Analysis and testing of SPHINCS algorithm

Report for the Computer System Security exam at the Politecnico di Torino

Giulia Milan (264976)
tutor: Ignazio Pedone

May 2020

Contents

1 Introduction

The idea of quantum computers was introduced by the physicist Richard Feynman in 1982. A quantum computer exploits quantum mechanics effects to its advantage [2]. For a period of time, this idea was solely of theoretical interest. In recent times post-quantum computing became an active area of study, also as a consequence of the Peter Shor's invention of an algorithm which claimed to be able to factor large numbers in polynomial time, assuming the availability of a quantum computer. Because of this, the topic of quantum computing has gained interest and many organizations around the world, such as Google and IBM, are racing to create practical quantum computers. Due to this race, there is an urgent need for developing new cryptographic algorithms that are able to resist a quantum attack. Indeed, existing algorithms, like RSA and DSA signature schemes, are based on mathematical primitives which can be broken by a quantum adversary. One of the main organizations involved in this process is the National Institute of Standards and Technology (NIST), which in 2016 opened the post-quantum crypto project, in order to select post-quantum cryptography (PQC) algorithms.

After an introduction on the main concepts of post-quantum cryptography and quantum computers we will have an overview on the principal types of PQC algorithms. Many different families of PQC algorithms exists: multivariate, lattice-based, code-based and hash-based algorithms. Among all of these existing families, in this work we are going to focus on hash-based algorithms, that are mainly used to implement digital signatures. They are interesting since their security relies only on the underlying hash function. Multiple hash-based signature algorithms exist. Here we will focus on a recent scheme called SPHINCS, which have been recently proposed to the NIST project and it is a promising candidate for a post-quantum secure digital signature scheme [6].

SPHINCS is a stateless digital signature algorithm. We will present the original implementation of it, along with the description of its main primitives, and then we will introduce SPHINCS+, the enhanced version of it. Later, we will focus on vulnerabilities and possible attacks against SPHINCS+.

In the last section of this paper we are going to evaluate SPHINCS+ performances, testing it in a real-world use-case scenario: it will be used as a mechanism of authentication in TLS protocol, exploiting the open source version of it, openSSL[1]. We will compare SPHINCS against current standard authentication mechanisms, which are realized through other digital signatures algortihms, such as RSA and DSA.

2 Post-quantum cryptography

In recent years, quantum computers have been an active area of research. Quantum computers are machines that exploit quantum mechanical phenomena in order to solve mathematical problems that are considered difficult or intractable for conventional computers. With a large-scale quantum computer, it will be possible to break many of the public-key cryptosystems currently used in our applications. This would be catastrophic, since it would compromise the confidentiality and integrity of all digital communications on the Internet [21].

This situation motivates the urgent use of a new cryptography. This new branch of cryptography is called post-quantum or quantum-safe or quantum-resistant cryptography and it must be designed to be safe against quantum attackers [12].

[1] OpenSSL: https://www.openssl.org/

For this reason, the goal of post-quantum cryptography (PQC) is to develop cryptographic systems that are secure against both a quantum and a classical computer, and can interoperate with existing communications protocols and networks [21].

In section 2.1 we will have an overview about PQC. Then we will present in a general way quantum computing and we are going to highlight why we need PQC, in subsection 2.2 and in subsection 2.3 respectively. Furthermore, we will present the NIST Post-quantum crypto project in subsection 2.4 and in the end, in subsection 2.5 we are going to discuss what are the main PQC algorithms.

2.1 Introduction

Cryptography is the study of secure communications techniques that transform data into formats that cannot be recognized by unauthorized users. Many different cryptographic algorithms exist, but they can be categorized into three main categories:

- Symmetric key cryptography: the encryption system relies on a shared secret between the two parties of a secure communication, the sender and the receiver. This secret is or can be used to generate the key of the symmetric algorithm.
- Asymmetric key criptography, also called public key cryptography: in this encryption system a pair of keys, a public and the corresponding private key, is used to perform cryptography. One of the keys is used to encrypt data while the other one is used to decrpyt them, or viceversa.
- Cryptographic hash functions: in their general definition, hash function does not need a key. They are used to map data of any size to fixed-size values. Those functions must be one-way function, that means that they are practically infeasible to invert.

The security of existing pulic-key cryptography relies on one of these mathematical problems: the integer factorization problem, the discrete logarithm problem or the elliptic-curve discrete logarithm problem. All those problems respect the computational hardness assumption: they are considered hard problems, that typically means that they can not be solved in a polynomial time. This was true until 1994 when the mathematician Peter Shor invented the so-called Shor's algorithm [17]. This is a polynomial-time quantum computer algorithm for integer factorization. The algorithm is significant because all of the three problems cited above can be solved in polynomial time on a sufficiently powerful quantum computer that runs Shor's algorithm [16]. This implies that public key cryptography might be easily broken, given a sufficiently large quantum computer.

For this reason, post-quantum cryptography has been introduced. Post-quantum cryptography studies new cryptography approaches that appear to be resistant to an attacker equipped with a quantum computer. The need of developing cryptosystems whose security relies on different, hard mathematical problems that are resistant to being solved by a large-scale quantum computer is real.

Although current experimental quantum computers lack processing power to break any real cryptographic algorithm, many cryptographers are designing new algorithms to prepare for a time when quantum computing becomes a threat.

Another reason for starting to design new criptographic algorithms is that we don't know when today's classic cryptography will be broken and most of all it's difficult and time-consuming to pull and replace existing cryptography from production software.

2.2 Quantum computing

As we saw before, current public key cryptography may be broken by a sufficiently powerful quantum computer. Computers that perform computation exploiting quantum-mechanical phenomena are known as quantum computers.

The idea of devices based on quantum mechanics was explored in the early 1980s by physicists and computer scientists such as Charles Bennet, Paul Benioff, David Deustch and Richard Feynman [18]. After Benioff proposed a quantum mechanical model of the Turing machine [40], Richard Feynman and Yuri Manin attempt to provide conceptually new kind of computers based on the principles of quantum physics.

Standard computers rely on the ability to store and manipulate information. They manipulate individual bits that store information in binary form, using the states 0 and 1. Instead, in a quantum computer the fundamental unit of information, which is called quantum bit or qubit, is not binary. A qubit can exist not only in a state corresponding to the logical state 0 or 1, as in a classical bit state, but also in states corresponding to a blend of superposition of those classical states [18]. In particular, three quantum mechanical properties are used in quantum computing to manipulate the state of a qubit, which are the following ones [41]:

1. *Superposition.* This is one of the fundamental principles of quantum mechanics exploited by qubits. It refers to a combination of states we would normally describe independently. In classical physics, a wave describing a musical note can be seen as many waves with different frequencies added together, superposed. Similarly, a quantum state in superposition can be seen as a linear combination of other quantum states. This quantum state obtained with superposition forms a new valid quantum state;

2. *Entanglement* is a quantum phenomenon not observable in the classical world, in which entangled particles behave together as a whole system. More in details, a pair of particles is entangled when the quantum state of each particle can not be described independently of the quantum state of the other particle - the entanglement applies both for pairs and groups of particles. While the quantum state of the system as a whole is well described, as it is in a definite state, the single parts of the system are not. When two qubits are entangled there exists a special connection between them in such a way that the outcome of the measurement on one qubit will always be correlated to the measurement on the other qubit. This is happens even if the particles are separated from each other by a large distance;

3. *Interference.* Finally, quantum states can undergo interference due to a phenomenon called phase. Quantum interference can be explained similarly to the wave interference phenomenon: when two waves are in phase, their amplitudes add, while, when they are out of phase, their amplitudes cancel.

So, whereas traditional computers are based on classical representations of the computational memory, quantum computing can transform memory into a superposition of multiple classical states [18]. In principle, following the previous statement, any computational problem that can be solved by a classical computer can also be solved by a quantum computer. Viceversa, according to [42], quantum computers obey to the Church-Turing thesis: any computational problem that can be solved by a quantum computer can also be solved by a classical computer. While this means that quantum computers provide no additional power over classical computers in terms of computability, they do provide additional power in terms of time and size complexity of solving certain problems, for which we don't have enough computational power to solve them [41]. This is the case of the previously mentioned Shor's algorithm.

This algorithm, invented in 1994 by Peter Shor, is a quantum algorithm able to factor integers. Later, this algorithm has been adapted to solve the discrete logarithm problem both in a finite field and in an elliptic curve group. Those problems provide security to the most used public key algortihms, RSA, DSA and ECDSA, respectively, which became vulnerable to this algorithm. In conclusion, one of the most used cryptographic protocols, RSA, is vulnerable to a quantum computer running the Shor's algorithm. This threat is not purely theoretical, since in 2001 a group at IBM factored 15 into 3x5 through Shor's algorithm on a quantum computer with 7 qubits [43]. After this, other independent groups run the Shor's algorithm on different implementations of quantum computers, e.g. based on photonic qubits [44] [45]. Later, in 2012, the factorization of 21 was achieved, which is the largest integer factored with Shor's algorithm [46]. Despite those factored values are still small, the invention of a quantum algorithm, capable of solving those problems in polynomial time, implemented on a real quantum computer makes the development of post-quantum cryptography strictly necessary.

2.3 The need of PQC

An impending need for deploying PQC exists. In this section we are going to focus on the main reasons which started the research on post-quantum cryptographic algorithms.

First of all, most of Internet security protocols rely on cryptography in order to provide security properties. All of those protocols, such as Transport Layer Security (TLS) protocol, have a similar basic structure: first, the communicating parties are authenticated to each other using public key cryptography in order to establish a shared secret key, which is then used in symmetric cryptography to create a communication that respects some security protocols, such as confidentiality and integrity.

Most public key cryptography algorithms rely on the assumption that the solution to their mathematical problems, such as factoring large numbers or computing discrete logarithms in finite field or elliptic curve groups, run in exponential or sub-exponential time. This means that it is infeasible for attackers to break those schemes.

As seen before, quantum mechanics allows for quantum computers, devices that operate on quantum bits, to solve certain types of problems much faster than classical computers. Shor's algorithm could efficiently - that means in polynomial time - factor large numbers and compute discrete logarithms, breaking all the most used public key cryptosystems [12].

For what concerns symmetric key schemes instead, such as Advanced Encryption Standard (AES), they would not be broken by quantum algorithms. However they would probably need bigger keys. While large-scale quantum computers do not yet exist, building quantum computers is an active area of research and this makes the deployment of PQC necessary [12].

From this it follows that also digital signatures security may be broken. All modern digital signature algorithms rely on hard mathematical problems:

- RSA (Rivest-Shamir-Adleman) algorithm relies on the hardness of integer factorization;
- DSA (Digital Signature Algorithm) relies on the discrete logarithm problem in a finite field, because it does not exist an efficient algorithm able to compute the discrete logarithm;
- ECDSA (Elliptic Curve Digital Signature Algorithm) is a variant of DSA using elliptic curve cryptography. It relies on the hardness of computing the discrete logarithm in an elliptic curve group.

All of those problems mentioned above can be solved in polynomial time with Shor's quantum algorithm, assuming the existance of a sizeable quantum computer [7]. This means that our systems of PKI, software signing, encrypted tunnels and so on would be vulnerable if quantum computer was available. Evidently, due to the recent adavances of quantum computing, the standardization of quantum-resistant public key algorithms is time-critical [7]: after a cryptographic algorithm is invented, it needs a lot of time to be used since it must be tested by mathematics and its security must be proved solid before starting to widely adopting it.

Indeed, post-quantum schemes, and signature schemes in particular, are mainly academic and don't have yet the kind of confidence that comes with an extended use in concrete scenarios. After their evaluation in real-world scenarios, they must be implemented and deployed in realistic contexts. This process has to be initiated as soon as possible, so those schemes can be trusted enough once large-scale quantum computers emerge [11].

For those reasons, in August 2016, the United States National Institute of Standards and Technology (NIST) has initiated a public project (post-quantum crypto project 1), a multiyear process to evaluate and standardize post-quantum public key encapsulation and signature mechanisms [12], as we will see in the following section.

It is important to notice that research into PQC is necessary even if large-scale quantum computes are never built: it's possible that solving the integer factorization or the discrete logarithm problems will be feasible one day with some non-quantum mathematical breakthroughs [12]. Having a different family of algorithms on which we can base public key cryptography protects us agains this possible scenario, giving us the ability of responding quickly to unexpected weaknesses in standard cryptographic algorithms.

2.4 NIST post-quantum crypto project

The interest in deploying post-quantum cryptographic algorithms is growing due to the desire to compete against the future possibility of a large-scale quantum computer.

The recent advances and attention to quantum computing have raised serious security concerns among IT professionals. The ability of a quantum computer to efficiently solve integer factorization and (elliptic curve) discrete logarithm problems poses a threat to current encryption, public key exchange and digital signature schemes [7].

Standards bodies and governmental agencies are really interested in publishing specifications of the most promising PQC schemes. Such schemes may resist to quantum attackers. Organization such as NSA, NIST and IETF are focusing towards post-quantum cryptography [11]. Among these, NIST and ETSI are the main protagonists. NIST is the United States National Institute of Standards and Technology and it initiated a public project to standardize post-quantum public key encapsulation and signature mechanisms, while ETSI, the European Telecommunications Standards Institute, formed a Quantum-Safe Working Group to make proposals for real-world deployment of those algorithms [19].

NIST initiated in 2016 an open call for quantum-resistant crypto algorithms. This process is in its second round where nine PQ signature algorithms and 17 key exchange schemes are studied for possible standardization [20].

The post-quantum signature algorithms are the following:

1. Crystals-Dilithium [2], which encloses two cryptographic primitives, Kyber and Dilithium, that are based over module lattices;

[2] https://pq-crystals.org/

2. Falcon [3] that is based on the short integer solution problem over NTRU lattices;
3. GeMSS [4], a fast multivariate based signature scheme;
4. LUOV [5], based on multivariate cryptography;
5. MQDSS [6] which is based on the hardness of the multivariate quadratic problem;
6. Picnic [7], which is based on a zero-knowledge proof system and hash functions and block ciphers;
7. qTESLA [8] which relies on the decisional Ring Learning With Errors problem adapted on ideal lattices;
8. Rainbow [9], a multivariable polynomial signature scheme;
9. SPHINCS+ [10] which is a stateless hash-based signature scheme.

In this paper we are going to focus on digital signatures algorithms. Among these, we will focus on the latter one, SPHINCS+, that is based on the SPHINCS signature scheme.

Even if those algorithms will be considered solid enough round after round, the adoption of PQC will also depend on the success of the transition of communication protocols and applications to adopt these new algorithms [8]. The main problem is that the actual integration of such schemes in PKI protocols and use-cases can be challenging for today's Internet, because of the key size overhead and the significant latency related to the heavy computational performance of these schemes [7].

2.5 PQC Algorithms

The main operations that need to be performed in cryptography to assure security on transactions and data are encrypt, decrypt, sign and verify. The goal of post-quantum cryptographic designers is improving the efficiency and usability of those operations performed by the new post-quantum cryptographic algorithms. A PQC algorithm must be designed to be secure against both quantum and classical computers.

There exists several classes of mathematical problems that are conjectured to resist attacks performed by quantum computers and have been used to construct public key cryptosystems [12]. A classification for post-quantum cryptographic algorithms can be based on the specific class of the mathematical problems exploited by the specific algorithm. The most important approaches of post-quantum cryptographic systems are the following ones:

- Multivariate cryptography. These cryptosystems are based on the difficulty of solving non-linear polynomials over a field. If the polynomials have degree two, they are called multivariate quadratic cryptosystems. Those class of algorithms are mainly used for building signature schemes because they provide an extremely short signature [3]. An example of multivariate algorithms is the Unbalanced Oil and Vinegar signature scheme.

[3] https://falcon-sign.info/
[4] https://www-polsys.lip6.fr/Links/NIST/GeMSS.html
[5] https://www.esat.kuleuven.be/cosic/pqcrypto/luov/
[6] http://mqdss.org/
[7] https://microsoft.github.io/Picnic/
[8] https://qtesla.org/
[9] https://link.springer.com/chapter/10.1007/11496137_12
[10] https://sphincs.org/

- Lattice-based cryptography. Their security relies on the assumption that certain optimization problems related to lattices can not be solved efficiently: these are called computational lattice problems. Some examples are the ring learning with errors key exchange (RLWE-KEX) algorithm, based on the learning with error problem and the NTRU scheme, which is based on the shortest vector problem in a lattice [12].

- Code-based cryptography. These cryptographic systems rely on error-correcting codes. An example is the McEliece public key encryption scheme, that is based in particular on the hardness of decoding a general linear code [12].

- Hash-based cryptography. It proposes to use hash functions for digitally signing documents. Hash-based schemes are based entirely on standard hash function properties. Thus, they are believed to be among the most quantum-resistant [12]. Some examples of hash-based algorithms are XMSS and SPHINCS.

First, these PQC algorithms must be secure against both quantum and classical computers. In second place, these systems have to interoperate with different communications protocols and networks.

The problem is that existing post-quantum schemes generally have several limitations. Compared with traditional RSA, DSA and ECDSA schemes, all post-quantum schemes have either larger public keys, larger ciphertexts or signatures, or slower runtime. In order to interoperate with different protocols, we need to design better signature and public key encryption schemes, which can operate with smaller keys and ciphertexts or signatures. Moreover, many post-quantum cryptographic schemes are based on mathematical problems that are, from a cryptographic perspective, quite new, and hence they have received comparably less cryptanalysis [12].

The main goal of PQC research is to make those schemes usable and flexible, while being analysed and approved by security experts, in order to develop fast and secure implementations for both high-performance servers and small embedded devices and to integrate those schemes into existing network infrastructure and applications [22].

In the following sections, we are going to focus on the main characteristics of all those classes of cryptographic systems.

