}_{\text{SPHINCS}^+}$ for a message \tilde{m} , we need to use a FORS instance, that results in $\tilde{\sigma}_{\text{SPHINCS}^+}$ using the first-layer WOTS instance at the same position as $\sigma_{\text{SPHINCS}^+}$ does. They state that this is possible on a classical computer with feasible effort. In the setting of SPHINCS⁺-128 [16], this takes an average of $\approx 2^9$ hash function invocations.

Forging WOTS Signatures As the WOTS signature scheme divides a message into message and checksum blocks and then signs each block individually, forging a WOTS signature requires forging a signature for each block. A signature for a block containing a message m_i consists of the m_i -th element of a hash chain as shown in Fig. 8.

Let m_i and \tilde{m}_i be the message in the i -th block of m and \tilde{m} respectively. For $m_i \leq \tilde{m}_i$, a signature for block i can be computed, by advancing in the hash chain $\tilde{m}_i - m_i$ times by applying the hash function. For $m_i > \tilde{m}_i$, we need to go back $m_i - \tilde{m}_i$ times in the hash chain. To do this, we can apply Grover’s algorithm.

Such a pre-image to a value of the hash chain must exist, as the original signature σ_{FORS} was generated using this value. As it might not be unique,

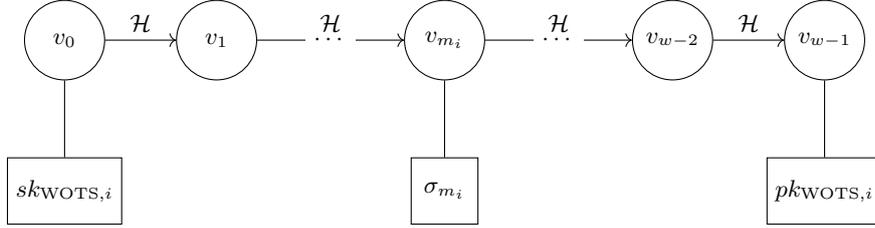


Fig. 8: Structure of one hash chain of a WOTS instance. $sk_{WOTS,i}$, $pk_{WOTS,i}$, m_i and σ_{m_i} are the secret key, public key, message and signature of the i -th block of the WOTS scheme with block length $\log_2 w$.

only the value used to generate σ_{FORS} is guaranteed to have a pre-image again, this means that instead of applying Grover's algorithm multiple times, to go back in the hash chain once in each step, we need to do a pre-image search on a recursive application of the hash function in a single step. If multiple pre-images of the recursive application of the hash function exist, any one of them produces a valid signature for that block.

Let the length of each block be $\log_2 w$. As the messages for the WOTS signature scheme are outputs of a hash function and therefore the message blocks m_i and \tilde{m}_i are blocks of an output of a hash function, it is reasonable to assume, that they are distributed uniformly at random from the set $\{0, \dots, w-1\}$, independently from each other. However we cannot assume this assumption to hold for the checksum blocks. Using this assumption for the message blocks, we can estimate the recursion depth of the hash function required for this attack. We will follow a simple approach, only considering a single pre-image search for a message block, neglecting the amount of pre-image searches required for checksum blocks and the possibility of searching for weak instances. A more detailed approach also considering the aforementioned aspects is beyond the scope of this work.

For the recursion depth required for a pre-image search we take the value d , so the probability of a recursion depth of $\geq d$ and $\leq d$ being required for the pre-image search is $\geq 1/2$. For the SPHINCS⁺ parameters proposed in [16] with $\log w = 4$, this results in $d = 5$.

Resource Estimate For forging a WOTS signature and carrying out this attack, we need to do multiple pre-image searches of a recursive application of the SHAKE-256 or Haraka512 hash function. Let n be the security parameter in bits in the following resource estimate.

For the Haraka instantiation, the input to the hash function consists of a 256 bit address and the n bit value searched for. The SHAKE-256 instantiation additionally gets an n bit public key seed as input.

Using $n = 128$, we will only go into detail for a hash function recursion depth of 5, as calculated previously. The gate count required for the Grover oracles for

Table 5: Gate count required by our implementation of the Grover oracles for a recursion depth 5 of the hash functions.

	T	CNOT	QubitClifford	T-Depth	Width
SPHINCS ⁺ -128-Haraka	12 187 371	27 667 810	3 789 310	1 369 814	1912
SPHINCS ⁺ -128-SHAKE-256	11 828 971	50 694 370	3 381 550	26 098	3968
SPHINCS ⁺ -256-Haraka	12 189 163	27 674 850	3 789 822	1 357 142	2680
SPHINCS ⁺ -256-SHAKE-256	11 830 763	50 700 770	3 382 062	27 250	4736
Grover Diffusion (128 bit)	1771	2530	1022	1139	–
Grover Diffusion (256 bit)	3563	5090	2046	2291	–

Table 6: Resource estimate for a pre-image search to forge a WOTS signature.