2.5.1 Multivariate

Multivariate cryptography is an asymmetric cryptographic approach based on the difficulty of solving non-linear polynomials over a finite field. A polynomial is an expression consisting of indeterminates or variables and coefficients, that involves only few operations: addition, subtraction, multiplication and positive integer exponents of variables. A polynomial in one indeterminate is called a univariate polynomial, a polynomial in more than one indeterminate is called a multivariate polynomial. Polynomials of degree two are quadratic polynomials. Multivariate quadratic cryptosystems, which are the most used, are based on quadratic polynomials, that are polynomials of degree two. The main security assumption of these schemes is the NP-hardness of the problem of solving nonlinear equations over a finite field [23].

The first multivariate scheme was proposed in 1988 by Tsutomu Matsumoto and Hideki Imai. This scheme was broken by Jacques Patarin who developed a scheme based on Hidden Field Equations [24]. Also this scheme has been broken, but some variants of it are still considered secure. One of those variants is the Unbalanced Oil and Vinegar scheme proposed by Patarin in 1999 [25]. The post-quantum algorithm of the second round of the NIST project is called

Rainbow, and it is based on the Unbalanced Oil and Vinegar (UOV) scheme. In the same way as the latter, it is based on the difficulty of solving systems of multivariate equations.

As explained in [3], multivariate-quadratic signature schemes are reasonably fast and they have extremely short signatures. The main disadvantage of UOV schemes is that they use very long key-lengths, which can require several kilobytes.

However, the long-term security of these schemes has been debated. Multiple attempts to build secure multivariate equation encryption algorithms have failed in the past. For now, some multivariate signature schemes, like Rainbow and UOV, are still considered secure and could provide the basis for a quantum secure digital signature.

It is important to note that UOV schemes are not widely used today: while some attack methods are already known, many more could appear if those schemes become widely used. The security of these algorithms still requires more investigation [25].

2.5.2 Lattice-based

Lattice-based cryptography refers to schemes that exploit lattices. A lattice in R^n is the set of all integer linear combinations of basis vectors $b_1, ..., b_n$ belonging to R^n. In 1996, Miklos Ajtai proposed the first cryptographic scheme directly based on lattice problems. In 1998, Jeffrey Hoffstein, Jill Pipher and Joseph H. Silverman introduced a lattice-based public-key encryption scheme, known as NTRU. Later in 2005, Oded Regev introduced the learning with errors problem, the security of which is based on lattice problems, and which now forms the basis of a variety of public key encryption and signature schemes [26].

According to [3] lattice-based crpytographic schemes are reasonably fast. Since both encryption and decryption in these schemes use only simple polynomial multiplication, these operations are very fast compared to other public key algorithms, such as RSA or elliptic curve cryptography. Moreover, lattice-based algorithms provide reasonably small signatures and keys for proposed parameters. Indeed, the ring learning with errors (ring-LWE) problem uses an additional structure which allows for smaller key sizes [12]. Also the NTRU scheme, which is based on the shortest vector problem in a lattice, allows for relatively small key sizes. The shortest vector problem asks to approximate the minimal Euclidean length of a non-zero lattice vector. This problem is thought to be hard to solve efficiently even with a quantum computer. Also the ring learning with errors key exchange (RLWE-KEX), proposed in 2011 by Jintai Ding, is one of the new class of public key exchange algorithms that are designed to be resistant agains an quantum adversary.

However, both RLWE-KEX and NTRU encryption schemes are not yet undergone a sufficient amount of cryptographic analysis and for this reason their quantitative security levels are highly unclear.

2.5.3 Code-based

Code-based cryptography includes all cryptosystems whose security relies on error correcting codes, chosen with some particular structure or from a specific family [28]. Basically, error correcting codes add redundancy to information. In the case of asymmetric code-based algorithms, an error-correcting code is selected for the description of the private key. For this particular code an efficient decoding algorithm is known and this algorithm must be able to correct a predefined number of errors.

For example, in 1978 the first public key encryption scheme was proposed by Robert McEliece [29] and it was based on the hardness of decoding a general linear code, which is known to be

NP-hard. The private key is a random binary Goppa code, a specific family of error correcting codes, and the public key is a random generator matrix of a randomly permuted version of that code. The ciphertext is a codeword to which some errors have been added, and only the owner of the private key - the Goppa code -, who knows the hidden algebraic structure of the code, can remove those errors [31]. During decades, some parameter adjustment have been required but, according to [27], no attack is known to seriously threaten the system, even with a quantum computer. Although the algorithm has never gained much acceptance in the cryptographic community, it is a candidate for post-quantum cryptography, as it is immune to attacks using Shor's algorithm [30].

Another code-based scheme is the Niederreiter cryptosystem, which is a variation of the McEliece cryptosystem: it is equivalent to McEliece from a security point of view, but its encryption is about ten times faster than the encryption of McEliece. Due to its fast encryption, a signature scheme can be construted from the Niederreiter cryptosystem . In general, code-based signature schemes provide short signatures, and in some cases have been studied enough to support quantitative security conjectures. However, those schemes have keys of many megabytes, and they would need even larger keys in order to be secure against quantum attackers [3].

2.5.4 Hash-based

Hash-based cryptography refers to schemes whose security relies on hash functions. This approach is limited to digital signatures schemes.

Unlike common signature schemes like RSA or DSA, hash-based schemes do not rely on the hardness of mathematical problems: they rely only upon the security of the underlying hash function. Therefore, security assumptions for those schemes are minimal [11] and, for this reason, they have been studied as an interesting alternative to digital signatures like RSA and DSA.

In 1979, Merkle first proposed to exploit hash functions for digitally signing documents [32]. Lamport Lamport [33] then implemented his proposal into a one-time signature scheme. Later, Lamport, Merkle, Diffie and Winternitz then converted Merkle's original scheme, that was a one-time signature scheme, into a many-time signature scheme [12], in order to allow a public and private key pair to be used to sign documents more than once. Besides the Merkle and the Lamport signatures schemes, newer and more efficient hash-based cryptographic systems exist: XMSS and SPHINCS.

This hash-based signature (HBS) family of algorithms is constructed on one or few-time-signature (OTS/FTS) schemes and on Merkle trees, used with secure cryptographic hash functions [7].

As the wordings "few-time signature" and "many-time signature" may suggest, the primary drawback of these algorithms is that for any public key, there is a limit on the number of signatures that can be made using the corresponding set of private keys. For this reason the interest in those signatures decreased, until the desire for cryptography that was resistant to quantum attacks came up: due to the fact that these schemes are based entirely on standard hash function properties, they are considered to be among the most quantum-resistant [12].

Unfortunately, most hash-based schemes, such as XMSS, are stateful. This means that they need to keep track of all produced signatures [5]: the signing operation reads a secret key and a message and generates a signature, but also generates an updated secret key [3]. This makes the algorithm not suitable for many kinds of practical applications, for instance when it is required to share a private key on different computers. In this case, it is necessary to synchronize all of them in order to keep the state, otherwise security could be void [6]. This is not acceptable for

many applications. Nevertheless, this situation does not happen in the case of SPHINCS, which is stateless. This is convenient since it is not necessary to keep track of the state. On the other side, signature sizes are significantly higher than for other stateful schemes, and SPHINCS has not yet benefited from the extensive scrutiny dedicated to XMSS until now [11]. We will look in deep into SPHINCS algorithm in section 4.

Summing up, the main critical issues with most HBS schemes are the following [11]:

1. the statefulness of the private key;
2. the existance of an upper bound on the maximal number of messages to be signed with a given secret key.

We will look in details at hash-based signature schemes in the next chapter.

3 Hash-based signatures

As already mentioned in subsubsection 2.5.4, hash-based cryptography refers to schemes based on hash functions. This approach is limited to digital signatures schemes and it is one of the most promising alternatives for post-quantum cryptography, and more specifically for post-quantum signatures [3][11].

One of the main advantages of the hash-based signature scheme is that it does not rely on the hardness of mathematical problems. For example, the security of common signature schemes like RSA and DSA is respectively based on the integer factorization and the discrete logarithm problems, as said in subsection 2.3. Those are considered hard problems, in the sense that they can not be solved in polynomial time by a standard computer. If those mathematical problems turn out to be solvable in a sufficiently efficient time, like polynomial time, the security of those algorithms is broken. This is what happened with the invention of the Shor's algorithm, which is a quantum algorithm capable of solving those problem in polynomial time, using a quantum computer. In the case of hash-based signatures, if the underlying function turns out to be breakable in the future, it can be easily replaced with a new hash construction [14].

Nowadays quantum computing is becoming an active area of research, thus interest on post-quantum cryptography and specifically on hash-based singatures scheme has increased. As proof of this, during the last decade international organizations like NSA, NIST and IETF started focusing towards PQC [11] and, as presented in subsection 2.4, NIST started the NIST post-quantum project in 2016.

In subsection 3.1 we will have a general overview on hash-based signatures. In subsection 3.2, we will introduce the building blocks for hash-based algorithms: cryptographic hash functions, one-time and few-time signature schemes and Merkle's trees. Then, in subsection 3.3, we will discuss about the statefulness property of hash-based algorithms. In the end, in subsection 3.4 we will show the main advantages of hash-based signatures.

3.1 Introduction

Digital signatures are an essential primitive of modern cryptography, as a mean to verify the authenticity of digital messages and documents sent through a non-secure channel. A valid digital signature mainly provides two security properties to the signed message:

1. Authentication: the message to the recipient was created by the claimed sender, that is the one who owns the corresponding private key of the public key the recipient is using to verify the signature of the message.

2. Integrity: the message was not altered in transit. Even if only one bit has been modified, the recipient notices it.

Moreover, with appropriate adjustments, digital signatures can also provide other security properties, such as non-repudiation.

Properties like authentication and non-repudiation can be achieved thanks to the fact that digital signatures are based on public key cryptography. This allows a signer to generate a pair of keys, the public key and the secret key, distribute the first one to the other users, and later issue signatures of messages [9]. These signatures are generated with the secret key, through the "sign" operation and can be verified with the public key, through the "verify" operation. Typically digital signatures do not sign the entire message, because data to be send can be large. Digital signatures are based on public key cryptography and this is slow compared to symmetric cryptography or hash functions. For this reason, only a digest of the message sent is signed by the sender.

Signatures are used in many applications, like financial transactions, contract management software and software distribution. Moreover, digital signatures are often used to implement electronic signatures[11]. For what concerns security protocols, the main purposes of signatures are the following ones [9]:

- They can be used in order to authenticate communicating parties, like in TLS and SSH protocols.
- They can be exploited to deploy a public-key infrastructure (PKI), as it happens for digital certificates.
- They can validate the integrity of packages in software stores, in the scope of software distribution.

As explained in subsection 2.3, the most common signature algorithms are RSA, DSA and ECDSA. They are perceived as being small and fast, as explained in [3], but one of the major threats to these algorithms is that they are no more secure if an attacker can build a large enough quantum computer [6]: these schemes relies on hard number theoretic problems, which can be solved in polynomial time by a quantum computer [17]. Polynomial time is so small that scaling up those algorithms in order to have secure parameters seems impossible [3].

In this situation, hash-based signatures are often mentioned by standard bodies as a viable solution for performing post-quantum authentication [11]. As a matter of fact, these schemes are considered to require minimal security assumptions: every digital signature scheme requires a one-way function and all the security of these signatures is based only on the underlying function [6]. If the hash function gets compromised, it can be replaced. For this reason, no other security assumption is needed. On the downside, the performance of HBS, in terms of both speed and size, has been an obstacle for adoption [5].

Initially, hash-based signature schemes were developed as one-time signature schemes in the late 1970s by Lamport [33], after Merkle's proposal of using hash functions for digital signatures [32]. Later this scheme was extended to a many-times signatures scheme [4], called Merkle

[11]Note that not all electronic signatures exploit digital signatures.

signature scheme. This scheme is based on particular structures called Merkle's trees, that we will see in details in subsubsection 3.2.4.

Anyway, Merkle's tree-based signature scheme requires to fix the number of signatures to be made at key-generation time [4]. This number must be kept small for performance reasons. Most importantly, this system is stateful: it requires users to remember a state. The state is represented by some information to remember how many signatures were already made with the generated key.

Nowadays, the basic construction of hash-based signatures is quite the same. Some ideas improved HBS for what concerns the performance, modifying some of the theoretical foundations of hash-based signatures: this culminated in XMSS [34], which is the first post-quantum signature scheme published as an RFC [35] by the CFRG[12]. The only disadvantage of XMSS is that it is stateful, and this does not fit the standard definition of signature schemes as for example stated in the NIST call for submissions. [4]

For this reason in 2015, a stateless hash-based signature scheme, called SPHINCS, was presented to the NIST post-quantum crypto project and now it is one of the nine remaining signature proposals in the second round of the project for standardization. We will deepen SPHINCS and its enhanced version SPHINCS+ in section 4.

3.2 Primitives

Diffrently from other signature schemes, hash-based signatures do not rely directly on mathematical primitives. Hash-based signature schemes are constructed upon the following main primitives:

1. Cryptographic hash functions;
2. One-time signatures (OTS);
3. Few-time signatures (FTS);
4. Merkle trees (Many-time signatures).

The security of this family of algorithms is based on the correct implementations of those three building blocks. Moreover, the signature specific proprieties of authentication, integrity and non-repudiation are provided mainly by the cryptographic hash function chosen. If the algorithm has been well structured and the underlying hash function respects some well-known properties - as we will explain in the following section - it is considered secure.

3.2.1 Cryptographic hash functions

All hash-based signature schemes rely on the security of an underlying cryptographic hash function.

A cryptographic hash function is a hash function that, given an input message, produces a fixed-length string of bytes. This string of bytes can be referred to as "hash value" or "digest", and in general it is much smaller than the input data. The input message can be a string of any length. In general, given a hash function H with an input message m, the computation of

[12]Crypto Forum Research Group: https://irtf.org/cfrg.

H(m) is a fast operation. Making a comparison, computation of hash functions is much faster than symmetric encryption.

In particular, a cryptographic hash function must be able to resist all known types of cryptanalytic attack. In order to define the security level of a cryptographic hash function, the following properties have been defined in theoretical cryptography:

- *Pre-image resistance.* Given a hash value h and a hash function H, it should be difficult to find an input message m such that h = H(m). This concept is related to one-way functions: it should be computationally hard to reverse hash functions. Functions that lack this property are vulnerable to preimage attacks, where an attacker who knows only the hash value h tries to find the input m.

- *Second pre-image resistance.* Given an input message m1 and a hash function H, it should be difficult to find a different input m2 such that H(m1) = H(m2). This property is also called "weak collision resistance". Functions which does not satisfy this property are vulnerable to second-preimage attacks: an attacker, knowing m1 and its hash h, may substitute the original message m1 with an illegitimate message m2, such that H(m2)=H(m1), without the victim noticing it.

- *Collision resistance.* Given a hash function H, it should be difficult to find two different messages m1 and m2 such that H(m1) = H(m2). This phenomenon is called a cryptographic hash collision. This property, also called "strong collision resistance", means it should be hard to find two different inputs of any length that result in the same hash. It requires a hash value at least twice as long as that required for pre-image resistance, otherwise it would be possible for an attacker to find collisions performing a birthday attack [36]. Note that if a hash function is collision-resistant then it is also second pre-image resistant.

Hash-based signatures require only second preimage or collision resistance of the cryptographic hash function. This kind of assumption is necessary for any digital signature scheme, but all other schemes require additional security assumptions, while HBS do not require any additional assumption. This characteristic makes them stand out among other digital signatures and even among other post-quantum schemes [15], since recent results [37] proved their security also against quantum adversaries [3].

In the end, remember that if a given hash function becomes insecure, the hash-based signature scheme keeps being secure as long as the hash function is replaced by a different and more secure one.

3.2.2 One-time signatures

All hash-based signatures schemes rely on one-time or few-time signatures schemes. In this section we are going to present the first one-time signature (OTS) scheme, the Lamport scheme, and an evolution of it, the Winternitz OTS. As we will see in section 4, SPHINCS hardly relies on the OTS and FTS schemes that we will present in these sections.

The basic idea of HBS is that the public key is a commitment of the secret key while the signature of a message consists of revealing some information, of the secret key, from which the verifier can recompute the commitment [15]. In those schemes one-time signatures are combined with Merkle tree structures. In OTS schemes, a one-time signing key, as the word suggests, can only be used to sign a single message securely, because a signature reveals part of

the signing key, which corresponds to the secret key. For this reason, it is practical to combine many of these keys in a larger structure: a Merkle tree.

The fist proposed OTS scheme was designed by Lamport in 1979 [33]. As most of existing signatures schemes, Lamport scheme relies on one-way functions, which are mostly hash functions. The difference with the other signatures is that Lamport scheme relies solely on the security of these one-way functions.

The simplest example of a OTS scheme is the following one, in which we want to sign a single bit. Given two integers, x and y, the couple (x, y) is the private key of the scheme, while the public key is $(h(x), h(y))$, where h is the hash function on which the scheme relies. If the single bit to be signed is 0, we decide to publish x as the signature, that is a part of the private key. If the bit to sign is 1, we publish y, the other part of the private key. Since to sign bit 0, we reveal the preimage of the first output and to sign bit 1, we reveal the preimage of the second output [3], hash-based signatures always reveal some information about the secret key. From this example we can deduce that this scheme can be used to sign a message only once. Indeed, if another bit is signed, an attacker who can store the signatures will be able to obtain the private key, and this would break this signature scheme. In case of signing multiple bits, first we compute a digest of the data we want to sign, in order to have a fixed-length message as an input. Assuming using SHA-256 algorithm to compute the digest, in order to sign 256 bits, we will need 256 private key pairs (x, y). Therefore, the private key will be $(x_0, y_0, x_1, y_1, ..., x_{255}, y_{255})$ and the public key will be $(h(x_0), h(y_0), h(x_1), h(y_1), ..., h(x_{255}), h(y_{255}))$. In order to sign a digest d in base 2 like: $d = (1011101...)_2$ we will publish $s = (y_0, x_1, y_2, y_3, y_4, x_5, y_6, ...)$ as the signature. The receiver of the signature computes the hash of s and, using the public key, "signs" in the same way the digest d of the message received. If those two values are equal the signature is valid. This scheme is highly impractical since signing a total of N bits of messages requires a sequence of N public keys [3]. Moreover, the secret key in this scheme can be used only once. Nevertheless, these schemes are used as building blocks for existing many-time signature schemes [9].

There are two main advantages of OTS:

1. they may be constructed from any one-way function;
2. the algorithms of signing and verification are very fast and cheap to compute, comparing to standard public-key signatures.

On the other side, they have a limit on the number of signatures to be made, their signatures have a significant length and the size of the public and private keys is relevant [47].

After Lamport's publication, Robert Winternitz proposed the Winternitz OTS (WOTS) scheme, in which he proposed to publish $h^w(x)$ instead of publishing $(h(x), h(y))$ as the public key. In this scheme the hash function is executed w times on the x secret key. For example, using w=16, $h^{16}(x)$ would be the public key, while x would be the private key. Consider a message $m = 1001_2$ expressed in terms of a number in base 10, $m = 9_{10}$. In order to sign it, the signature would be $s = h^9(x)$. The receiver will then receive the message $m = 9_{10}$ and he will "sign" it using the public key, computing $x_1 = h^9(h^{16}(x))$, where $h^{16}(x)$ is the known public key. This value must be compared to x_2, obtained from the signature s and the public key, so $x_2 = h^{16}(h^9(x))$. If $x_1 = x_2$, then the signature is considered valid. If an attacker stores the signature s for $m = 9_{10}$, he could hash it in order to obtain $h^{10}(x)$, that would be a valid signature for $m = 10_{10}$. In order overcome this limitation, a short checksum, to be signed as well, can be added after the message.

Differently from the Lamport scheme, the WOTS scheme can sign many bits at once. The number of bits to be signed at once is determined by the Winternitz parameter w. The larger is this parameter the shorter are the signatures and keys, while sign and verify algorithms get slower. In practice, a common value for w is 16.

The WOTS scheme has been improved through multiple variants. An important variant has been introduced in 2011 by Bunchmann [48]. This variant,instead of using normal hash functions, it uses families of functions parametrized by a key, like a MAC. The private key is the list of the keys used in the MAC, and each bit of the message will dictate how many times to iterate the MAC computation. For example, if we define a W-OTS scheme to work with messages in base 4, the private key would be (sk_1, sk_2, sk_3, sk_4) and the public key is $(x|f^3_{sk_1}(x)|f^3_{sk_2}(x)|f^3_{sk_3}(x)|f^3_{sk_4}(x))$. To sign the message $m = 1032_4$ the signatures will be $s = (f_{sk_1}(x), sk_2, f^3_{sk_3}(x), f^2_{sk_4}(x))$. Note that in this case the signature of 3_4 is equal to the fourth column of the public key $f^3_{sk_3}(x)$. This is not a problem since in a real scenario we would use a checksum applied to the message to be signed.

Since its usage by the SPHINCS algorithm, we now want to focus on a particular variant of WOTS schemes: the WOTS+ scheme, proposed by Hulsing [50]. This variant has shorter signatures and a higher security level in respect of the WOTS variant introduced by Bunchmann. It uses a certain number of function chains starting from random inputs. Those functions belong to the family of keyed functions and the random inputs represent the secret key, which is the same as before $(sk_1, sk_2, ...)$. The public key consists of the final outputs of the chains, which can be the end of each chain. A signature is computed by mapping the message to one intermediate value of each function chain [50]. Moreover, before the one-way function is applied, a random value called mask is XORed with the input. With WOTS+ hash functions with shorter outputs can be used, in respect to the other WOTS variants, achieving the same level of security. Also longer hash chains can be used. In both ways, the signature size is reduced.

All of those schemes follow the general idea that the secret key is used as the input to a sequence of one-way functions which generate a sequence of intermediate results and finally the public key. The one-wayness property of the functions implies that it is infeasible for an attacker to compute the secret key, or any intermediate result, from the public key [49].

3.2.3 Few-time signatures

Starting from OTS schemes, few-time signature schemes were introduced, which can be used for more than one signature. Main examples of FTS are the HORS scheme and the HORS with Trees (HORST) scheme, which extends HORS. Here we will focus on HORS since it provides the building block for many-time signature schemes, such as the SPHINCS signature scheme. Later, in the SPHINCS section we will introduce the HORST scheme, basing our knowledge on the HORS scheme described in this section.

Few-time signature (FTS) schemes were designed to be reusable a few times and still remain secure [9]: their security decreases gradually with the multiple usage of the few-time key.

In 2002, L. Reyzin and N. Reyzin proposed the first few-time signature, called Hash to Obtain Random Subset (HORS), which is actually a r-time signature, where r is a predefined value [51]. This schema improved exisiting OTS schemes, being the fastest one-time signature available.

The security of HORS relies on two main primitives: one-way functions and the subset-resilience problem.

The first basic idea relies on one-way functions and it is very similarly to other OTS: a list of integers is generated, which will be the private key, then hashing these integers the public

key is obtained. In order to sign, a selection function S will provide a list of indexes according to your message m, which has to be a one-way function. Giving an example, we can set 2 parameters $t = 5$ and $k = 2$: t is the length of the private key and k is the number of outputs of the selection function S and, consequently, k is also the lenght of the signatures s. With these values, interpreting messages as integers, we are able to sign messages m whose value in base 10 is smaller than $\binom{t}{k} = 10$. The private key is the list of integers $(s_1, s_2, s_3, s_4, s_5)$, while the public key is the list of values obtained from the private key using another one-way function f: $(f(s_1), f(s_2), f(s_3), f(s_4), f(s_5))$ To sign the integer m=2, the selection function S(m) outputs two values, for example: $S(2) = (1, 4)$. So, to sign m=2, you publish the signature $s = (s_1, s_4)$. Using a proper selection function S, it is impossible to sign two messages with the same parts of the private key. But still, after two signatures it's easy to forge new ones and break the scheme.

The actual construction of the HORS signature scheme enhances this basic idea with the usage of a "subset-resilient" function, instead of having a simple one-way function, in order to select the various indexes. Thus, the selection function S is replaced by a function H that makes it infeasible to find two messages m1 and m2 such that $H(m_2) \subseteq H(m_1)$.

Extending this concept, if we want the scheme to be a r-times signature scheme, where r is the maximum number of signatures to be done with a private key, it should be infeasible to find a message m' such that $H(m') \subseteq H(m_1) \cup ... \cup H(m_r)$. Actually, this is the definition of the subset-resilient problem: the selection function H is r-subset-resilient if any attacker is not able to find in polynomial time a set of r+1 messages that would confirm the previous formula, even with small probability.

The selection function H is hard to reverse, due to its onewayness: for this reason, even knowing the signatures of a previous set of messages, it's hard for an attacker to know what messages would use such indexes. In theory, the selectio function is realized by using a random oracle, in practice by using a hash function: therefore this scheme is called "Hash to Obtain Random Subset".

Summing up, in order to sign a message m, the procedure is the following one:

1. The message m is hashed in order to obtain a fixed-length digest $h = hash(m)$, using an hash algorithm like SHA-256;
2. The digest h is then splitted into k parts $h_1, ..., h_k$. Previously, the k value was the number of outputs of the selection function. Since now we are using a hash function, which provide a single output per input, we will need to split the digest into these k parts;
3. The subset-resilient function H is used to interpret each h_j value into an integer i_j;
4. In the end, the signature is $s = (sk_{i_1}, ..., sk_{i_k})$, where sk_i is the i-th secret key.

In order to expand OTS and FTS constructions, since it is not feasibile to have a signature that can be used just few times, those schemes need to be combined together with particular structures, called Merkle trees [15].

3.2.4 Merkle trees

In order to allow many-time signatures, the basic idea is to use multiple OTS. For the first signature, the first OTS key pair is used, and then it's never used again. The second time that a signature is needed, a second OTS key pair is used and never reused, and so on. This

repetitive procedure could be bad because the public key would consist of all the OTS public keys. Moreover, if the signature scheme should be used a lot, many OTS public keys would be needed. For this reason, it's necessary to find a way of reducing the storage amount of keys.

For what concerns private keys, a seed in a pseudo-random number generator can be used to generate all the secret keys. In this way it is not mandatory to store any secret key, only the seed, from which all secret keys can be generated.

For what concerns public keys instead, Merkle trees are used to link all of these OTS public keys to one main public key, without having a too large to be practical public key. This solution has been invented by Merkle in 1979 [32], and it's referred to as the Merkle Signature Scheme (MSS). Starting with a one-time signature scheme, Merkle constructs a many-time signature scheme [3]: a Merkle tree is a basic binary tree where every node is a hash of its childs, the root is the public key and the leaves are the hashes of the OTS public keys. These trees allow an efficient verification of the contents of wide data structures and they are suitable for generating a significant amount of signatures without breaking the security of the scheme.

For generating the signature keys, OTS (or FTS) schemes are constructed and their corresponding public keys become leaves in the Merkle tree [7]. The global public key corresponds to the root of the resulting Merkle tree and is used to verify each OTS. Its value is an output obtained with a selected hash function.

In order to sign something, the signer uses the first OTS public key, which is the first leave of the tree, and then never use it again. Then he uses the second OTS public key and so on, n times, where n is the number of leaves. The n value is identified by the number of OTS generated. With this approach, the signer can sign a total of n messages.

Then, the verifier has to recover the OTS public key from the signature and check if the global public key, the root of the Merkle tree, is equal to the authentication value associated with the OTS public key and the corresponding authentication path [15]. In this way, the validity of this global public key is related to a given OTS public key, validated using a sequence of tree nodes. This sequence is what we called the authentication path, and it is represented by a sequence of hashes operations, whose length depends on the height of the tree. This path is stored as part of the signature and, as said before, it allows a verifier to reconstruct the node path between the global public key and the OTS one.

Therefore, a signature of a many-time signature scheme is also called a full signature, to distinguish it from other kinds of signatures [3]. A full signature computed with the i-th OTS key contains four parts:

1. The index i of the used OTS key pair in the tree, that is the index of the signing leaf. Since leaves are OTS, you can not reuse that key pair: this makes our scheme stateful, since the signer needs to keep track of the indices i that has already used [9];
2. The OTS public key;
3. The OTS signature, which can be verified using the previous OTS public key;
4. The authentication path, which can be for example the set of sibling nodes on the path from the OTS public key to the root. The authentication path is a list of nodes, hence a list of hashes, which allow us to recompute the root, that is the global public key.

An example of a plain Merkle tree is presented in figure .After signing a message with the first OTS (A), the authentication path would be the one highlighted in BOLD in the figure. We can see that we can compute the global public key with the OTS public key A and the two

hashes $H(B)$ and $H(H(C)|H(D))$, which are the neighbor nodes of all the nodes in the path from the signing leaf to the root of the Merkle tree. Thus we can verify that the obtained signature was originated from a Merkle tree with that specific global public key [**?**]: the root can be recomputed from the OTS public key of index i[13] and the corresponding authentication path for the i-th OTS public key [15]. Exploiting the usage of the authentication path, it is not necessary to know all the OTS public keys to verify the global public key. This technique saves space and computation.

Also, the global public key value is the output of a selected hash function, so typically it has a size of 32 bytes - common hash functions such as SHA-256 produce outputs of 32 bytes. Moreover, the global private key instead is generally handled using a pseudo-random number generator, applied to a seed. In this way it is sufficient to store the seed value. OTS private keys can be directly derived afterwards from the seed value using that generator. In this way, the global private key can be small, typically 32 bytes.

Hence, this approach generates small signatures, small secret keys, exploiting pseudo-random generators, and small public keys [3]. Unfortunately, as exposed in [15], this Merkle Signature Scheme has two main disadvantages:

- First, key generation and signature time are exponential in the tree height h as the whole tree must either be stored or recomputed each time a signature is performed, to generate the structure from the seed [3]. The problem of tree traversal is critical to the signing performance;
- Second, the signer must keep track of the used OTS key pairs and this need makes the scheme stateful.

For the first point, some improvements have been desgined so far [3][34]. For what concerns the last point instead, it will be further analysed in the following section.

3.3 Statefulness

Most of hash-based algorithms are stateful. This means that the overall system requires users to remember a state. This is the case of a traditional Merkle's tree-based signature scheme [4], such as the eXtended Merkle Signature Scheme (XMSS) [35] or the Leighton-Micali Signature (LMS) [39], another common HBS scheme. Those are the most mature HBS schemes [7], but they do not fit standard APIs neither the standard definition of signatures in cryptography [3]: due to their stateful property the sign operation reads a secret key and a message and generates a signature, but it also generates an updated secret key (the secret key is updated at every signature). This means that if the update fails then security disintegrates [3] - e.g. the update could fail if a key is copied from one device to another, or if it is backed up and later restored.

The state could be represented by some information on how many signatures were already made with the key, in order to keep track of all produced signatures. Merkle's design of HBS strongly relies on iterating over signing keys in order [5]: since a stateful HBS is based on a one-time signature, the signer needs to ensure that the OTS private key is never reused.

Practically, it can be difficult to deal with a state. For instance when sharing a private key on different servers that wish to sign messages concurrently, one has to synchronize all of them or security is broken [6]. The state management requirement is an important disadvantage, as

[13]Remember that to ensure that each OTS key pair is used only once, the OTS key pairs are used in a predefined order, for example using the leaves of the tree from left to right [3].

stated by IETF and NIST [7], since it is not acceptable for many applications. Moreover, if an adversary is able to manipulate the state, the security of the scheme degrades significantly [14].

To overcome these limitations, stateless signature schemes have been proposed, such as SPHINCS [9]. However, due to its stateless property, it produces larger signatures and is much less efficient than stateful signature schemes, like XMSS [7]. SPHINCS is built upon the theoretical work by Goldreich [5], who proposed the first stateless HBS scheme.

The structure proposed by Goldreich is so large that, approximately, one can pick a signing key at random each time and can reasonably assume that the key has not been used before [5]. This is essential for many real-world scenarios, as the one previously mentioned of distributed servers, where continuously updating a stateful key pair is often impossible.

SPHINCS improves this construction in several aspects. According to [6], SPHINCS demonstrates that stateless schemes can be practical, even with a reduced overall performance, since it provides a reasonable signature size, of 41 KB, and is able to compute hundreds of signatures per second on a modern CPU.

We will focus on the stateless property of SPHINCS algorithm in subsubsection 4.1.1.

3.4 Advantages of hash-based

Summing up, the main advantage of hash-based signatures is that their security relies only on specific cryptographic properties of the underlying hash function. It follows that if the chosen hash function is broken one day, it can be easily replaced with a new one.

Besides, as explained in [14], from a quantum point of view, the effectiveness of generic attacks against hash functions can not exceed Grover's algorithm [38]. This is a quantum algorithm, invented in 1996, which finds with high probability the unique input to a black box function, given a particular output value, using at most $\sqrt{N}$ evaluations of the function, where N is the size of the domain of the function. In our case, the black box function is the hash function: Grover's algorithm is able to find a preimage x, given the hash function H and the hash value h, such that H(x) = h. This means that a quantum adversary can not obtain more than a square-root speed-up compared to classic preimage search, in which the preimage is found at most with N evaluations, where N is the cardinality of the set of possible inputs [14]. Since HBS schemes are based on hash functions, this result holds also for these schemes.

Moreover, as described in [3], for stateful schemes, hash-based signing is reasonably fast, even without hardware acceleration. Verification is faster, since it directly exploits hash functions. For some hash-based signatures, like XMSS, signatures and keys are reasonably small. This is not the case for the stateless schemes, like SPHINCS, since the stateless propriety does not come for free: it requires bigger keys, with an increase of both the signing time and the signature size [10].

In addition, some hash-based signature schemes, like XMSS, are considered forward secure [34], meaning that previous signatures remain valid if a secret key is compromised.

Until now, while the security of hash-based signatures has been fully analysed, their concrete integration in popular security protocols has not been addressed so far [11], mainly because most HBS schemes are stateful: requiring a state that must be constantly kept up to date is totally different from the signature schemes that are currently in use in many real-world scenarios [10]. Indeed, the stateful property makes these algorithms, such as XMSS, not to fit the standard definition of signature schemes, as the definition stated in the NIST project [4]. For this reason, we are going to focus on a new stateless HBS scheme, SPHINCS, in section 4.

4 SPHINCS

SPHINCS is a recently proposed stateless hash-based signature framework [3]. Before it, all practical hash-based schemes required to keep track of a state. In many real-world use cases, a stateful signature scheme is problematic [10]. For this reason, in 2015, SPHINCS has been proposed, demonstrating that it is not strictly necessary to deal with a state for a hash-based scheme [10]. Being stateless allows this scheme to be a suitable replacement for current signature schemes [3].

It has been known for many years that one can theoretically build hash-based signature schemes without a state: this concept was first introduced by Goldreich in 2004 [52]. However, SPHINCS is the first practical stateless hash-based signature scheme, which provides security against quantum attacks [3].

SPHINCS is carefully designed so that its security can be based on weak standard-model assumptions of the underlying cryptographic hash function [3] and it exploits a randomized tree-based structure. With "randomized" we mean that indexes are selected randomly rather than sequentially [11], as it happens for example with the traditional Merkle Signature Scheme. However, the stateless property has some downsides: there is an increase in signing time and in signature size with respect to stateful schemes [10].

More in details, the tree-based SPHINCS structure relies on HORS with trees (HORST), an improvement of the HORS few-time signature scheme that we analysed previously in section 3.2.3. The usage of a few-time signature scheme instead of a one-time signature scheme is preferred since multiple signing with a given key pair does not cause a complete break in security as in one-time signature schemes, but results in a progressive shrinking of security [11].

In 2015 with the original submission of the algorithm [3], an efficient high-security instantiation of SPHINCS has been proposed, SPHINCS-256. This was the first instantiation proposed and, according to [3], it signs hundreds of messages per second on a modern 4-core 3.5GHz Intel CPU. Its signatures are 41 KB long, public keys and private keys are 1 KB long. The signature scheme, and in particular the SPHINCS-256 instantiation, is designed to provide long-term 2^{128} security even against quantum attackers [3]. However, until now SPHINCS has not yet benefited from the extensive scrutiny that other HBS schemes did already undergo, such as the XMSS scheme [11].