SPHINCS ⁺ instantiation	Gate count	T-Depth	T-Depth-Times-Width
SPHINCS ⁺ -128-Haraka	$1.0 \cdot 2^{89}$	$1.0 \cdot 2^{84}$	$1.9 \cdot 2^{94}$
SPHINCS ⁺ -128-SHAKE-256	$1.5 \cdot 2^{89}$	$1.3 \cdot 2^{78}$	$1.3 \cdot 2^{90}$
SPHINCS ⁺ -256-Haraka	$1.0 \cdot 2^{153}$	$1.0 \cdot 2^{116}$	$1.3 \cdot 2^{159}$
SPHINCS ⁺ -256-SHAKE-256	$1.5 \cdot 2^{153}$	$1.4 \cdot 10^{142}$	$1.6 \cdot 2^{154}$

this attack for both of the hash functions and for the diffusion operator are shown in Table 5.

As with the previous attack, for $n = 128$, $\approx 1.6 \cdot 2^{63}$ Grover iterations are required for one pre-image attack. We can again combine this with the gate counts from Table 5 to evaluate the cost metrics for this attack. This is shown in Table 6. As mentioned previously, the cost metric does not capture that this attack requires multiple pre-image attacks of variable recursion depths.

As the Haraka512 hash function is used here and not the Haraka-Sponge hash function used in the previous attack, the amount of applications of the underlying permutation is the same for the Haraka and SHAKE-256 instantiation, with the Haraka permutation requiring fewer quantum gates explaining the results of the gate count metric. As in the previous attack, Haraka performs worse in the T-Depth-Times-Width metric, as the Haraka permutation has a significantly higher T-Depth.

5 Fault-tolerant Cost

In this section, we give tight cost estimates of carrying out the *most promising* attack on XMSS signatures in Section 4.1. In particular, we analyze the resource requirements for the SPHINCS⁺-128 parameter sets, i.e. Haraka and SHAKE-256 hash function. A comparison of all results can be found in Table 7. The analysis follows the approach by [1], but optimizes the parallelization of the magic state distillation.

(a) Distance of the error correcting code d_i and number of logical and physical qubits for each layer i .

Layer	i	d_i	Q_i^{log}	Q_i^{phy}
Top	1	19	16	12800
Bottom	2	9	240	48000

(b) Pipelining the production of 3 magic states allows to reuse the qubits from the bottom layer in the top layer.

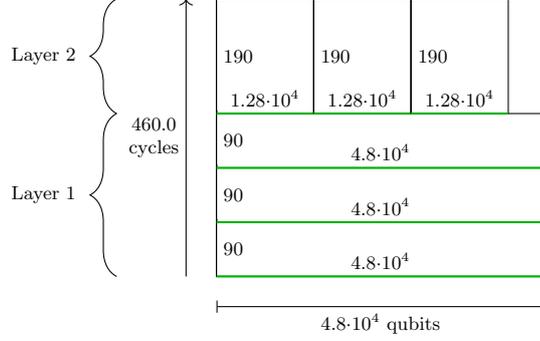


Fig. 9: Magic state distillation scheme for attacking SPHINCS⁺-128.

5.1 Haraka

Setup The entire Grover circuit for the attack using the Haraka hash function consists of $T_{\text{Haraka}} = 3.54 \cdot 10^{25}$ T-gates, $G_{\text{Haraka}}^{\text{cnot}} = 8.02 \cdot 10^{25}$ CNOT-gates, and $G_{\text{Haraka}}^c = 1.1 \cdot 10^{25}$ QubitClifford-gates, with the Hadamard-gates dominating the QubitClifford gates, thus other types of gates are ignored. The circuit has a width of $Q_{\text{Haraka}}^w = 1400$ and a T-depth of $T_{\text{Haraka}}^d = 4.01 \cdot 10^{24}$.

Magic State Distillation Given the desired output error rate relative to the size of the circuit $p_{\text{out}} = 1/T_{\text{Haraka}}^d$ and the assumptions given in Section 2.4 one can determine the number of layers of magic distillation required. We require two layers as in Fig. 9a, each with a surface code distance d_i , number of logical Q_i^{log} and respectively physical qubits Q_i^{phy} . In total, the number of logical qubits for a single distillery is $Q_{\text{MD, Haraka}}^{\text{log}} = 240$.