In the following sections, we will describe the structure of the SPHINCS algorithm and analyze its main components. Then, in section **??** we will outline the main characteristics of SPHINCS-256, the first proposed instantiation of the algorithm. In 2017, a new version of the SPHINCS scheme has been proposed. This enhanced version is called SPHINCS+, which, according to [5], it has significant advantages over the state of the art in terms of speed, signature size, and security, and is among the nine remaining signature schemes in the second round of the NIST PQC standardization project. For this reason, in section 4.2, we will introduce the main improvements introduced by this enhanced version and for the subsequent sections we will mainly focus on SPHINCS+.

4.1 Definition of the algorithm: SPHINCS

Before focusing on SPHINCS we want to give a general overview of XMSS, a stateful scheme relying on the MSS scheme. That is because we want to show what are the main differences and innovations introduced by SPHINCS in respect to most recent hash-based signature schemes, among which XMSS is the most common one.

Generally speaking, XMSS main construction looks like a Merkle tree with some differences. The main problem of the Merkle tree signature scheme, described in section 3.2.4, is that key generation and signature time are exponential in the height h of the tree. As explained in [3], XMSS solves these performance problems. Key generation time is significantly reduced using a hypertree of several layers of trees. In this way, during key generation only one tree on each layer has to be generated. The signature time instead is reduced with respect to MSS using stateful algorithms, that exploit the ordered use of the OTS key pairs. Combining hypertrees with the ordered use of trees the signing time is significantly reduced.

In order to look closer to the XMSS hypertree structure, the figure is presented. As illustrated in the figure, a XMSS tree has a mask - identified by the b letter in the figure, and that has a different value for every node - which is first XORed to the corresponding child node and then those results are hashed together in their parents node. Another difference with Merkle trees is that a leaf of the XMSS is not a hash of a OTS public key, as it happened for the Merkle Signature Scheme, but it is the root of another tree called L-tree [34].

A L-tree is not a full binary tree, but it is an unbalanced binary tree [3]. A node that has no right sibling is lifted to a higher level of the L-tree until it becomes the right sibling of another node [34].

With this construction, it follows the same idea explained before: masks are applied to hash its nodes. This mask is different from the one of the main XMSS tree, but it is common to all the L-trees. Inside the leaves of each L-tree the elements of a WOTS+ public key are stored: so a L-tree is used to hash WOTS+ public keys [3]. According to Huelsing, storing the WOTS+ public key makes possible to use a second-preimage resistant hash function, instead of a collision resistant one. Finally, the global public key is composed of the root node of the XMSS tree, the bit masks used in the XMSS tree and a L-tree.

Given this, we want to introduce SPHINCS. It is the most recent hash-based signature scheme and it is stateless. SPHINCS is based on many trees. The first tree is showed in figure. As it happens in the XMSS scheme, each node is computed hashing the concatenation of the previous nodes - the child nodes - XORed with a level bitmask. The level bitmask is different for each level, but it's the same for each node of a fixed level. The public key of the first tree is the root, computed as before hashing the result of the XOR operation between the level bitmask and the concatenation of the child nodes, along with the bitmasks for all the levels of this tree (Q_1 and Q_2 in the figure). The leaves of the tree are the hashed public keys of WOTS+ L-trees. The WOTS+ L-trees are similar to the XMSS L-trees of before, except that their bitmask is unique per level. Each leaf, containing one WOTS+ signature, allows to sign another tree. So now there is a second layer of four SPHINCS trees, containing themselves WOTS+ public keys at their leaves. This procedure is iterated according to some initial parameters, that we will present in [**?**]. Finally, at layer 0, the WOTS+ signatures won't sign other SPHINCS tree but HORST structures, as showed in figure. A HORST or HORS tree is like a L-tree but containing a HORS few-time signatures instaed of a Winternitz one-time signature. Those HORST structures will be used to sign messages. Exploiting FTS intead of OTS schemes increases the security of the scheme since signing a message with the same HORS key does not break the scheme itself.

To sign a message m, the signer first creates a "randomized" hash of m and a "random" index. Since in SPHINCS everything is deterministic, those "randomized" values are actually computed with a Pseudo Random Function (PRF). The index indicates what HORST must be picked to sign the randomized hash of m. Picking the index deterministically according to a message makes the scheme stateless: signing the same message again should use the same HORST, while signing two different messages should make use of two different HORST with good probabilities.

Before going into the details of the SPHINCS scheme that we explained so far, we will first describe its components in order to have a clearer understanding of the building blocks of this scheme. Those components will be described in the next sections following the presented order:

1. First we will focus on the stateless construction, which was invented by Goldreich, who proposed the first HBS scheme: the stateless SPHINCS constructions evolves directly from this scheme;

2. Then, we will have a brief recap on the one-time signature scheme exploited in SPHINCS, the WOTS+ scheme, that we presented in section 3.2.2;

3. We will provide an overview on binary hash trees used by SPHINCS. These trees are similar to the classical binary Merkle trees, exposed in section 3.2.4, but their construction has been slightly modified, with respect to traditional Merkle trees, in the SPHINCS context;

4. After this, we will present the few-time signature scheme HORST, the principal component of the SPHINCS scheme. Note that HORST is the evolution of the HORS scheme described in section 3.2.3;

5. Finally we will present the SPHINCS many-time signature scheme, describing its key generation procedure, its signature and verification algorithms, and summarizing its main parameters.

4.1.1 Goldreich's construction

As explained in section 3.3, most of the practical HBS schemes are stateful, meaning that a state needs to be maintained as a part of the secret key [14]. While this is not a problem in some applications, having to maintain a state for digital signature schemes can have several negative consequences. In scenarios where there are multiple instances of the key stored in different places - as what happens for backups, management of distributed servers or management of different devices owned by the same user -, the state needs to be constantly synchronized among those copies [10]. This makes stateful schemes highly impractical and not suitable for many real-world scenarios.

All of those stateful schemes are improvements of the Merkle Signature Scheme introduced in section 3.2.4, like the XMSS scheme explained in the previous section. The design of the Merkle Signature Scheme heavily relies on iterating over signing keys in order, avoiding the reuse of the same signing key [5].

Already in 1986, Goldreich recognized the problem of having a stateful scheme and proposed a solution. He proposed to create a tree of such depth that, roughly, one can pick a OTS key pair randomly for each signature and reasonably assume it has not been used before [5]: with a sufficient large structure, the probability of reusing a certain OTS key pair becomes insignificantly small [10]. In this way, there is no need to keep track of the OTS keys already used to sign messages. The problem of this solution is how to create such a large tree. He proposed to avoid computing the entire tree: instead of simply hashing nodes together, as for the typical Merkle tree construction, in Goldreich trees an OTS key pair is attached to each tree node and is used to sign the child nodes [10]. This can be realized if and only if the OTS keys of the nodes along the path, starting from a random leaf and arriving to the root node, are deterministically generated: this can be done exploiting a pseudorandom function with a secret seed and the index of the node as inputs [10].

Upon this idea, later in 2004, Goldreich proposed a stateless hash-based signature scheme [52], based on a binary tree of one-time signatures. As explained in [9], each node of this tree corresponds to an OTS instance. Each internal node, so each non-leaf node, authenticates the public keys of its two child nodes by means of the OTS. For example, the OTS of node i signs $(pk_{left(i)}, pk_{right(i)})$ with the secret key sk_i. The global public key is the OTS public key of the root node. The leaf OTS key pairs are used to sign the messages. The secret key is a seed value that is used to generate all the OTS key pairs of the tree pseudorandomly[14]: each OTS instance is generated by a pseudo random generator taking as argument this seed and the index i of the node inside the tree [9]. In this way, the tree is generated on demand, since the root node needs to be computed only at key generation time, and only the nodes on the path need to be computed for each signature.

Since the scheme is constructed upon one-time signatures, it is important to ensure that a single key pair is never used to sign two different messages. This constraint is automatically verified for internal nodes, that only sign the public keys of their children, but it must be enforced for each OTS leaf. In other words, the path, identified by the symbol ρ , must be different for every message. Goldreich proposed to use a random path ρ for every message [9]. In this way, to sign a message a OTS leaf is selected randomly: instead of applying a public hash function to the message to determine the index of the OTS key pair, the index is selected randomly. Following this construction, given a tree of height h, the full signature of the message m contains [3]:

1. The path $\rho = (\rho_1, ..., \rho h)$, where ρ_i is a bit which indicates left or right node;
2. The authentication path A, composed by and authentication A_i for each h layers. Each A_i is computed with all the OTS public keys in the path ρ, from leaf h to the root, and all the public keys of the sibling nodes on this path, signed with the secret key for the i layer. The formula is $A_i = sign(sk_{\rho_i}, (pk_{left(\rho_i)}, pk_{right(\rho_i)}))$;
3. The one-time signature σ of the message computed with the leaf belonging to the path ρ, $\sigma = sign(sk_{\rho_h}, m)$.

Basically, the Goldreich scheme replaces hashing with signing throughout the tree and replaces hash digests with OTS signatures for the nodes in the authentication path included in each signature [10]. For this reason, the main drawback of this approach is the signature size: the parameters required for this construction in order to provide a sufficient security level and a reasonable number of signatures per key pair produce signatures of above 1 MB [6]. According to [3], a signature size like this would dominate the traffic in many applications, such as in operating system updates or in HTTPs connections to an average size web page. Since the Goldreich scheme is highly inefficient, due to its huge signature size, the SPHINCS approach has been introduced in order to drastically reduce signature size [3]. We will present it in section 4.1.5.

4.1.2 WOTS+

We will now recall the Winternitz one-time signature (WOTS+) scheme. It is a one-time signature scheme: the private key must be used to sign exactly one message. When it is reused, security degrades [**?**].

The WOTS+ variation of WOTS is designed to reduce signature size. [**?**] As in WOTS the Winternitz parameter w (generally w=16) is used to configure the efficiency trade-off (trade-off

[14]Note that the pseudo random function should be deterministic for a given seed, so the consistency of the scheme is guaranteed.

between signature size and number of computations.[?]). WOTS+ implements a time/space trade-off parametrised by the Winternitz parameter $w = 2^W$. A small value of w results in a faster scheme, but leads to larger keys and signatures, as the scheme signs W bits at once by hashing a secret value at most $2^W - 1$ times. [?] Likewise, one then derives l (consisting of l1 and l2) from this parameter and the security setting n = 256 as follows: [?]

$$l1 = \lceil \frac{n}{logw} \rceil$$

$$l2 = \lfloor \frac{log(l1(w-1))}{logw} \rfloor + 1$$

$$l = l1 + l2$$

In the plain WOTS scheme, a function F is applied to the secret key several times to produce a hash chain. In WOTS+, however, we take into account the bitmasks. [?]

In each iteration, before applying F, the input is XORed with a round-specific bitmask Qi. The chaining function then looks as follows (where the base case is $c^0(x) = x$): [?] $c^i = F(c^{i-1}(x) \oplus Qi)$

WOTS+ uses the function F to construct the following chaining function $c^i(x, \mathbf{r})$. [?]

For a security parameter $n \in N$, given a second-preimage resistant hash function $F : \{0,1\}^n \to \{0,1\}^n$ and a set of bitmasks $r = (r_1, ..., r_{w-1}) \in \{0,1\}^{(w-1)\times n}$, the WOTS+ chaining function $c^i(x, r)$ is defined as [?]

On input of value $x \in \{0,1\}^n$, iteration counter $i \in N$, and bitmasks $r = (r_1, ..., r_j) \in \{0,1\}^{n\times j}$ with $j \geq i$, the chaining function works the following way. [?]

In case i = 0, c returns x, i.e., $c^0(x, r) = x$. For $i > 0$ we define c recursively as [?]

$$c^i(x, r) = F(c^{i-1}(x, r) \oplus r_i)$$

i.e. in every round, the function first takes the bitwise xor of the previous value $c^{i-1}(x, r)$ and bitmask ri and evaluates F on the result. [?]

We write $r_{a,b}$ for the substring $(r_a, ..., r_b)$ of r. In case b ¡ a we define $r_{a,b}$ to be the empty string. [?]

In order to guarantee deterministic signatures, WOTS+ key pairs are also seeded using SK1 and their location in the tree. [?] This seed is then expanded to a secret key of l pieces, sk = (sk1, ..., skl). [?]

Specifically, we include pseudorandom key generation and fix the message length to be n, meaning that a seed value takes the place of a secret key in our description. [?]

Given n and w, we define the values of l_1, l_2 and $l = l_1 + l_2$. [?]

Generating the key is very similar to traditional WOTS; we simply apply the chaining function c for a total of w-1 times to each part of sk to obtain $(pk_1, ..., pk_l)$. [?] Then in SPHINCS you proceed by builiding a hash tree on top of these public key parts. [?]

The sum of these, len, represents the number of n-bit values in an uncompressed WOTS + private key, public key, and signature. [?]

Now we describe WOTS+ key generation, signature and verification algorithms:

1. Key Generation ($sk, pk \leftarrow WOTS.kg(S, \mathbf{r})$). On input of seed $S \in \{0,1\}^n$ and bitmasks belong $\{0,1\}^{n\times(w-1)}$ the key generation algorithm computes the internal secret key as $sk = (sk1, ..., skl) \leftarrow Gl(S)$, i.e., the n bit seed is expanded to l values of n bits. [**?**] The keys are derived from an initial secret key S which is expanded with a pseudo-random generator to obtain a secret key $sk = (sk_1, ..., sk_l)$. [**?**] The public key pk is then computed by applying the chaining function on each part of the secret key [**?**] $pk = (pk1, ..., pkl) = (c^{w-1}(sk1, r), ..., c^{w-1}(skl, r))$. Note that S requires less storage than sk; thus we generate sk and pk on the fly when necessary. [**?**] **The WOTS + key pair.** In the context of SPHINCS+ , the WOTS + private key is derived from a secret seed SK.seed that is part of the SPHINCS + private key, and the address of the WOTS + key pair within the hypertree, using PRF. [**?**] The corresponding public key is derived by applying F iteratively for w repetitions to each of the n-bit values in the private key, effectively constructing len hash chains. Here, F is parameterized by the address of the WOTS + key pair, as well as the height of the F invocation and its specific chain, in addition to a seed PK.seed that is part of the SPHINCS + public key. [**?**] In contrast to previous definitions of WOTS + , and as a direct consequence of the use of tweakable hash functions to mitigate multi-target attacks, we do not use so-called l-trees to compress the WOTS + public key. Instead, the public key is compressed to an n-bit value using a single tweakable hash function call to Th len . We use 'WOTS + public key' to refer to the compressed public key. [**?**]

2. Signature ($\sigma \leftarrow WOTS.sign(M, S, \mathbf{r})$. On input of an n-bit message M, seed S and the bitmasks r, the signature algorithm first computes a base-w representation of M, A message M is signed by first computing the base w representation of the message: $M = (M_1...M_{l_1})$, $M_i \in \{0, ..., w-1\}$. That is, M is treated as the binary representation of a natural number x and then the w-ary representation of x is computed. [**?**] Next it computes the checksum $C = \sum_{i=1}^{l_1}(w - 1 - M_i)$ and its base w representation $C = (C1, ..., C_{l_2})$. We concatenate these values and obtain B. [**?**] The length of the base w representation of C is at most l2 since $C \leq l1(w-1)$. We set $B = (b_1, ..., b_l) = M \parallel C$, the concatenation of the base w representations of M and C. Then the internal secret key is generated using Gl(S) the same way as during key generation. The signature is computed as $\sigma = (\sigma 1, ..., \sigma l) = (c^{b_1}(sk1, \mathbf{r}), ..., c^{b_l}(sk_l, \mathbf{r}))$. [**?**] **WOTS+ signature**. An input message m is interpreted as len 1 integers m i, between 0 and w - 1. We compute a len 1 checksum C = sigma maiusc i=1 (w - 1 - m i) over these values, represented as string of len 2 base-w values C = (C 1 , . . . , C len 2). This checksum is necessary to prevent message forgery: an increase in at least one m i leads to a decrease in at least one C i and vice-versa. Using these len integers as chain lengths, the chaining function F is applied to the private key elements. This leads to len n-bit values that make up the signature. [**?**]

3. Verification (l'apostrofo sta per pk computed). ($pk' \leftarrow WOTS.vf(M, \sigma, \mathbf{r})$). On input of an n-bit message M, a signature σ , and bitmasks $\mathbf{r}$, the verification algorithm first computes the b_i , $1 \leq i \leq l$ as described above. Then it returns: [**?**] The process of verifying a signature σ of a message M with the public key pk is done in a similar way. First, we have to recompute B and then compute [**?**]

$$pk' = (pk'1, ..., pk'l) = (c^{w-1-b_1}(\sigma_1, \mathbf{r}_{b_1+1,w-1}), ..., c^{w-1-b_l}(\sigma_l, \mathbf{r}_{b_l+1,w-1}))$$

. Note that the correct bitmasks have to be used in each step of the chaining function to get the correct results. The final step is to recompute the root of the L-tree and check

if pk' = pk. [?] **WOTS+ verification**. The verifier can then recompute the checksum, derive the chain lengths, and apply F to complete each chain to its full length. This leads to the chain heads that are hashed using Th len to compute the n-bit public key. [?]

4.1.3 Binary hash trees and L-trees

SPHINCS uses a hash tree (also known as Merkle tree). [?]

The central elements of the SPHINCS construction are full binary hash trees. [?] We use the construction proposed for the XMSS already mentioned in ?? shown in figure BOH.

A single tree. To be able to sign 2^h messages, the signer derives 2^h WOTS + public keys. We use these keys as leaf nodes. To construct a binary tree, one repeatedly applies H on pairs of nodes, parameterized with the unique address of this application of H as well as the public seed PK.seed. [?]

In SPHINCS, a binary hash tree of height h always has 2^h leaves which are n bit strings L_i, $i \in [2^h - 1]$. Each node $N_{i,j}$, for $0 < j \leq h, 0 \leq i < 2^{h-j}$, of the tree stores an n-bit string. [?] To construct the tree, h bit masks $Q_j \in \{0,1\}^{2n}$, $0 < j \leq h$, are used. For the leaf nodes define $N_{i,0} = L_i$. The values of the internal nodes $N_{i,j}$ are computed as: [?]

$$N_{i,j} = H((N_{2i,j-1} \parallel N_{2i+1,j-1}) \oplus Q_j)$$

We also denote the root as $ROOT = N_{0,h}$. [?]

One of the WOTS+ leaf nodes is used to create a signature on an n-bit message. Simply publishing the WOTS + signature is not sufficient, as the verifier also requires information about the rest of the tree. For this, the signer includes the so-called authentication path. [?]

An important notion is the authentication path $Auth_i = (A0, ..., Ah - 1)$ of a leaf L_i shown in figure As already explained before in ??, $Auth_i$ consists of all the sibling nodes of the nodes contained in the path from L_i to the root. [?] Given a leaf L_i together with its authentication path $Auth_i$, the root of the tree can be computed using Algorithm 1. [?]

The verifier first derives the WOTS+ public key from the signature, and then uses the nodes included in the authentication path to reconstruct the root node. A tree of trees. To make it sufficiently unlikely that random selection of a leaf node repeatedly results in the same leaf node being selected, a SPHINCS tree needs to be considerably large. [?]

In order to reduce the size of this public key we build a hash tree on top of it to obtain pk. As l is usually not a power of two the L-tree construction is used. This structure is similar to a binary tree, however if there is an odd number of nodes on a level the rightmost node is lifted up one level. The root of the resulting tree is then used as the public key pk. [?]

In addition to the full binary trees above, we also use unbalanced binary trees called L-Trees. [?] These are exclusively used to hash WOTS+ public keys. The l leaves of an L-Tree are the elements of a WOTS+ public key and the tree is constructed as described above but with one difference: a left node that has no right sibling is lifted to a higher level of the L-Tree until it becomes the right sibling of another node. Apart from this the computations work the same as for binary trees. [?] The L-Trees have height $\lceil logl \rceil$ and hence need $\lceil logl \rceil$ bitmasks. [?]

4.1.4 HORST

The other important component of SPHINCS is a few-time signature scheme. SPHINCS uses HORST, which is a variant of HORS ?? with an additional tree structure. HORST has two parameters t and k, as HORS. [?]

The problem with HORS is that it has large public keys. Of course, one can replace the public key in any signature system by a short hash of the original public key, but then the original public key needs to be included in the signature; this does not improve the total length of key and signature. [?]

As a better FTS for SPHINCS we introduce HORS with trees (HORST). Compared to HORS, HORST sacrifices runtime to reduce the public key size and the combined size of a signature and a public key. [?]

A HORST public key is the root node of a binary hash tree of height log t, where the leaves are the public key elements of a HORS key. This reduces the public key size to a single hash value. For this to work, a HORST signature contains not only the k secret key elements but also one authentication path per secret key element. Now the public key can be computed given a signature. A full hash-based signature thus includes just k log t hash values for HORST, compared to t hash values for HORS. [?]

The configuration of HORST consists of two parameters that control the security level, signature size, and key sizes: t and k. [?]

HORST signs messages of length m and uses parameters k and $t = 2^\tau$ with $k\tau = m$. HORST improves HORS using a binary hash-tree to reduce the public key size from tn bits to n bits and the combined signature and public key size from tn bits to $(k(\tau - x + 1) + 2^x)n$ bits for some $x \in N \setminus \{0\}$. [?] The value x is determined based on t and k such that $k(\tau - x + 1) + 2^x$ is minimal. It might happen that the expression takes its minimum for two successive values. In this case the greater value is used. [?] In contrast to a one-time signature scheme like WOTS, HORST can be used to sign more than one message with the same key pair. However, with each signature the security decreases. Like for WOTS+ our description includes pseudorandom key generation. [?] We now describe the algorithms for HORST:

1. Key generation ($pk \leftarrow HORST.kg(S, Q)$). On input of seed $S \in 0, 1^n$ and bitmasks $Q \in \{0, 1\}^{2n \times logt}$ the key generation algorithm first computes the internal secret key $sk = (sk_1, ..., sk_t) \leftarrow G_t(S)$. [?] In order to generate the secret key we expand a secret S to obtain $sk = (sk_1, ..., sk_t)$, as for WOTS+ key generation.[?] The elements of this list are used to generate the leaves of a binary tree [?]: The leaves of the tree are computed as $L_i = F(sk_i)$ for $i \in [t - 1]$ and the tree is constructed using bitmasks Q. The public key pk is computed as the root node of a binary tree of height log t. [?] We then compute a hash tree on top of these leaves and the public key is the root node. [?]

2. Signature ($(\sigma, pk) \leftarrow HORST.sign(M, S, Q)$). On input of a message $M \in \{0, 1\}^m$, seed $S \in \{0, 1\}^n$, and bitmasks $Q \in 0, 1^{2n \times logt}$ first the internal secret key sk is computed as described above. [?] For signing, the message m is split into k pieces of length log t giving us $M = (M_1, ..., M_k)$. Next, we interpret each M i as an integer and compute the signature as $\sigma = (\sigma 1, ..., \sigma k, \sigma k + 1)$. [?] Then, let $M = (M_0, ..., M_{k-1})$ denote the k numbers obtained by splitting M into k strings of length log t bits each and interpreting each as an unsigned integer. [?] The signature $\sigma = (\sigma_0, ..., \sigma_{k-1}, \sigma_k)$ consists of k blocks $\sigma_i = (sk_{M_i}, Auth_{M_i})$ for $i \in [k - 1]$ containing the M_ith secret key element and the lower $\tau - x$ elements of the authentication path of the corresponding leaf $(A_0, ..., A_{\tau-1-x})$. [?] The block σ_k contains all the 2^x nodes of the binary tree on level $\tau - x(N_{0,\tau-x}, ..., N_{2^x-1,\tau-x})$. In addition to the signature, HORST.sign also outputs the public key. [?] For pieces of length to address a piece of HORST, the nodes along the authentication path from these hashes to the root of the tree are also required as part of the signature. [?]

3. Verification ($pk' \leftarrow HORST.vf(M, \sigma, Q)$). First, the received parts of the secret key are hashed using F. [?] On input of message $M \in 0, 1^m$, a signature σ, and bitmasks

$Q \in 0,1^{2n \times logt}$, the verification algorithm first computes the M_i , as described above. Then, for $i \in [k-1]$, $y_i = \lfloor Mi/2^{\tau} - x \rfloor$ it computes $N'_{y_i,\tau-x}$ using Algorithm 1 (non e lo stesso di PRIMA???) with index M_i, $L_{M_i} = F(\sigma_i^1)$, and $Auth_{M_i} = \sigma_i^2$. It then checks that $\forall i \in [k-1] : N'_{y_i,tau-x} = N_{y_i,tau-x}$, i.e., that the computed nodes match those in σ_k . If all comparisons hold it uses σ_k to compute and then return $ROOT_0$, otherwise it returns **fail**. [**?**] Finally, the nodes in sigma k+1 are used to recompute the root of the tree which has to be equal to pk. [**?**] Verification is quite similar: first, the revealed pieces of k the message is split into sk are hashed, and parts. These parts are then interpreted as integers and used to place the pieces of sk on the appropriate leaves. Using the nodes supplied in the signature, the path to the root node can then be computed. This is done for all nodes and authentication paths checking that all agree on the same root. If this is not the case, verification fails. [**?**]

4.1.5 SPHINCS construction

Given all of the above we can finally describe the algorithms of the SPHINCS construction.

Since the Goldreich scheme is highly inefficient, due to its huge signature size, the SPHINCS approach has been introduced in order to drastically reduce signature size [**?**].

SPHINCS solves this by combining the approach of Goldreich with traditional Merkle trees in a nested construction and few-time signatures. This results in a stateless scheme with signatures of 41 KB and private and public keys of 1 KB each. At hundreds of messages per second on a modern 4-core 3.5GHz Intel CPU, it is shown to be sufficiently fast for many practical applications. [**?**]

In 2015, Bernstein et al. proposed SPHINCS, the first practical scheme that has the particularity of being stateless.[**?**] This was a huge step in the development of hash-based schemes, as they can potentially become the next digital signatures standard in a post-quantum world. [**?**]

SPHINCS introduces several improvements upon the Goldreich construction, such as the usage of a few-time signature scheme (HORST) in the leaves, instead of OTS schemes, for signing messages. Replacing the OTS leaves with FTS leaves increases the resilience to path collisions, thus reducing the overall height h of the tree [**?**].

As the leaf nodes of the SPHINCS tree are randomly selected, there is a trade-off to be made between the size of the tree and the likelihood of selecting the same leaf node twice. To sway this trade-off towards allowing smaller trees, SPHINCS uses a few-time signature scheme (FTS) at the bottom of the tree. [**?**]

The HORST variation then proceeds to build a hash tree on top of public key components. The root of this tree is the actual HORST public key pk. For this tree, SPHINCS also makes use of bitmasks. [**?**]

At its core, SPHINCS heavily relies on hash trees. The construction of these trees is slightly different from the classical binary Merkle trees. After concatenating the values of the two child nodes, they are not immediately fed to a hash function, instead, two bitmasks are applied first. [**?**]

XORing with bitmasks is introduced as part of a linear hashing scheme, and it is employed in order to construct binary hash trees that do not require the underlying hash function to be collision resistant. Instead, second- preimage resistance is sufficient to attain unforgeability. This hash-tree construction is the one described above, and is used in SPHINCS. [**?**]

Rather than increasing h' (and incurring the insurmountable cost of computing $2^{h'}$ WOTS + public keys per signing operation), we create a hypertree. [**?**]

Moreover SPHINCS considers the Goldreich's construction a hypertree with h layers, in which each layer is a tree of height 1. So, SPHINCS generalizes this idea to a hypertree, of height h, with d layers of trees of height h/d [**?**]. So, intermediate nodes are replaced by binary hash trees (Merkle's trees) containing WOTS+ structures, so that each tree signs 2^h children instead of 2, in order to form a complete hyper-tree structure [**?**]. Each Merkle tree (hash tree) on the path needs to be fully generated for every signature, so the signing time is increased. Anyway this reduces the size of signatures, because less OTS instances are included in a signature, that are the ones that form the path ρ. Winternitz OTS instances account for the majority of the signature size [**?**].

This construction serves as a certification tree. [**?**] The WOTS + leaf nodes of the trees on the bottom layer are used to sign messages (or, in our case, FTS public keys), while the leaf nodes of trees on all other layers are used to sign the root nodes of the trees below. [**?**]

The WOTS + signatures and authentication paths from a leaf at the bottom of the hypertree to the root of the top-most tree constitutes an authentication path. Crucially, all leaf nodes of all intermediate trees are deterministically generated WOTS+ public keys that do not depend on any of the trees below it. [**?**] This means that the complete hypertree is purely virtual: it never needs to be computed in full. During key generation, only the top-most subtree is computed to derive the public key. We define the total tree to be of height h and the number of intermediate layers to be d, retroactively setting h' to be h/d. [**?**]

A SPHINCS keypair completely defines a "virtual" structure which we explain first. SPHINCS works on a hyper-tree of height h that consists of d layers of trees of height h/d. Each of these trees looks as follows. The leaves of a tree are 2 h/d L-Tree root nodes that each compress the public key of a WOTS + key pair. [**?**] Hence, a tree can be viewed as a key pair that can be used to sign 2 h/d messages. The hyper-tree is structured into d layers. On layer d - 1 it has a single tree. On layer d - 2 it has 2 h/d trees. The roots of these trees are signed using the WOTS + key pairs of the tree on layer d- 1. In general, layer i consists of 2 (d-1-i)(h/d) trees and the roots of these trees are signed using the WOTS + key pairs of the trees on layer i + 1. [**?**] Finally, on layer 0 each WOTS + key pair is used to sign a HORST public key. [**?**] We talk about a "virtual" structure as all values within are determined choosing a seed and the bitmasks, and as the full structure is never computed. The seed is part of the secret key and used for pseudorandom key generation [**?**]

To support easier understanding, in figure DA METTERE shows the virtual structure of a SPHINCS signature, i.e. of one path inside the hyper-tree. It contains d trees Tree i i belong [d - 1] (each consisting of a binary hash tree that authenticates the root nodes of 2 h/d L-Trees which in turn each have the public key nodes of one WOTS + keypair as leaves).

We use a simple addressing scheme for pseudorandom key generation. An address is a bit string of length a = dlog(d + 1)e + (d - 1)(h/d) + (h/d) = dlog(d + 1)e + h. The address of a WOTS + key pair is obtained by encoding the layer of the tree it belongs to as a log(d+1)-bit string (using d-1 for the top layer with a single tree). Then, appending the index of the tree in the layer encoded as a (d - 1)(h/d)-bit string (we number the trees from left to right, starting with 0 for the left-most tree). [**?**]

SPHINCS uses the Merkle Signature Scheme (MSS) to construct an MTS that combines many W-OTS + key pairs using a Merkle tree. [**?**]

Finally, appending the index of the WOTS + key pair within the tree encoded as a (h/d)-bit string (again numbering from left to right, starting with 0). The address of the HORST key pair is obtained using the address of the WOTS + key pair used to sign its public key and placing d as the layer value in the address string, encoded as dlog(d + 1)e bit string. To give an example: In SPHINCS-256, an address needs 64 bits. [**?**]

To reach this goal, the sphincs tree is designed as a Goldreich tree whose nodes are Merkle trees. [**?**]

The nested trees construction forms the base of SPHINCS. The complete structure consists of a total of h = 60 layers, divided over d = 12 layers of sub-trees. This can be viewed as a hypertree of two levels of abstractions, where each node in the global tree represents a sub-tree. Each of these sub-trees then consists of on layer i h/d = 5 layers of nodes themselves. [**?**]

SPHINCS uses a nested tree structure consisting of 12 layers of trees of height 5. Each tree is a binary tree where the leaves are the public key of a WOTS + key pair. The top layer consists of a single tree and each key pair in the leaves is used to sign the root of another tree. Hence, on the second layer we will have 32 trees. This process is repeated until we reach the bottom layer. On the bottom layer we use the final WOTS + keys to sign a HORST public key, which is then used to sign the message. [**?**]

In this new configuration, each leaf in a Merkle tree is used to sign the root of a Merkle tree located in the layer below. Moreover, leaves of a sphincs tree sign a public key of a few-time signature (FTS) scheme, which security is not compromised if the same key pair is used on few different messages. In order to have a quick overview of sphincs, one can see it as a combination of 3 types of trees. [**?**] Namely: [**?**] 1. The sphincs hypertree: a Goldreich tree of height h (60 in sphincs-256) organised in d layers (12 in sphincs-256). Each layer's leaf signs the root of a Merkle tree. 2. Merkle trees of size h/d (= 5 in sphincs-256) whose leaves are public keys for the OTS used in the Goldreich construction: wots. 3. The FTS used to sign the message is signed by the last layer of the hypertree. We now delve a bit deeper into sphincs's machinery. [**?**] The OTS key pairs of the leaves of the trees on the bottom layer are not used to authenticate more sub-trees. Instead, they are used to authenticate the public key of a few-time signature scheme (FTS). An FTS behaves similarly to an OTS, but can be used several times before revealing too much of the secret key. [**?**] By using an FTS rather than an OTS, SPHINCS does not require as many leaf nodes to maintain the same security level: the required maximal probability of selecting the same node repeatedly can be much higher without breaking the system. These FTS keys are used to sign the actual messages. [**?**]

As with Merkle trees, the digest at the root of the tree is used to authenticate the entire structure by constructing authentication paths. All the sub-trees are then chained together as in Goldreich's system. [**?**]

A sphincs tree of height h can be seen as a Goldreich tree of d layers with Merkle trees of height h/d instead of nodes. Some modifications are made: in the leaves of sphincs's Merkle subtrees, all wots public keys are compressed as follows: their l parts are considered as leaves of a binary hash tree; this tree's root is then computed applying this rule: if a node has no sibling, then it is lifted to a higher level in the tree until it has one. The tree's root stands as the compressed wots public key. Like in Goldreich's construction, where each node is indexed, each leaf has an address in sphincs, which contains its layer in the sphincs hypertree, the number of its Merkle tree in the layer and its position in the Merkle tree. [**?**]

In this way, SPHINCS demonstrates that stateless schemes can be practical, providing a reasonable signature size (of 41 KB for the SPHINCS-256 instantiation, which provides 128-bit security against quantum computers [**?**]), while computing hundreds of signatures per second on a modern CPU [**?**].

We also introduce some optimizations that further compress signatures. For the SPHINCS-256 parameters, switching from HORS to HORST reduces the FTS part of the full signature from about 2 MB to just 16 KB. [**?**]

Now we want to analyse the three algorithms of key generation, signature and verification.

Key generation $((SK, PK) \leftarrow kg(1^n))$.

The key generation algorithm first samples two secret key values $(SK_1, SK_2) \in \{0,1\}^n \times \{0,1\}^n$. The value SK1 is used for pseudorandom key generation. The value SK2 is used to generate an unpredictable index in sign and pseudorandom values to randomize the message hash in sign. [?]

For generating the keys in SPHINCS we choose two random 256-bit values SK1 SK2. The first value is used during the key generation and the second one for signing. [?] Furthermore, we need to generate all the bitmasks Q for WOTS+ , HORST and the binary hash trees. For the public key pk we only need to compute the root of the tree at the top and therefore have to generate the 32 WOTS + key pairs. [?]

Also, p uniformly random n-bit values Q ¡-(freccia con dollaro) 0, 1 pxn are sampled as bitmasks where p = maxw-1, 2(h+dlog 'e), 2 log t. [?]