The layers can be optimized based on the cost metrics from Section 2.4, i.e. $\text{COST}_{\text{IQC}} = \text{COST}_{\text{SCC}} \cdot (Q_G + Q_{\text{MD}})$ and $\text{COST}_{\text{SCC}} = \text{SCC} \cdot G^d$, thus increasing the number of cycles scales the cost by both, cycles and qubits. Consider an increase of cycles by a factor X and an increase of qubits by a factor Y . Then the optimal distillery can be found by computing $\min_{X,Y} XQ_G + XYQ_{\text{MD}}$.

Surface Code The gates in the circuit are embedded into a surface code of distance $D_{G, \text{Haraka}} = 25$, with $p_{\text{out}} = 1/(G_{\text{Haraka}}^{\text{cnot}} + G_{\text{Haraka}}^c)$ as targeted error rate. This results in each of the $Q_{G, \text{Haraka}}^{\text{log}} = 1400$ logical qubits to require 1352 physical qubits. In total, the algorithm requires $Q_{G, \text{Haraka}}^{\text{phy}} \approx 1.89 \cdot 10^6$ physical qubits.

Results On average, about 9 T-gates are applied in each layer of T-depth. The number of physical qubits is dominated by the surface code, therefore we suggest to compute all magic states in parallel using 3 magic state distilleries and $Q_{MD}^{\text{phy}} = 3 \cdot 48000 = 1.6 \cdot 10^4$ physical qubits in $\text{SCC}_{\text{Haraka}}^m = 460$ cycles.

The average number of gates per layer of T-depth for each CNOT

$$G_{\text{Haraka}}^{\text{cnot}} / Q_{\text{Haraka}}^w \cdot T_{\text{Haraka}}^d \approx 0.0143$$

and each and QubitClifford gate

$$G_{\text{Haraka}}^c / Q_{\text{Haraka}}^w \cdot T_{\text{Haraka}}^d \approx 0.002$$

is significantly smaller than the number of surface code cycles required to implement a single layer required for magic state distillation.

Therefore, the total number of surface code cycles for the entire algorithm is dominated by the magic state distilleries, which is

$$\text{COST}_{\text{SCC}} = \text{SCC}_{\text{Haraka}}^m \cdot T_{\text{Haraka}}^d = 460 \cdot 4.01 \cdot 10^{24} \approx 1.5 \cdot 2^{90}.$$

The total number of logical qubits required is 2120. With 200ns per surface code cycle, this would take $1.17 \cdot 10^{13}$ years. The total cost of running the attack is then

$$\text{COST}_{\text{IqcHaraka}} = \text{COST}_{\text{SCC}} \cdot (1400 + 3 \cdot 240) = 1.5 \cdot 2^{90} \cdot (2120) \approx 3.91 \cdot 10^{30} \approx 1.55 \cdot 2^{101}.$$

5.2 SHAKE-256

Setup When using the SHAKE-256 hash function, our quantum circuit for the entire Grover algorithm for the attack contains $G_{\text{SHAKE-256}}^T = 1.72 \cdot 10^{25}$ T-gates and $G_{\text{SHAKE-256}}^{\text{cnot}} = 7.35 \cdot 10^{25}$ CNOT-gates. It also contains $G_{\text{SHAKE-256}}^c = 4.92 \cdot 10^{24}$ QubitClifford gates, most of which are Hadamard-gates. We will ignore any QubitClifford gates, that are not Hadamard-gates. The quantum circuit has a logical width of $Q_{\text{SHAKE-256}}^w = 3456$ qubits and a T-Depth of $T_{\text{SHAKE-256}}^d = 6.92 \cdot 10^{22}$.

Magic State Distillation The number of layers and thus the values for magic state distillation are reminiscent to those of Section 5.1, in particular, of Fig. 9a.

Surface Code The distance of the surface code remains as $D_{G,\text{SHAKE-256}} = 25$, with the same targeted error rate. This results in each of the $Q_{G,\text{SHAKE-256}}^{\text{log}} 3456$ logical qubits to require 1953 physical qubits. In total the algorithm requires $Q_{G,\text{SHAKE-256}}^{\text{phy}} \approx 6.75 \cdot 10^6$ physical qubits.

Results On average, about 249 T-gates are applied in each layer of T-depth. Therefore, we suggest to use 83 distilleries each generating 3 states in parallel, using a total of $Q_{MD}^{\text{phy}} = 83 \cdot 48000 = 3.98 \cdot 10^6$ physical qubits in $\text{SCC}_{\text{SHAKE-256}}^m = 460$ cycles.

Table 7: Fault-tolerant cost for our attack from Section 4.1 using the SHAKE-256 and Haraka hash functions. The collision attack of [10] refers to attacking the internal state of Haraka.