These bitmasks are used for all WOTS + and HORST instances as well as for the trees. [?] In the following we use Q WOTS + for the first w - 1 bitmasks (of length n) in Q, Q HORST for the first 2 log t, Q L-Tree for the first 2dlog 'e, and Q Tree for the 2h strings of length n in Q that follow Q L-Tree. [?] The remaining part of kg consists of generating the root node of the tree on layer d - 1. Towards this end the WOTS + key pairs for the single tree on layer d - 1 are generated. The seed for the key pair with address A = (d - 1——0——i) where i belong [2 h/d - 1] is computed as S A ¡- F a (A, SK 1), evaluating the PRF on input A with key SK 1. [?] In general, the seed for a WOTS + key pair with address A is computed as S A ¡- F a (A, SK 1) and we will assume from now on that these seeds are known to any algorithm that knows SK 1 . The WOTS + public key is computed as pk A ¡- WOTS.kg(S A , Q WOTS +). The ith leaf L i of the tree is the root of an L-Tree that compresses pk A using bit masks Q L-Tree . Finally, a binary hash tree is built using the constructed leaves and its root node becomes PK 1. [?] The SPHINCS secret key is SK = (SK 1 , SK 2 , Q), the public key is PK = (PK 1 , Q). kg returns the key pair ((SK 1 , SK 2 , Q), (PK 1 , Q)). [?] The secret key is then (SK1, SK2 , Q) and the public key (pk, Q). [?]

Signature algorithm $(\Sigma \leftarrow sign(M, SK))$.

The first step is to select a HORST key to sign the message. We use a pseudorandom function to compute the index idx of the HORST key pair which we then use to sign a randomized digest R derived from m giving us the signature sigma HORST. Note that the HORST key pair is fully determined by this idx and the secret key S. [?]

On input of a message M belong $\{0,1\}*$ and secret key $SK = (SK_1, SK_2, Q)$, sign computes a randomized message digest $D \in \{0,1\}^m$. [?] First, a pseudorandom $R = (R_1, R_2) \in \{0,1\}^n \times \{0,1\}^n$ is computed as $R \leftarrow F(M, SK_2)$. Then, D ¡- H(R 1 , M) is computed as the randomized hash of M using the first n bits of R as randomness. The latter n bits of R are used to select a HORST keypair, computing an h bit index i ¡- Chop(R 2 , h) as the first h bits of R 2. [?]

Note, that signing is deterministic, i.e. we need no real randomness as all required 'randomness' is pseudorandomly generated using PRF F. [?]

In SPHINCS, however, the key pair is selected based on the message hash itself. In order to prevent attackers from specifically targeting certain key pairs, some random or unknown factor still needs to be included ? this is what compute a bitstring (idxkR) and the message as input, and use part R SK 2 is for. We first SK 2 using a pseudorandom function that takes idx to select an FTS key pair. The second is used to compute a randomized digest D of the message. This digest is what we will be signing. As a practical result of all this, the selection of an FTS key pair is completely deterministic with respect to a secret key a message SK 2 and M. [?]

Compute D = H(R 1 ——m); [?]

idx¡- the h leftmost bits of R 2. Generate the horst key pair of index idx. [**?**]

Given index i, the HORST key pair with address A HORST = (d——i(0, (d - 1)h/d)——i((d - 1)h/d, h/d)) is used to sign the message digest D, i.e., the first (d - 1)h/d bits of i are used as tree index and the remaining bits for the index within the tree. The HORST signature and public key (sigma H , pk H) ¡- (D, S A HORST , Q HORST) are computed using the HORST bitmasks and the seed S A HORST ¡- F a (A HORST , SK 1). [**?**]

sigma H = signature of D using this horst key pair; [**?**] sigma 0 = signature of the horst public key using the wots key pair at layer 0, which is (in compressed form) in the leaf f 0 with ANDf 0 = d——idx. [**?**]

The SPHINCS signature sigma maiuscola = (i, R 1 , sigma H , sigma W,0 , Auth A 0 , . . . , sigma W,d-1 , Auth A d-1) contains besides index i, randomness R 1 and HORST signature sigma H also one WOTS + signature and one authentication path sigma W,i , Auth A i , i belong [d-2] per layer. These are computed as follows: The WOTS + key pair with address A 0 is used to sign pk H , where A 0 is the address obtained taking A HORST and setting the first dlog(d + 1)e bits to zero. This is done running sigma W,1 ¡- (pk H , S A 0 , Q WOTS +) using the WOTS + bitmasks. [**?**]

Then the authentication path Auth i((d-1)h/d,h/d)) of the used WOTS + key pair is computed. Next, the WOTS + public key pk W,0 is computed running pk W,0 ¡- WOTS.vf(pk H , sigmaW,0 , Q WOTS +). The root node Root 0 of the tree is computed by first compressing pk W,0 using an L-Tree. [**?**]

After selecting a particular FTS key pair, the secret key of this key pair needs to be generated (based on a seed derived from its location and and is then used to sign D and produce the signature message-specic randomness sigma F T S .SK 1) Together with the R generated above and the index idx of the selected sigma maiusc. As key pair, this signature forms the first part of the SPHINCS signature SPHINCS uses an OTS and an FTS for which the public keys can be derived from their respective signatures, there is no need to include the FTS public key here. [**?**]

The next step is to generate the WOTS + key pair which signs the HORST public key used when computing sigma HORST. This again depends entirely on S and the position in the tree and gives us the WOTS + signature sigma w,1 . The public key for this WOTS + signature is part of another tree and needs to be authenticated again. [**?**]

Then Algorithm 1 is applied using the index of the WOTS + key pair within the tree, the root of the L-Tree and Auth i((d-1)h/d,h/d)) . This procedure gets repeated for layers 1 to d - 1 with the following two differences. On layer 1 minore uguale j ¡ d, WOTS + is used to sign Root j-1 , the root computed at the end of the previous iteration. The address of the WOTS + key pair used on layer j is computed as A j = (j——i(0, (d - 1 - j)h/d)——i((d - 1 - j)h/d, h/d)), i.e. on each layer the last (h/d) bits of the tree address become the new leaf address and the remaining bits of the former tree address become the new tree address. [**?**]

The public key of this OTS needs to be authenticated, so we compute all nodes along its authentication path throughout the tree in tau d and include those in sigma maiusc as well. We refer to the nodes along the authentication path in the selected tree on layer d as Auth d. [**?**]

Finally, sign outputs sigma maiuscola = (i (i di indidce, idx), R 1 , sigma H , sigms W,0 , Auth A 0 , . . . , sigma W,d-1 , Auth A d-1). [**?**]

While progressing up the hypertree, all OTS signatures and nodes along the authentication paths need to be added to sigma maiusc . Altogether, the SPHINCS signature Σ now contains

the message-specific randomness R , the index idx of the selected FTS key pair, the FTS signature σ F T S and d pairs of OTS signatures and sequences of nodes along the authentica Auth 1), ... , (sigma OT S,d , Auth d)). [?]

We therefore compute the authentication path Auth w,1 for pk w,1 . This procedure of signing the root with a WOTS + key pair and computing the authentication path is repeated until we reach the top layer. [?]

Verification algorithm ($b \leftarrow vf(M, \Sigma, PK)$). The verification process consists of recomputing the randomized digest for the message and first verifying sigma HORST. [?] If this is successful we continue with the verification of sigma w,1 and all further signature sigma w,i until we reach the root of our tree. If all verifications succeed and the root of the top tree equals pk the signature is accepted. [?]

Compute D = H(R 1 ——m). Compute the horst public key assuming sigma H is valid [?] On input of a message $M \in \{0,1\}*$, a signature Σ, and a public key PK, the algorithm computes the message digest D ¡- H(R 1 , M) using the randomness R 1 contained in the signature. The message digest D and the HORST bitmasks Q HORST from PK are used to compute the HORST public key pk H ¡- HORST.vf(D, sigma H , Q HORST) from the HORST signature. If HORST.vf returns fail, verification returns false. [?] The HORST public key in turn is used together with the WOTS+ bit masks and the WOTS+ signature to compute the first WOTS+ public key pk W,0 ¡- WOTS.vf(pk H , sigma W,0 , Q WOTS +). An L-Tree is used to compute L i((d-1)h/d,h/d) , the leaf corresponding to pk W,0 . Then, the root Root 0 of the respective tree is computed using Algorithm 1 with index i((d - 1)h/d, h/d), leaf L i((d-1)h/d,h/d) and authentication path Auth 0. [?]

Similar to the way the signature was generated, we now continue up the tree along the authentication paths while verifying the signatures on the root nodes of each sub-tree. [?]

Then, this procedure gets repeated for layers 1 to d - 1 with the following two differences. [?] First, on layer 1 ¡= j ¡ d the root of the previously processed tree Root j-1 is used to compute the WOTS + public key pk W,j. [?] Second, the leaf computed from pk W,j using an L-Tree is L i((d-1-j)h/d,h/d) , i.e., the index of the leaf within the tree can be computed cutting off the last j(h/d) bits of i and then using the last (h/d) bits of the resulting bit string. [?] The result of the final repetition on layer d - 1 is a value Root d-1 for the root node of the single tree on the top layer. This value is compared to the first element of the public key, i.e., PK 1 = Root d-1 . If the comparison holds, vf returns true, otherwise it returns false. [?]

compare this last root to the sphincs public key: accept if and only if they are equal. [?]

Let's summarize in the end its main parameters:

Talking about parameters, SPHINCS uses a hyper-tree (a tree of trees) of total height h belong N, where h is a multiple of d and the hyper-tree consists of d layers of trees, each having height h/d. The components of SPHINCS have additional parameters which influence performance and size of the signature and keys: the Winternitz one-time signature WOTS naturally allows for a space-time tradeoff using the Winternitz parameter w belong N, w ¿ 1; the tree-based few-time signature scheme HORST has a space-time tradeoff which is controlled by two parameters k belong N and t = 2 tau with tau belong N and ktau = m. [?] For SPHINCS-256 we select parameters that provide 128 bits of security against quantum attackers and keep a balance between signature size and time. [?] As a running example we present concrete numbers for SPHINCS-256, the principle instanstiation of the algorithms uses the following values; the choices are explained in Section 4. For SPHINCS-256 we use n = 256, m = 512, h = 60, d = 12, w = 16, t = 2 16 , k = 32. [?]

SPHINCS-256 uses which add up to a total size of 32 * 32 = 1024 32 bitmasks in Q ,bytes. Bitmasks thus account for the largest part of the keys. In general, the number of bitmasks is

determined by the part of the scheme that requires the largest number of them the FTS, the OTS or the hash trees. [**?**]

4.2 Updated algorithm: SPHINCS+

SPHINCS+ improves upon SPHINCS in terms of speed and signature size. This is achieved by introducing several improvements that strengthen the security of the scheme and thereby allow for smaller parameters. The main reason for this is the large flexibility offered by the many parameter options. This allows users to make highly application- specific trade-offs with regards to the signature size, the signing speed, the required number of signatures and the desired security level, and even account for platform considerations such as memory limits or hardware support for specific hash functions. [**?**]

One of our main contributions in this context is a new few-time signature scheme that we call FORS. [**?**] Our second main contribution is the introduction of tweakable hash functions and a demonstration how they allow for a unified security analysis of hash-based signature schemes. We give a security reduction for SPHINCS+ using this abstraction and derive secure parameters in accordance with the resulting bound. [**?**]

Among the main distinguishing contributions of SPHINCS + is the introduction of a new few-time signature scheme: FORS. Another important change from SPHINCS to SPHINCS + is the way leaf nodes are chosen. SPHINCS+ uses publicly verifiable index selection. These two changes together make it harder to attack SPHINCS+ via the few-time signature scheme and hence allow us to choose smaller parameters. [**?**] With the same goal, we apply multi-target attack mitigation techniques as proposed in [33], making it harder to attack SPHINCS+ using a (second-)preimage attack. [**?**]

4.2.1 Enhancements

from https://huelsing.net/wordpress/?p=558 Since we published SPHINCS, nice ideas to reduce the signature size and to speed up the scheme. So, for the NIST submission we reviewed all of them and built SPHINCS+ in the process. SPHINCS+ is SPHINCS in terms of high level scheme design but we tweaked some of the internals - especially the few-times signature scheme. SPHINCS + builds on SPHINCS by introducing several improvements:

- Multi-target attack protection: We added the mitigation techniques which we proposed in our PKC 2016 paper (Mitigating Multi-Target Attacks in Hash-Based Signatures). This was already the plan when publishing that paper. We just discussed it in terms of XMSS there as that was the smaller example. The basic concept is to use a different hash function for each call. Each hash function call is keyed with a different key and applies different bitmasks. Keys and bitmasks are pseudorandomly generated from an address specifying the context of the call, and a public seed. To get an abstraction we now introduced the notion of tweakable hash functions which in addition to the input value takes a public seed and an address. We apply the mitigation techniques from [10] using keyed hash functions. Each hash function call is keyed with a different key and applies different bitmasks. Keys and bitmasks are pseudorandomly generated from an address specifying the context of the call, and a public seed. For this we introduce the notion of tweakable hash functions which in addition to the input value take a public seed and an address. -Tree-less WOTS + public key compression: The last nodes of the WOTS + chains are not compressed using an L-tree but using a single tweakable hash

function call. This call again receives an address and a public seed to key this call and to generate a bitmask as long as the input. [?] SPHINCS + makes use of several different function families with cryptographic properties. SPHINCS + applies the multi-target mitigation technique from [33] using the abstraction of tweakable hash functions from above. [?] In addition to several tweakable hash functions, SPHINCS + makes use of two PRFs and a keyed hash function. [?] Input and output length are given in terms of the security parameter n and the message-digest length m, both to be defined more precisely below. [?] Tweakable hash functions. The constructions described in this work are built on top of a collection of tweakable hash functions with one function per input length. For SPHINCS + we fix P = 0, 1 n and T = 0, 1 256 , limit the message length to multiples of n, and use the same public parameter for the whole collection of tweakable hash functions. We write Th l : 0, 1 n x 0, 1 256 x 0, 1 ln -¿ 0, 1 n , for the function with input length ln. [?] There are two special cases which we rename for consistency with previous descriptions of hash-based signature schemes: F : Pseudorandom functions and the message digest. [?] SPHINCS+ makes use of a pseudorandom function PRF for pseudorandom key generation, PRF : 0, 1 n x 0, 1 256 -¿ 0, 1 n , and a pseudo-random function PRF msg to generate randomness for the message compression: PRF msg : 0, 1 n x 0, 1 n x 0, 1 * -¿ 0, 1 n . To compress the message to be signed, we use an additional keyed hash function H msg that can process arbitrary length messages: H msg : 0, 1 n x 0, 1 n x 0, 1 n x 0, 1 * -¿ 0, 1 m. [?] Definition 1 (Tweakable hash function). Let n, alpha belong N, P the public parameters space and T the tweak space. A tweakable hash function is an efficient function Th : P x T x 0, 1 -¿ 0, 1 at the power of n, MD ¡- Th(P,T , M) mapping an alpha-bit message M to an n-bit hash value MD using a function key called public parameter P belong P and a tweak T belong T. [?] We sometimes write Th P ,T (M) in place of Th(P,T , M). We use the term public parameter for the function key to emphasize that it is intended to be public. Tweaks are used to define context and take the role of nonces when it comes to security. In SPHINCS+ we use as public parameter a public seed PK.seed which is part of the SPHINCS+ public key. As tweak we use a hash function address ADRS which identifies the position of the hash function call within the virtual structure defined by a SPHINCS+ key pair. This allows us to make the hash-function calls for each SPHINCS + key pair and position in the virtual tree structure of SPHINCS + independent from each other. [?]

- Tree-less WOTS+ public key compression: The last nodes of the WOTS+ chains are not compressed using an L-tree but using a single tweakable hash function call. This call again receives an address and a public seed to key this call and to generate a bitmask as long as the input. This was not possible before without blowing up the public key. Now, as we generate bitmasks pseudorandomly, this has no influence on public key size.

- FORS: We replace the few-time signature scheme (FTS) HORST by FORS - Forest Of Random Subsets. A FORS key pair does not consist of a single monolithic tree but of k trees of height log t. The leaves of these trees are the hashes of t secret key elements making a total secret key of kt elements. The public key is the tweakable hash of the concatenation of all the root nodes as for the WOTS+ public key. The big difference to HORST is that we now got a dedicated set of secret key values per index derived from the message. While it first might look as if the key size goes up and speed goes down, this only applies when using the same parameters as for HORST. The advantage is that we can now use much smaller parameters and thereby in the end gain in signature size and speed. FORS: A HORST key pair does not consist anymore of a single monolithic tree. Instead it consists of k trees of height a The leaves of these trees are the hashes of the 2 a secret key elements. The public key is the tweakable hash of the concatenation of all the root nodes

as for the WOTS + public key. A FORS key pair can be used to sign k2 a bit message digests. The digest is first split into k strings m i of length 2 a bits each. Next, every m i is interpreted as an integer n [0, 2 a - 1]. Here m i selects the m i -th secret key element of the i-th tree for the signature. The signature also contains the authentication paths for all the selected secret key elements, which means one path of length a per tree. Verification uses the signature to reconstruct the root nodes and compresses them using the tweakable hash. [**?**] As the few-time signature scheme in SPHINCS+, we define FORS: Forest of Random Subsets, an improvement of HORST. FORS security is captured in Section 4, where we introduce a new security notion for hash functions for this very reason. The new security notion strengthens the notion of target subset resilience as previously used to analyze HORS and HORST. FORS is defined in terms of integers k and t = 2 a , and can be used to sign strings of ka bits. The FORS key pair. The FORS private key consists of kt random n-bit values, grouped together into k sets of t values each. In the context of SPHINCS + , these values are deterministically derived from SK.seed using PRF and the address of the key in the hypertree. [**?**] To construct the FORS public key, we first construct k binary hash trees on top of the sets of private key elements. Each of the t values is used as a leaf node, resulting in k trees of height a. We use H, addressed using the location of the FORS key pair in the hypertree and the unique position of the hash function call within the FORS trees. As in WOTS + , we compress the root nodes using a call to Th k . The resulting n-bit value is the FORS public key. FORS signatures. Given a message of ka bits, we extract k strings of a bits. Each of these bit strings is interpreted as the index of a single leaf node in each of the k FORS trees. The signature consists of these nodes and their respective authentication paths (see Figure 3). [**?**] The verifier reconstructs each of the root nodes using the authentication paths and uses Th k to reconstruct the public key. As part of SPHINCS + , a FORS signature is never verified explicitly. Rather, the resulting public key is used as a message, to be implicitly checked together with a WOTS + signature. [**?**]

- Verifiable index selection: In SPHINCS the HORST key pair used to sign the message was selected generating an index in a pseudorandom manner. As a secret seed value was involved, a verifier was unable to check if that index was actually generated that way. This had the disadvantage that an adversary targeting HORST was able to mount a multi-target attack, using one hash computation to target all HORST instances at once. This is not possible anymore. We now compute the index together with the message digest as $(md \parallel idx) = H(R, PK, M)$ where PK is the SPHINCS+ public key and R is a (pseudo-)randomly generated value which becomes part of the signature. Hence, every message that an adversary tries becomes now directly linked to a FORS instance and is unusable for any other instance. In addition, this allows us to omit the index in the SPHINCS+ signature. The message digest is now computed as follows. First, we deterministically generate randomness R = PRF(SK.prf, OptRand, M). Where OptRand is a 256 bit value, per default 0 but can be filled with random bits e.g. taken from a TRNG to avoid deterministic signing (this might be desirable to counter side channel attacks). Then we compute message digest and index as (md——idx) = H msg (R, PK, M) where PK = (PK.seed, PK.root) contains the top root node and the public seed. Hence, we can omit the index in the SPHINCS signature as it would be redundant. This allows to tighten HORST security. [**?**]

- SPHINCS + -'robust' and SPHINCS + -'simple': The updated, Round 2 submission of SPHINCS + adds new, more simple instantiations of the tweakable hash functions similar to those in the LMS proposal for stateful hash-based signatures [14]. This splits the instantiations into the new 'simple' instantiations and the established 'robust' instantia-

tions. The 'simple' instantiations have the advantage of better speed and the drawback of a security argument which in its entirety only applies in the random oracle model. Consequently, the 'robust' instantiations have a more conservative security argument but are slower.