SPHINCS ⁺ -		SHAKE-256		Haraka
Collision Attack [10]	#Grover Iterations		–	$1.32 \cdot 2^{102}$
	Time-Space Product		–	$1.51 \cdot 2^{153}$
	#Classical hash function invocations		–	$2^{129.5}$
Our Attack on 128	#Distilleries	ϕ	83×3	3×3
	#Log. Qubits	Q^{log}	23876	2120
	#Total Phys. Qubits	Q^{phy}	$8.65 \cdot 10^6$	$2.03 \cdot 10^6$
	#Total ECC cycles	COST _{SCC}	$1.6 \cdot 2^{84}$	$1.5 \cdot 2^{90}$
	logical-qubit-cycles	COST _{lqc}	$1.17 \cdot 2^{99}$	$1.55 \cdot 2^{101}$
Our Attack on 256	#Distilleries	ϕ	42×4	9×1
	#Log. Qubits	Q^{log}	$1.7 \cdot 10^5$	$0.38 \cdot 10^5$
	#Total Phys. Qubits	Q^{phy}	$5.8 \cdot 10^7$	$1.5 \cdot 10^7$
	#Total ECC cycles	COST _{SCC}	$1.02 \cdot 2^{152}$	$3.95 \cdot 2^{154}$
	logical-qubit-cycles	COST _{lqc}	$1.31 \cdot 2^{169}$	$1.44 \cdot 2^{171}$

The average number of gates per layer of T-depth for each CNOT

$$g_{\text{SHAKE-256}}^{\text{cnot}} / q_{\text{SHAKE-256}}^w \cdot T_{\text{SHAKE-256}}^d \approx 0.31$$

and each and QubitClifford gate

$$g_{\text{SHAKE-256}}^c / q_{\text{SHAKE-256}}^w \cdot T_{\text{SHAKE-256}}^d \approx 0.021.$$

Again, magic state distillation dominates, resulting in a total number of

$$\text{COST}_{\text{SCC}} = \text{SCC}_{\text{SHAKE-256}}^m \cdot T_{\text{SHAKE-256}}^d = 460 \cdot 6.92 \cdot 10^{22} \approx 1.6 \cdot 2^{84}$$

surface code cycles. The total number of logical qubits required is 23876. With $200ns$ per surface code cycle, this would take $2.02 \cdot 10^{11}$ years. The total cost of running the attack is then

$$\text{COST}_{\text{lqcSHAKE-256}} = \text{COST}_{\text{SCC}} \cdot (3456 + 83 \cdot 240) = 1.6 \cdot 2^{84} \cdot (23876) \approx 7.44 \cdot 10^{29} \approx 1.17 \cdot 2^{99}.$$

6 Conclusion

We presented quantum implementations for the Haraka (and respectively SHAKE-256) hash function in the context of the SPHINCS⁺ signature scheme.

Subsequently, we proposed and reviewed multiple points of attack in the SPHINCS⁺-128-Haraka signature scheme based on applying Grover’s algorithm to find pre-images. A tight estimate of the resources required to carry out the most promising attack on a fault tolerant quantum computer is given. Our attack, that forges a signature in $1.55 \cdot 2^{101}$ steps, improves over the previously best known attack on SPHINCS⁺-128-Haraka.

Following the suggestion by NIST to review the security in terms of a maximal depth for quantum circuits, it is clear that for a depth of 2^{96} the attack can be implemented without any further constraints and would be more efficient than the classical counter part. For a depth of 2^{40} and 2^{64} the overhead induced by error correction needs to be reevaluated and optimized to the respective depth. A detailed analysis is out of scope for this paper and left as future work.

Acknowledgements

This work was supported by funding of the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL) under project number 46.23.01 and 46.23.02. We thank the reviewers for the useful comments and remarks.

References

1. Amy, M., Di Matteo, O., Gheorghiu, V., Mosca, M., Parent, A., Schanck, J.: Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. In: Avanzi, R., Heys, H. (eds.) *Selected Areas in Cryptography – SAC 2016*. pp. 317–337. Springer International Publishing, Cham (2017)
2. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccaktools, <https://github.com/KeccakTeam/KeccakTools>
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic sponge functions (2011), https://keccak.team/sponge_duplex.html
4. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique cryptanalysis of the full AES. In: Lee, D.H., Wang, X. (eds.) *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security*, Seoul, South Korea, December 4–8, 2011. *Proceedings. Lecture Notes in Computer Science*, vol. 7073, pp. 344–371. Springer (2011). https://doi.org/10.1007/978-3-642-25385-0_19, https://doi.org/10.1007/978-3-642-25385-0_19
5. Boyar, J., Peralta, R.: A small depth-16 circuit for the AES S-Box (2012)
6. Boyer, M., Brassard, G., Høyer, P., Tapp, A.: Tight bounds on quantum searching. *Fortschritte der Physik* **46**(4-5), 493–505 (Jun 1998). [https://doi.org/10.1002/\(sici\)1521-3978\(199806\)46:4/5<493::aid-prop493;3.0.co;2-p](https://doi.org/10.1002/(sici)1521-3978(199806)46:4/5<493::aid-prop493;3.0.co;2-p), [http://dx.doi.org/10.1002/\(SICI\)1521-3978\(199806\)46:4/5<493::AID-PROP493>3.0.CO;2-P](http://dx.doi.org/10.1002/(SICI)1521-3978(199806)46:4/5<493::AID-PROP493>3.0.CO;2-P)
7. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: Lucchesi, C.L., Moura, A.V. (eds.) *LATIN’98: Theoretical Informatics*. pp. 163–169. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)

8. Bravyi, S., Kitaev, A.: Universal quantum computation with ideal clifford gates and noisy ancillas. *Phys. Rev. A* **71**, 022316 (Feb 2005). <https://doi.org/10.1103/PhysRevA.71.022316>
9. Castelnovi, L., Martinelli, A., Prest, T.: Grafting trees: A fault attack against the SPHINCS framework. In: Lange, T., Steinwandt, R. (eds.) *Post-Quantum Cryptography*. pp. 165–184. Springer International Publishing, Cham (2018)
10. Chailloux, A., Naya-Plasencia, M., Schrottenloher, A.: An efficient quantum collision search algorithm and implications on symmetric cryptography. In: Takagi, T., Peyrin, T. (eds.) *Advances in Cryptology – ASIACRYPT 2017*. pp. 211–240. Springer International Publishing, Cham (2017)
11. Dang, Q.H.: Secure hash standard (shs). National Institute for Standards and Technology (2008). <https://doi.org/10.6028/NIST.FIPS.180-4>
12. Fowler, A.G., Devitt, S.J., Jones, C.: Surface code implementation of block code state distillation. *Scientific Reports* **3**(1), 1939 (Jun 2013). <https://doi.org/10.1038/srep01939>
13. Fowler, A.G., Mariantoni, M., Martinis, J.M., Cleland, A.N.: Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A* **86**, 032324 (Sep 2012). <https://doi.org/10.1103/PhysRevA.86.032324>
14. Gidney, C., Ekerå, M.: How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* **5**, 433 (2021). <https://doi.org/10.22331/q-2021-04-15-433>, <https://doi.org/10.22331/q-2021-04-15-433>
15. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. p. 212–219. STOC '96, Association for Computing Machinery, New York, NY, USA (1996). <https://doi.org/10.1145/237814.237866>
16. Hulsing, A., Bernstein, D.J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Kampanakis, P., Kolbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Aumasson, J.P., Westerbaan, B., Beullens, W.: SPHINCS+-Submission to the 3rd round of the NIST post-quantum project (2020)
17. Jaques, S., Naehrig, M., Roetteler, M., Virdia, F.: Implementing grover oracles for quantum key search on AES and LowMC. In: Canteaut, A., Ishai, Y. (eds.) *Advances in Cryptology – EUROCRYPT 2020*. pp. 280–310. Springer International Publishing, Cham (2020)
18. Jaques, Samuel: Quantum Cost Models for Cryptanalysis of Isogenies. Master’s thesis, University of Waterloo (2019), <http://hdl.handle.net/10012/14612>
19. Jones, N.C., Van Meter, R., Fowler, A.G., McMahon, P.L., Kim, J., Ladd, T.D., Yamamoto, Y.: Layered architecture for quantum computing. *Phys. Rev. X* **2**, 031007 (Jul 2012). <https://doi.org/10.1103/PhysRevX.2.031007>
20. 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 (Feb 2017). <https://doi.org/10.13154/tosc.v2016.i2.1-29>
21. National Institute for Standards and Technology: Advanced Encryption Standard (AES) (2001). <https://doi.org/10.6028/NIST.FIPS.197>
22. National Institute for Standards and Technology: SHA-3 standard: Permutation-based hash and extendable-output functions (2015). <https://doi.org/10.6028/NIST.FIPS.202>
23. National Institute for Standards and Technology: Post-quantum cryptography call for proposals (2017), <https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf>

24. National Institute for Standards and Technology: Post-quantum cryptography round 3 (2020), <https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions>
25. Nielsen, M.A., Chuang, I.L.: Quantum computation and quantum information, chap. 2,4. Cambridge University Press, Cambridge New York (2010)
26. Roetteler, M., Naehrig, M., Svore, K.M., Lauter, K.: Quantum resource estimates for computing elliptic curve discrete logarithms. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017. pp. 241–270. Springer International Publishing, Cham (2017)

A pre-image Attack on the Hash Functions using Grover’s Algorithm

Grover The Grover iteration consists of evaluating the predicate on the input qubits and flipping their phase if the predicate evaluates to 1. The predicate first applies our implementation of the hash function \mathcal{H} and then applies a classically controlled NOT-gate on the output qubits of \mathcal{H} , followed by a multi-controlled NOT-gate to flip the phase of the target elements. We apply the adjoint implementation of the previous operations to uncompute any ancillary registers. The second step is the Grover diffusion operator.