4.2.2 Final implementation?

The final algorithm is like this, regarding key and signature. Regarding the SPHINCS+ key pair. The public key consists of two n-bit values: the root node of the top tree in the hypertree, and a random public seed PK.seed. In addition, the private key consists of two more n-bit random seeds: SK.seed, to generate the WOTS+ and FORS secret keys, and SK.prf, used below for the randomized message digest. [**?**]

The SPHINCS + signature. The signature consists of a FORS signature on a digest of the message, a WOTS + signature on the corresponding FORS public key, and a series of authentication paths and WOTS + signatures to authenticate that WOTS+ public key. To verify this chain of paths and signatures, the verifier iteratively reconstructs the public keys and root nodes until the root node at the top of the SPHINCS+ hypertree is reached. Two points have not yet been addressed: the computation of the message digest, and leaf selection. [**?**]

Here, SPHINCS+ differs from the original SPHINCS in subtle but important details. First, we pseudorandomly generate a randomizer R, based on the message and SK.prf. R can optionally be made non-deterministic by adding additional randomness OptRand. This may counteract side-channel attacks that rely on collecting several traces for the same computation. Note that setting this value to the all-zero string (or using a low-entropy value) does not negatively affect the pseudorandomness of R. Formally, we say that R = PRF(SK.prf, OptRand, M). R is part of the signature. Using R, we then derive the index of the leaf node that is to be used, as well as the message digest as (MD——idx) = H msg (R, PK.seed, PK.root, M). [**?**] In contrast to SPHINCS, this method of selecting the index is publicly verifiable, preventing an attacker from freely selecting a seemingly random index and combining it with a message of their choice. Crucially, this counteracts multi-target attacks on the few-time signature scheme. As the index can now be computed by theverifier, it is no longer included in the signature. [**?**]

Finally, we present three different incarnations of SPHINCS+: SPHINCS+-SHA3 (using SHAKE256) SPHINCS+-SHA2 (using SHA2) SPHINCS+-Haraka (using the Haraka short-input hash function) end url

The submission proposes three different signature schemes: SPHINCS+-SHAKE256, SPHINCS+-SHA-256, SPHINCS+-Haraka. These signature schemes are obtained by instantiating the SPHINCS+ construction with SHAKE256, SHA-256, and Haraka, respectively. end sphincs

from shincs.org The second round submission of SPHINCS+ introduces a split of the above three signature schemes into a simple and a robust variant for each choice of hash function. The robust variant is exactly the SPHINCS+ version from the first round submission and comes with all the conservative security guarantees given before. The simple variants are pure random oracle instantiations. These instantiations achieve about a factor three speed-up compared to the robust counterparts. This comes at the cost of a purely heuristic security argument. end sphincs

These changes allowed us to define parameter sets with signature sizes of as little as 8kb! Admittedly, at NIST security level 1 and a few seconds for signing. However, the implementation is unoptimized and for many use-cases like certificate or code signing about one second actually seems fine. Even at the highest security level (NIST 5) we get down to 30kb at about the same speed. This is compared to SPHINCS with 41kb. Moreover, NIST asked to assume a key pair

is used for 2^{64} signatures. For the old SPHINCS we assumed 2^{50} signatures. This number of signatures has a huge impact on SPHINCS(+) signature size. Hence, if we are willing to make less conservative assumptions on the number of signatures (I personally find 2^{50} already pretty conservative) we could get even smaller signatures. Of course, the parameters we proposed are just what we consider well balanced. At the same level of security there exists a magnitude of possible parameters (one can still get smaller at the cost of speed).

5 SPHINCS+ criticalities

For what concerns criticalities we will focus on the newest version of the algortihm, that is SPHINCS+.

The schemes of the sphincs family are arguably the most practical stateless schemes, and can be implemented on embedded devices such as FPGAs or smart cards. This naturally raises the question of their resistance to implementation attacks. [**?**]

Security notions: Of course, this abstraction is only useful for us if it provides some security properties. What we require from tweakable hash functions are two properties, which we call post-quantum single function, multi-target-collision resistance for distinct tweaks (pq-sm-tcr) and post-quantum single function, multi-target decisional second-preimage resistance for distinct tweaks (pq-sm-dspr). pq-sm-tcr. Essentially, sm-tcr is a variant of target-collision resistance. It is a two-stage game where an adversary A i [**?**]

Security Evaluation (including estimated security strength and known attacks) The security of SPHINCS + is based on standard properties of the used function families. These in turn can be derived from the properties of the hash functions used to instantiate those function families. For the robust instantiations, these properties can be derived from standard model properties of the used hash function and the assumption that the PRF used within the instantiations of the tweakable hash functions (to generate the bitmasks) can be modeled as a random oracle. We want to emphasize again that this assumption about the random oracle is limited to the pseudorandom generation of bitmasks. For the simple instantiations, these properties can be derived from the assumption that the used hash function behaves like a random oracle even in the presence of quantum adversaries which are given quantum oracle access to the function. [**?**]

Disclaimer: The following two subsections present a security reduction for SPHINCS + . This security reduction makes a statistical assumption about the used hash function which does not hold for a random function and, consequently, should not hold for a good cryptographic hash function. This assumption essentially states that every possible input to F has at least one colliding value under F (which we call sibling). It is trivial to construct a hash function for which it is reasonable to conjecture this property. Just take for example SHA-256, apply it once, truncate the result to 248 bits and apply SHA-256 again. However, this would have to be paid for by a factor 2 penalty in speed. [**?**]

Theorem 9.1 For security parameter n belong N, parameters w, h, d, m, t, k as described above, SPHINCS + is existentially unforgeable under post-quantum adaptive chosen message attacks if [**?**] - F, H, and T are post-quantum distinct-function multi-target second-preimage resistant function families, - F fulfills the requirement of Eqn. 14, - PRF, PRF msg are post-quantum pseudorandom function families, - PRF BM is modeled as a quantum-accessible random oracle, and - H msg is a post-quantum interleaved target subset resilient hash function family. [**?**]

5.1 Overview on attacks

from 01 regarding specifically sphincs and not sphincs+ As demonstrated in [**?**], regarding SPHINCS (not SPHINCS+ but the theorem holds also there) there is a theorem which states that SPHINCS is secure as long as the used function (families) provide certain standard security properties. These properties are fulfilled by secure cryptographic hash functions, even against quantum attacks. For the exact statement in the proof we use the notion of insecurity functions. An insecurity function InSec p (s; t, q) describes the maximum success probability of any adversary against property p of primitive s, running in time ¡= t, and (if given oracle access, e.g. to a signing oracle) making no more than q queries. To avoid the non-constructive high-probability attacks discussed in [11], we measure time with the AT metric rather than the RAM metric. Properties are one-wayness (ow), second-preimage resistance (spr), undetectability (ud), secure pseudorandom generator (prg), secure pseudorandom function family (prf), and gamma-subset resilience (gamma-sr). As a complement to the above reduction, we now analyze the concrete complexity of various attacks, both pre-quantum and post-quantum. [**?**] -subset resilience pag 19 -one-wayness pag 20 -second-preimage resistance -prf, prg, undetectability [**?**]

Hash functions can be implemented very efficiently on constrained devices and it is not surprising that several implementations of hash-based signatures on micro-controllers have been proposed [RED + 08,HBB12], including an ARM implementation of sphincs. However, embedded devices are known to be sensitive to physical attacks such as side-channel analysis or fault attacks. Since the seminal article of Boneh, DeMillo and Lipton [BDL97], fault attacks have proved to be the strongest kind of cryptanalysis on embedded devices. In a fault attack, we suppose that the attacker is strong enough to corrupt the internal state of an algorithm during its execution. While this supposes a rather powerful attacker, these conditions can often be fulfilled in real life and generally result in devastating attacks. However, to the best of our knowledge, no fault attack against hash-based signatures has been publicly proposed. [**?**]

Security models for signature schemes. We briefly recall some classical security notions for signature schemes. [**?**] Definition 21 Existential forgery - An adversary is able of existential forgery if there exists a message m such that she can exhibit a valid (message, signature) pair (m, sigma *) where sigma * was not produced by the legitimate signer. [**?**] Definition 22 Universal forgery - An adversary is able of universal forgery if for any message m, she can exhibit a valid signature sigma * . Any attacker able of universal forgery is able of existential forgery. For more formal notations, we refer the reader to e.g. [**?**]

5.2 Hash function vulnerabilities

paper post quantum bernstein This signature system requires a standard cryptographic hash function H that produces 2b bits of output. For b = 128 one could choose H as the SHA- 256 hash function. Over the last few years many concerns have been raised regarding the security of popular hash functions, and over the next few years NIST will run a competition for a SHA-256 replacement, but all known attacks against SHA-256 are extremely expensive. end paper bernstein

The performance of SPHINCS strongly correlates with the performance of two functions F and H which have the following security requirements [**?**] - Preimage Resistance: For a given output y it should be computationally infeasible to find an input x 0 such that y = f (x 0). - Second-Preimage Resistance: For a given x and y = H(x) it should be computationally infeasible to find x 0 6 = x such that f (x 0) = y. - Undetectability: It should be computationally infeasible for an adversary to predict the output. [**?**]

For F we require preimage resistance, second-preimage resistance and undetectability, while H has to be second-preimage resistant. The best generic attacks against an ideal function with an output size of n bits require 2 n calls to the function respectively 2 n/2 on a quantum computer using Grover's algorithm. In the case of SPHINCS an attacker with access to a quantum computer should not be able to succeed in violating any of these properties with less than 2 128 calls to the underlying function. [**?**] Contrary to a generic cryptographic hash function these requirements are very different. For instance we do not require those functions to be collision resistant, which in general is a much stronger requirement. Various cryptographic hash functions in the past have been broken in this setting like MD4 [35], MD5 [37] or SHA-1 [36] and while one can construct collisions in practice for all these functions, finding a preimage is still very costly, even for MD4 [30,22]. The second difference is that these functions have a fixed input size. Most hash functions only reach their best performance for longer messages and several attacks are also only applicable for long messages. [**?**] Before, we discuss the different choices we first take a closer look at how many calls to these functions are required for key generation, signing and verification (also see Table 1). For generating the key in SPHINCS we need to do 32 WOTS + key generations (and the corresponding L-tree) and construct the hash tree. In total this amounts to 32 Â· (67 Â· 15) = 32160 computations of F and (32 Â· 66) + 31 = 2143 computations of H. [**?**] For signing we need to compute one HORST signature and 12 trees which include the costs for one WOTS + key generation each. Note that the WOTS + signature can already be extracted while generating the WOTS + key pairs. This means that one signature requires at least 65536 + (12 Â· 32160) = 451456 calls to F and 65535 + 12 Â· 2144 + 2143 = 93406 calls to H. For verification we need one HORST verification and 12 WOTS + verifications (including the L-tree) which corresponds to at most 12 Â· (67 Â· 15) + 32 = 12092 calls to F and (12 Â· (66 + 5)) + 383 = 1235 calls to H. [**?**]

Given that currently deployed signature schemes (DSA, ECDSA) are much more efficient than hash-based signature schemes, the latter would mostly be useful in a post-quantum setting - unless an efficient classical attack against (EC)DSA is discovered. Hence, it is important to have in mind the complexity of generic quantum attacks against hash functions. We consider a hash function family F n that outputs n bits. [**?**] In the classical world, generic attacks against (second-)preimage resistance have a complexity of 0(2 n) (i.e. brute-force guessing the preimage), and generic attacks against collision resistance have a complexity of 0(2 n/2) (due to the birthday paradox). [**?**] For this reason, hash-based schemes have often tried to rely only on (second-)preimage resistance of the underlying hash function rather than collision resistance, as n can potentially be chosen twice smaller. [9] In the post-quantum world, Grover's algorithm [Gro96] allows to find preimages with a complexity of 0(2 n/2). However, collision resistance is not affected: even though Brassard et al. [BHT98] proposed a quantum algorithm making 0(2 n/3) quantum queries, a finer analysis by Bernstein [Ber09] showed that this method would require so much hardware that the overall complexity wouldn't beat the classical algorithm of complexity 0(2 n/2). To summarize, assuming only generic attacks against F n , the post-quantum security of F n is n/2 bits for preimage resistance as well as collision resistance (Table 1.1).[**?**]

5.3 Differential power analysis attack

We show that at least a 32-bit chunk of the SPHINCS secret key can be recovered using a differential power analysis attack due to its stateless construction. We present novel differential power analyses on a BLAKE-256-based pseudorandom function for SPHINCS-256 in the Hamming weight model. It compromises the security of a hardware implementation of SPHINCS-256. Our analysis is supported by a hardware implementation of BLAKE for SPHINCS. [**?**]

In particular, the side-channel resistance of hash-based signature (HBS) schemes has not been evaluated systematically so far. HBS schemes rely on the security of an underlying hash function, and use a binary hash tree structure. While these schemes are conjectured to be "naturally" side-channel resistant [14], a deeper look is desirable to uncover potential weaknesses and increase confidence in them. We provide a side-channel analysis (SCA) of two prominent HBS schemes: XMSS (including its variant XMSS MT) and SPHINCS. We chose them because XMSS is being standardised, SPHINCS is the only practical stateless HBS scheme. [**?**]

SPHINCS relies on XMSS MT , HORST, and a stateless way of addressing hash-based instances within the scheme. Since the HORST hash tree construction does not leak anything about its secret key, we can assume this component to be side-channel resistant. Moreover, XMSS MT can also be assumed secure by the previous analysis. This leaves us only with the stateless way of computing the PRNG seeds, which we now analyse. This analysis was initially studied in [12]. SPHINCS-256 PRF In SPHINCS-256, the W-OTS + and HORST secret seeds are generated with BLAKE-256(sk 1 —— A) where sk 1 belong 0, 1 256 is the SPHINCS secret key, A belong 0, 1 64 the address of the instance, and BLAKE-256 the hash function [1]. Recovering sk 1 would therefore result in a total security break. We now present a 6-DPA attack on the BLAKE hash function in the context of SPHINCS-256 that recovers one 32-bit chunk of the secret key sk 1. [**?**]

The BLAKE-256 compression procedure takes 12 similar rounds during which the input is mixed. Similarly as in Sec. 3, the goal is to subsequently recover intermediate values at certain points in the procedure, to eventually recover one secret chunk. As these values are mixed with variable values early in the procedure, the DPAs focus on the first two rounds. Within SPHINCS-256, the first round is summarised in Alg. 2. Here, the values v i for 0 ¡= i ¡ 15 are initialised with known constant values. [**?**] Setup and implementation The SCA was performed on an Arduino Due microcontroller, based on the Atmel SAM3X8E Cortex-M3 CPU. Power consumption was collected by placing a local near-field probe on the chip at the position shown in Fig. 6. The attack considers the BLAKE-256 reference implementation [1] with an additional assumption: the addition of (v a + v b) at lines 1 and 4 in Alg. 3 is performed before the rest. This makes the recovery of v a or v b alone harder, but should not affect our results. We provide the code that was used for evaluating the attack at [17]. [**?**]

To confirm the practicality of the attack, we performed the first two DPAs of our attack on real traces. In both cases, the candidate with the bigger correlation factor in absolute value always happens to be the right value. Similar results were found with the least significant bits, which confirms that the overall attack can be successfully mounted, as the other DPAs target the same kind of operations. [**?**]

Impact. The described attack recovers s 4 , the fifth 32-bit chunk of sk 1 . This makes the stateless construction of SPHINCS-256 vulnerable to DPA. Recovering this chunk potentially leads to the recovery of other chunks, but additional investigation is required. [**?**]

Countermeasures. In order to mitigate the effect of this attack, we suggest hiding the order of the Mix procedures. During a BLAKE-256 round, the first four calls - as well as the next four - do not depend on each other. Their order can thus be rearranged randomly. This forces an attacker to synchronise the collected traces, making the DPAs more complex. [**?**]

An attack on the BLAKE-256-based PRF used within SPHINCS-256. It is practical for the actual parameters of SPHINCS-256. Besides these two found vulnerabilities, we performed a thorough analysis of the building blocks of both XMSS and SPHINCS. Our results confirm the conjecture that XMSS provides strong protection against differential power analysis attacks. This further increases the confidence in the security of stateful HBS schemes, which contributes to a rigorous standardisation process. [**?**]

5.4 Fault injection attacks

In [?] they evaluate the susceptibility of SPHINCS to fault injection attacks.