The implementation for a Grover oracle for a pre-image search can be seen in Fig. 10. The last step of the Grover iteration is the diffusion operator, which is omitted from the description.

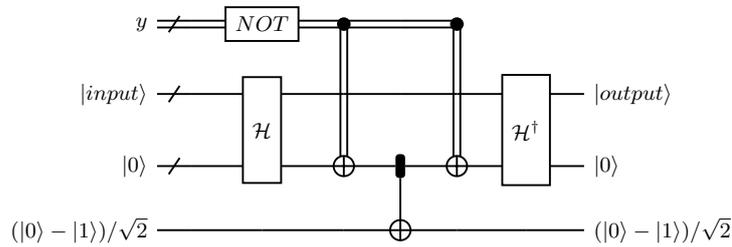


Fig. 10: Implementation of our Grover oracles for a pre-image search for $y = \mathcal{H}(x)$.

Pre-image Attack A pre-image attack on the hash functions is implemented by using Grover’s algorithm. The gate counts for the implementation of the Grover oracles for the SHAKE-256 and the Haraka-based sponge hash function for an input and output length of 128 and 256 bits, as well as the diffusion operator is shown in Table 8.

Combining this with the $\approx 1.6 \cdot 2^{63}$ Grover iterations, that are required for this, we can evaluate the gate count and T-Depth-Times-Width cost metrics

Table 8: Resource requirements for a Grover oracle for a pre-image attack on SHAKE-256 and the Haraka-based sponge hash function with different input and output lengths.

Hash function	length	T	CNOT	QubitClifford	T-Depth	Width
Haraka-Sponge	128	1 220 331	2 769 122	379 402	138 327	1400
SHAKE-256	128	1 184 491	5 071 842	338 614	3635	3456
Haraka-Sponge	256	2 440 683	2 769 122	758 794	276 865	1656
SHAKE-256	256	1 186 283	5 074 914	339 126	4787	3712
Grover diffusion	128	1771	2530	1022	1139	–
Grover diffusion	256	3563	5090	2046	2291	–

Table 9: Resource estimate for a generic pre-image search of SHAKE-256 and the Haraka-based sponge hash function with input and output length of 128 and 256 bit.

Hash function	length	Gate count	T-Depth	T-Depth-Times-Width
Haraka-Sponge	128	$1.6 \cdot 2^{85}$	$1.7 \cdot 2^{80}$	$1.1 \cdot 2^{91}$
SHAKE-256	128	$1.2 \cdot 2^{86}$	$1.8 \cdot 2^{75}$	$1.5 \cdot 2^{87}$
Haraka-Sponge	256	$1.6 \cdot 2^{150}$	$1.7 \cdot 2^{145}$	$1.4 \cdot 2^{156}$
SHAKE-256	256	$1.2 \cdot 2^{150}$	$1.2 \cdot 2^{152}$	$1.4 \cdot 2^{140}$

for finding a 128 bit pre-image for these two hash functions. The results for this can be seen in Table 9. While we include all these resource estimates for completeness, we want to empathize again, that more efficient attacks exist for Haraka in the 256-bit case.

B Further Attacks

In addition to the two attacks proposed in Section 4, we also considered an attack using a colliding message digest and an attack using forged FORS signatures.

B.1 Forging SPHINCS⁺ signatures using a colliding message digest

We only cover this attack for completeness. More efficient attacks exist both for SPHINCS⁺ and for finding a colliding message digest.

If we can find a new message, that results in the same primary message digest as a given message with a given valid signature, we can find a new message, where the same signature is valid.

Given a SPHINCS⁺ public key $\text{pk}_{\text{SPHINCS}^+}$, a message m with a valid signature $\sigma_{\text{SPHINCS}^+}$, we can find a new message \tilde{m} which results in the same primary message digest when verifying $\sigma_{\text{SPHINCS}^+}$ for \tilde{m} than when verifying $\sigma_{\text{SPHINCS}^+}$ for m . To do this, we first compute the primary message digest for

m and $\sigma_{\text{SPHINCS}^+}$. We can do this, as $m, \sigma_{\text{SPHINCS}^+}$ and $\text{pk}_{\text{SPHINCS}^+}$ contain everything required to do this. We can then apply Grover’s algorithm to the hash function used to compute the message digest, to find a new message \tilde{m} , that results in the same primary message digest.

While we do not know if such a pre-image exists, we can make a similar argument as in Section 4, if the length of \tilde{m} is the same as the length of the primary message digest. Assuming the hash function behaves like a random oracle, we can adapt Eq. (2) and have a lower bound of $1 - 1/e$ for the probability, that a pre-image exists. While it is possible to search for a message with length longer than the primary message digest to have a higher probability of a pre-image existing, this increases the size of the resulting quantum circuit. For the two SPHINCS⁺-128 parameter sets given in [16], the length of the primary message digest is 240 and 272 bits respectively.

When verifying the signature $\sigma_{\text{SPHINCS}^+}$ for \tilde{m} , first the primary message digest is computed. This message digest is equal to the primary message digest computed when verifying $\sigma_{\text{SPHINCS}^+}$ for m , because \tilde{m} was chosen, so these message digests are equal. All later computations, that are done to verify a signature only use the primary message digest and derived values, but not the message itself. This means that after computing the primary message digest, the signature verification for m and for \tilde{m} use the same values therefore returning the same result. As we assumed $\sigma_{\text{SPHINCS}^+}$ is valid for m , $\sigma_{\text{SPHINCS}^+}$ is also valid for \tilde{m} .

The success probability of this attack is equal to the success probability of being able to find a \tilde{m} that has the same primary message digest as m with $\sigma_{\text{SPHINCS}^+}$. We again consider the success probability of Grover’s algorithm to be 1. Therefore, the attack is successful if a pre-image exists for a given hash value, which is the case with probability $\geq 1 - 1/e$.

Resource estimate For finding a message, that has the same primary message digest, we need to do one pre-image search of the SHAKE-256 hash function or the Haraka-based sponge hash function using Grover’s algorithm. Let n be the security parameter and let N be the length of the primary message digest.

For the Haraka instantiation, the input to the hash function consists of the n bit value from $\sigma_{\text{SPHINCS}^+}$, a n bit root public key value and a N bit value for \tilde{m} , which is the message \tilde{m} we are searching for using Grover’s algorithm. For the SHAKE-256 instantiation, the input to the hash function consists of a n bit value from $\sigma_{\text{SPHINCS}^+}$, a n bit public key seed, a n bit root public key value and a N bit value.

For the two SPHINCS⁺-128 parameter sets given in [16], the primary message digest has a length 240 and 272 bits respectively. We will restrict ourselves to the SPHINCS⁺-128s parameter set with $N = 240$. In a similar analysis than the one for the forged XMSS signatures, we can find out that this attack using the Haraka hash function requires two applications of the Haraka512 permutation, but one of these can be precomputed resulting in a total of three applications of the permutations for computing the hash value and another three applications

for uncomputation. The attack using the SHAKE-256 hash functions requires one application of the Keccak permutation. The gate counts for implementing these Grover oracles can be seen in Table 10.

Table 10: Gate count required by our implementation of the Grover oracles and the diffusion operator.

	T	CNOT	QubitClifford	T-Depth	Width
SPHINCS ⁺ -128s-Haraka	1 221 899	2 771 810	379 850	139 335	1624
SPHINCS ⁺ -128s-SHAKE-256	1 186 059	5 074 530	339 062	4643	3680
Grover diffusion (240 bits)	3339	4770	1918	2147	–

For $N = 240$, Grover’s algorithm requires $\approx 1.6 \cdot 2^{119}$ iterations. Using this, we can evaluate the cost metrics. The results for this can be seen in Table 11.

Table 11: Resource estimate for a pre-image search for reusing an existing signature for a new message.

SPHINCS ⁺ instantiation	Gate count	T-Depth-Times-Width
SPHINCS ⁺ -128s-Haraka	$1.6 \cdot 2^{141}$	$1.3 \cdot 2^{147}$
SPHINCS ⁺ -128s-SHAKE-256	$1.2 \cdot 2^{142}$	$1.2 \cdot 2^{144}$

For the gate count metric, we can see that for both hash functions, the attack requires approximately the same amount of quantum gates. Looking at the depth-times-width cost metric, the attack using the Haraka hash function performs worse than the attack using the SHAKE-256. This is again caused by the implementation of the Haraka hash function having a significantly higher T-depth.

B.2 Forging SPHINCS⁺ signatures on the FORS component

Our fourth and final approach for forging SPHINCS⁺ signatures is to attack the FORS component. This is again a universal forgery attack and as with previous attacks, we require a message m with an existing signature $\sigma_{\text{SPHINCS}^+}$. As this attack has a lot of similarities as the attack on WOTS and XMSS, we will not go into much detail with this attack, but only give an outline of how it works.

FORS uses multiple hash trees, where the leaves of each tree are the hash of pseudorandomly generated secret values. Each hash tree signs a block of the message. The FORS public key is the hash of all root nodes. Let m_i be the value in the i -th block of the message, then the i -th hash tree signs m_i by revealing the m_i -th secret value in the hash tree along with the authentication path of

the corresponding leaf node. A signature for a block m_i is verified by using the secret value at position m_i in the hash tree along with the authentication path to recompute a candidate root value for the hash tree. This is then used to recompute a candidate FORS public key, which is verified against the real FORS public key.

If we are given a valid SPHINCS⁺ signature $\sigma_{\text{SPHINCS}^+}$ for a message m and we can forge a FORS signature for a different message digest also using the same FORS instance, we can reuse the hypertree signature σ_{HT} in $\sigma_{\text{SPHINCS}^+}$ for forging a SPHINCS⁺ signature for a different message \tilde{m} .

Custom selection of a FORS instance When forging a signature for a message \tilde{m} given m with its signature $\sigma_{\text{SPHINCS}^+}$, the attack only works, if the address computed from the message digest of \tilde{m} with our forged signature points to the same FORS instance as the address from m with $\sigma_{\text{SPHINCS}^+}$. We can do this with the same argument as in the attack on WOTS in Section 4.2. In contrast to that attack, the estimated amount of steps required for this custom selection of a FORS instance is between 2^{62} and 2^{66} , depending on the used parameter set. While this is a large amount of steps, as we will see later on, it is still by far less than the classical resources required for the quantum attack.

Forging a FORS signature To forge a FORS signature, we proceed as follows: We first select all secret values and values for the authentication paths at random, except for the topmost sibling-node in the authentication path of the last hash tree. This value is selected, so that when using it to compute a candidate public key for a given message results in the given WOTS public key. Again, we do this using Grover’s Algorithm.

This attack is comparable to the XMSS attack, as we apply Grover’s algorithm to the same node in a hash tree as in that attack. The difference is, however, that in this attack, we need to apply the hash function twice. In the first application of the hash function, we compute the root node r_k of the k -th hash tree and in the second application, we compute a FORS public key pk_{FORS} as depicted in Fig. 11. This means that we either need to do two pre-image searches or we need to do one pre-image search with a recursion depth of 2. In contrast to the WOTS attack in Section 4.2, where we knew that a pre-image exists, we do not know that in this case. Therefore both are valid approaches and we do not need to choose one over the other.

What we do however want to know is the probability that a pre-image exists. In previous attacks, we used Eq. (2) for this, but as we need to do two pre-image searches here (or a pre-image search on a recursive application of a hash function) cannot apply this equation directly. What we can however do is apply this equation for each application of the hash function. This means that for a matching value to exist, a pre-image has to exist and this pre-image needs to have a pre-image itself. This translates to a probability of $\geq (1 - 1/e)^2$ for a pre-image existing for the two applications. If we assume the success probability

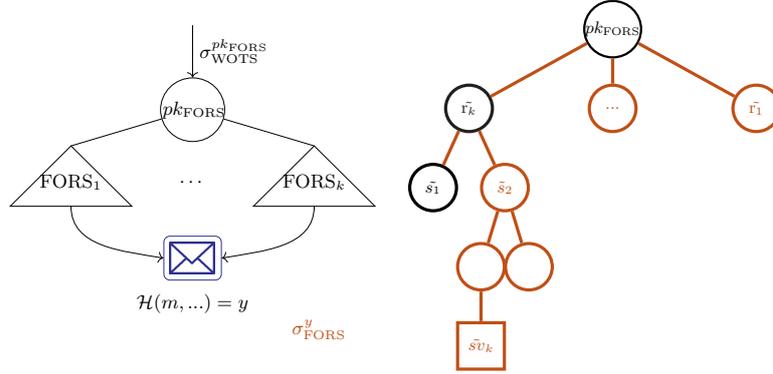


Fig. 11: Position of the forged FORS signature in the SPHINCS⁺ tree and of the forged values in the respective FORS tree. Note that the leaves of the tree depend on the forged message, thus the roots that are hashed into the pk_{FORS}^k cannot be chosen freely. The first node that can be used for a pre-image attack the sibling s_1 on a second level of a hash tree.

of Grover's algorithm to be 1, as in previous attacks, we can calculate the success probability for this attack to be $\geq (1 - 1/e)^2$.