Fault injection attacks. A fault, either natural or malicious, is a misbehaviour of a device that causes the computation to deviate from its specification. For example, a fault can flip bits in a certain memory cell, corrupting the value held in this register. In a fault injection attack, an adversary is able to actively inject malicious faults into a cryptographic device, such that its process outputs faulty data. Invalid outputs, potentially combined with valid ones, are then used to reconstruct parts of secret data, such as signing keys. Research during the last two decades found that many widely used schemes, when implemented without specific fault protection, can be broken by fault attacks. The first successful attack dates back to 1997 [4], where Boneh et al. exploited both a faulty and a valid RSA signature to recover the private key of a device. [?] These attacks are primarily relevant for embedded cryptographic devices like smart cards, where it is reasonable to assume that an adversary has direct access to the device and can control when cryptographic operations are performed. Faults can be induced in various ways. The most classical ones include exposing the device to voltage variations or manipulating the clock frequency, leading to operation outside of the tolerance of the cryptographic devices. When analysing the fault vulnerability of an implementation, the capability of the adversary needs to be taken into account. This involves measuring the accuracy of the injected faults in terms of location, timing, number of bits affected, success probability, duration, ... The more precisely the fault injection can be tuned, the more powerful are the possible attacks. This paper brings new insights into the subject by focusing on the practical details to perform the fault attack on a generic embedded device. We show how a low-cost injection of a single glitch is enough capability to obtain exploitable faulty signatures, and highlight which procedures are critical to protect. Finally, we conclude on a discussion on how to completely thwart the attack on stateful schemes, and why protecting stateless schemes is a difficult task. [?]

This section describes how a universal SPHINCS signature forgery can be created by injecting faults during its signing procedure. First, we explain the different steps of the attack and highlight the procedures that lead to exploitable faulty signatures. Then, the described attack is simulated to illustrate the complexity of the attack in terms of faulty signatures obtained. The section ends with a practical verification on an Arduino Due board using voltage glitching. The attack is based upon Castelnovi et al. [8] who showed in 2018 that SPHINCS-like schemes are notably intolerant to the presence of faults during execution. This is because their CMSS structure uses OTS to sign the roots of intermediate Merkle trees. Since the subtrees are not supposed to change from one execution to another, they are re-computed on-the-fly and re-signed using the same OTS instance. Therefore, a corruption on the root computation results in an OTS instance signing a different message, weakening its security. Attacking SPHINCS with faults first requires signing a message M to obtain a valid signature. [?] By re-signing the same M , the scheme will produce the same signature, passing through the exact same path of the hypertree. However, if an error occurs during the construction of any non-top subtree 0 ¡= i ¡ d - 1, the algorithm will b where sigma output a different signature sigma maiusc b W,i 6 = sigma W,i . [?] Combining the secret values from sigma b W,i and sigma W,i , the attack consists of attempting to forge a signature sigma W,i for a known subtree. Once the subtree is successfully forged, it can be used to maliciously produce (R 1 0 , R 2 0 , sigma H, sigma W,j, Auth j)0 for 0 ¡= j ¡= i and, thus, maliciously signs an arbitrary M . The rest of the signature is simply taken from the valid signature, i.e., (sigma W,j, Auth 0 j) = (sigma W,j , Auth j) for i ¡ j ¡ d. [?]

Processing faulty signatures. As the fault attack forces an OTS to be reused, a post-processing procedure is required to exploit the faulty signatures. Since we have no information on the

corrupted messages, the W-OTS + secret values inside the faulty signatures need to be identified. This is done by using the W-OTS + public key, which can be extracted only with a correct W-OTS + signature of the attacked subtree. [**?**] Using the public key of the W-OTS + instance, the secret values can be identified by correctly guessing all the blocks of the corrupted subtree root. Each block can be confirmed separately by applying c i (x, r) a number of times equivalent to the presumed value of the block. The resulting values are stored in sigma i 0 and correspond to the blocks b 0 i for 0 ¡= i ¡ '. Once the W-OTS + secret values have been recognized, the universal forgery is created by trying to graft a subtree on the existing SPHINCS structure. For this, the attack creates a subtree from a random secret key SK 0 1 and tries to sign its root using the recovered values. If (b 1 , . . . , bl) denote the result of splitting the root in W-OTS + blocks, the signature succeeds if b i ¡= b 0 i for 1 ¡= i ¡= l. In this case, a valid W-OTS + signature for the grafted subtree is then forged. This subtree can then be used to sign the root of all the previous subtrees, which consequently allows universal forgeries. Otherwise, the attempt can be repeated over with a different SK 0 1 until it succeeds. [**?**]

Algorithm optimally exploits the information obtained from the available faulty signatures to create a universal forgery on SPHINCS. As noticed by Castelnovi et al. [8], the attack also applies to the newer schemes SPHINCS +. [**?**] The SPHINCS + scheme, for its part, proposes an optional randomiser to avoid deterministic signing. Thanks to this feature, focusing a single subtree becomes harder, as even signing a same message takes a different path in the hypertree. However, as the subtrees on higher layers are likelier to repeat, the attacker might inject a fault on the penultimate layer (i.e., i = d - 1). More injections are required than with its deterministic variants, but because of the birthday paradox, a W-OTS + instance is being re-used within 2 h/2d successful corruption. Even with the maximum level of security, h = 64 and d = 8, making a re-use successfully occur every 16 corruption in average. HORST in SPHINCS + is replaced by FORST, but is an inefficient target, as argued before. [**?**]

In our attack scenario, we fixed a secret key SK 1 and signed a message to first obtain a valid SPHINCS signature. Then, to speed up the process, we considered only a portion of the signing algorithm; namely, the construction of a single subtree. In other words, the attacked piece of code takes the address of a W-OTS + instance used in the path of the SPHINCS signature, constructs the subtree that contains this instance, and authenticates it by printing its authentication path within the subtree 1026 ms. By synchronising on its rising edge, we would be able to target a specific subprocedure to inject glitch. The fault process requires a single corruption of any of the following subprocedures: - W-OTS + public key generation at the leaves of a subtree, including: [**?**] - Address computation. - Secret seed derivation with PRF. - Secret values derivation with PRNG. - Hash-chain application. - L-tree compression of W-OTS + public key. - Intermediate Merkle tree nodes computation. [**?**] The HORST layer can also be targeted, but the forgery of a HORST instance is more expensive than an MSS subtree. In our proof of concept, we arbitrarily toggled a second GPIO during the computation of the tree node v 3 [1] to synchronise our glitch injection. The injection was therefore controlled to randomly corrupt any instruction of this particular node compression. Note that, given the length of the overall computation, glitching VDDCORE blindly by synchronising on the UART communication would have led to equivalent results. Also, the attack is still possible even if the adversary is unable to control the message but requires more faulty signatures. Moreover, if the fault occurs on a node below the authentication path, the resulting faulty signature is stealthy [8], meaning that a verifier will still accept it as valid. [**?**]

Results. Using the algorithm in Fig. 4 with the q = 85 faulty signatures, we were able to recover all of the attacked W-OTS + secret values. This obviously allowed us to graft a subtree on the first trial. The forged subtree allowed us to forge another signature for a custom HORST instance which was then used to produce a signature for an arbitrary message. The signatures

for the subtrees above these were simply taken from a valid signature, which resulted in an overall SPHINCS signature for a message of our choice. The details for all of these signatures are provided online in [9], as the results are too lengthy to be shown in paper. [?]

Protecting hash-based schemes that implement CMSS, such as SPHINCS, from fault attacks is challenging. This is because the main vulnerability comes from the intermediate Merkle trees being signed by instances of OTS. As this feature is usually a core improvement in the practicality of these schemes, they cannot be "algorithmically fixed". Consequently, additional countermeasures need to be considered. [?] Currently, the only fault detection countermeasure for hash-based stateless digital signatures, such as SPHINCS, was developed by Mozaffari-Kermani et al. [15]. Their study first presents a method that detects faults by recomputing subtrees with swapped nodes, as well as an enhanced hash function that inherently protects against faults. This method unfortunately does not cover every aspect of the fault attack, as other vulnerable instructions need to be taken care of. Interestingly, their study was released before fault attacks on hash-based were researched. [?] The only way to completely thwart the fault threat in a CMSS structure is to compute every OTS once, and store them so the result can be output every time needed. Similar caching techniques can also be designed for stateless schemes. This technique is however not totally effective, as the attacker can still work around the cache to obtain faulty signatures. [?] Let us finally mention classic error detection and correction mechanisms. All the vulnerable instructions could be recomputed several times and compared to each other, so a mismatch can be detected or the majority of the same result can be taken. These recomputations could be done by different hardware modules, so obtaining the same fault is unlikely. Note that faulty signatures still verify as valid signatures in the majority of cases. Thus, verifying after signing is not an effective way of detecting faults. We consider innovative ways of protecting stateless hash-based signatures as interesting future work. [?]

SPHINCS-like structures on embedded devices can be easily defeated by low-cost fault injections. As the future NIST standards will definitely need to be industrialised, the candidates should be designed in a way that already protects against physical attacks. Unfortunately for SPHINCS and its variants, their designs are inherently vulnerable to fault attacks. Moreover, the lack of definitive countermeasure discourages embedded implementation. [?]

We propose the first fault injection attack against the sphincs framework. The attack is done in two steps, a faulting part and a grafting part: [?] 1. The faulting step. Two signatures for the same message are queried. During the second signature computation, a fault is provoked so that an OTS inside the sphincs framework ends up signing a different value than the first time. Usually, an OTS key is only used to sign a single value, but our fault attack compels it to do otherwise. [?] 2. The grafting step. We show that the knowledge of the two signatures - the correct one and the faulted one - can be exploited to recover parts of the secret key of the OTS which was subjected to the fault, and therefore to partially compromise it. In turn, an attacker will use this compromised OTS as a mean to authenticate a tree different from the one it is supposed to authenticate. The attacker then generates a tree which is entirely under its control, and will use the compromised OTS to graft it to the sphincs tree, which is why we call this step the grafting step. [?] The grafted tree is chosen by the attacker and independent from the secret key, while allowing to generate valid signatures for some messages. We show that it is more than enough to provide universal forgery ability to an attacker while explaining how she can achieve it. The attack requires little power from the attacker - which makes it practical - and produces valid signatures, which renders it particularly stealthy. Whereas this attack comes with a non-negligible computational cost for each forgery, we propose trade-offs to lower this cost by slightly increasing the number of faulted signatures available to the attacker. Our attack is generic in the sense that it targets the sphincs framework: it is successful regardless of the underlying hash function used, and is indifferent to the specificities of the original sphincs,

gravity-sphincs or sphincs +. [**?**]

As we have been seen before, the entire attack depends on the capability of the attacker to obtain two distinct wots signatures for the same secret key. In the context of the sphincs framework, the whole construction of the hyper-tree is deterministic and a signature is entirely dependent of both the message and the secret key. This characteristic leads to the fact that no OTS can sign distinct messages, thus ensuring the security of the scheme. In the following we present a fault injection attack allowing an attacker to recover the signature of two different messages with the same wots key. [**?**]

Metti percentuali su riuscita attacchi su SPHINCS+ in [**?**].

Generic countermeasures such as making the signature computation redundant can complicate our attack, but they may incur a significant overhead (for redundancy, a factor 2 in time and space). Indeed, a simple verification of the signature would not be efficient in our case as the attack provides valid signatures. Moreover, only a small part of the execution will be faulted and thus the redundancy must be checked for each of the roots of a Merkle sub-tree. However redundant computation is an efficient way to significantly constrain the attacker to a more powerful model as the fault should be exactly replicable on both executions. [**?**]

Countermeasures that are specific to the sphincs framework in [**?**], like ?. Last countermeasure: finally, sphincs + includes the (optional) possibility to randomize the FTS index selection. At first sight, this randomization looks like it would make our attack harder. However, by first faulting a signature and then observing regular signatures until the top OTS of a regular signature matches the OTS of the faulted signature, one can effectively mount our grafting attack. [**?**]

In this paper we propose the first fault attack against signature schemes of the sphincs family. After an initial cost of a single faulted message, it allows to forge signatures for any message at an (offline) cost of 2 34 hashes per message. [**?**] For any of the targeted schemes, we can forge any message at a cost of about 2 20 hashes functions knowing only 3 faulted messages. Moreover, the fault model is very permissive. While our attack can be thwarted by generic (but possibly costly) countermeasures against fault attacks, we did not find any specific countermeasure. As demonstrated by this work, the deterministic nature of several hash-based signatures and their internal use of OTS can be a weakness against fault attacks. On the defensive side, an interesting line of work would be to propose hash-based constructions which offer some innate resilience against fault attacks. On the offensive side, a natural extension of this work would be to implement the proposed fault attack in practice. [**?**]

6 Test in a use case scenario: the TLS protocol (to be updated)

From a computational efficiency point of view, KeyGen, Sign and Verify have moderate to slow speed (ranging from a few to hundreds of Mcycles). end url

from 01 The parameters, functions, and resulting key and signature sizes of SPHINCS-256 are summarized in Table 1. This section describes how these parameters and functions were chosen. potrebbe essere utile per scegliere i parametri nella parte di experimental evaluation end 01

However, we have also shown that eliminating the state does not come for free. While verification is fast for stateful and stateless schemes, signing with stateful XMSS is about 30 times faster than signing with stateless SPHINCS. The 589 018 151 cycles used by SPHINCs may be acceptable for non-interactive applications that need long term security and cannot maintain a

data size (kB)	*raw transfer time* (ms)	*secure transfer time* (ms)
128	20	22
256	22	30
512	26	37
1024	30	99

Table 1: Experimental results.

state, but we believe that further algorithmic improvements to stateless hash-based signatures will be needed to enable deployment on a broader scale. [**?**]

The performance of SPHINCS directly relates to the underlying cryptographic hash function and therefore the performance of this function is critical, which will be the main focus of this work. The requirements for this function also differ from the classic use cases for cryptographic hash functions, as we do not require collision resistance and the inputs for most calls are rather short, typically 256 or 512 bits. [**?**]

Sezione Experimental evaluation Describe:

- general purpose of the experiments (functional evaluation, performance evaluation, comparison with other stuff)
- experimental setup (hardware and software, including detailed build and configuration instructions if needed)

Then describe each experiment: its specific purpose (e.g. testing a specific feature of the protocol), the command run, and the output (expected and actual). You can use a table like Tab. 1 to group the results (for example if the same experiment was repeated with several data sizes)

For performance testing, remember to run not only experiments with various data size but also – in case of a client-server protocol – stress tests for the server (i.e. increasing number of clients simultaneously requesting attention from the server). Normally the throughput is measured in Mbps.

from 01 useful for parametri forse SPHINCS uses a hyper-tree (a tree of trees) of total height h belong N, where h is a multiple of d and the hyper-tree consists of d layers of trees, each having height h/d. The components of SPHINCS have additional parameters which influence performance and size of the signature and keys: the Winternitz one-time signature WOTS naturally allows for a space-time tradeoff using the Winternitz parameter w belong N, w ¿ 1; the tree-based few-time signature scheme HORST has a space-time tradeoff which is controlled by two parameters k belong N and t = 2 tau with tau belong N and ktau = m. As a running example we present concrete numbers for SPHINCS-256; the choices are explained in Section 4. For SPHINCS-256 we use n = 256, m = 512, h = 60, d = 12, w = 16, t = 2 16 , k = 32. WOTS + . We now describe the Winternitz one-time signature (WOTS +) from [26]. end 01

from sphincs.org The submission proposes three different signature schemes:

SPHINCS+-SHAKE256 SPHINCS+-SHA-256 SPHINCS+-Haraka These signature schemes are obtained by instantiating the SPHINCS+ construction with SHAKE256, SHA-256, and Haraka, respectively.

The second round submission of SPHINCS+ introduces a split of the above three signature schemes into a simple and a robust variant for each choice of hash function. The robust

variant is exactly the SPHINCS+ version from the first round submission and comes with all the conservative security guarantees given before. The simple variants are pure random oracle instantiations. These instantiations achieve about a factor three speed-up compared to the robust counterparts. This comes at the cost of a purely heuristic security argument. end sphincs.org

PAPER 03 c'Ã¨ la parte di parameter selection and sphincs+ instances che potrebbe essere interessante per fare i test, e anche la parte di performance and comparison

from 06 Once algorithms for quantum-resistant key exchange and digital signature schemes are selected by standards bodies, adoption of post-quantum cryptography will depend on progress in integrating those algorithms into standards for communication protocols and other parts of the IT infrastructure. In this paper, we explore how two major Internet security protocols, the Transport Layer Security (TLS) and Secure Shell (SSH) protocols, can be adapted to use post-quantum cryptography. First, we examine various design considerations for integrating post-quantum and hybrid key exchange and authentication into communications protocols generally, and in TLS and SSH specifically. These include issues such as how to negotiate the use of multiple algorithms for hybrid cryptography, how to combine multiple keys, and more. Subsequently, we report on several implementations of post-quantum and hybrid key exchange in TLS 1.2, TLS 1.3, and SSHv2. We also report on work to add hybrid authentication in TLS 1.3 and SSHv2. These integrations are in Amazon s2n and forks of OpenSSL and OpenSSH; the latter two rely on the liboqs library from the Open Quantum Safe project.

This highlights the importance of beginning to plan for the transition to post-quantum cryptography early. There are several steps in such a transition for network protocols. First, each network protocol must be evaluated for any constraints that make it challenging to add new algorithms with potentially new characteristics, such as lack of ability to replace or negotiate cryptographic algorithms, or limitations on sizes of keys or packets. Next, specific choices must be made in how to integrate the new algorithm into the protocol: engineering choices, such as how parameters and keys are represented in network packets, and cryptographic choices, such as how keying material is used. Furthermore, these designs must be done in a way that preserves backward compatibility with endpoints (and middle boxes) that have not yet been upgraded, while achieving desirable protocol functionality for upgraded endpoints. Finally, the transition to post-quantum cryptography includes a twist not seen in previous cryptographic transitions: the use of two (or more) algorithms simultaneously, in what is being called "hybrid" mode. There have been suggestions that some parties may decide to use both traditional (e.g., elliptic curve Diffie-Hellman) and post-quantum algorithms together for a variety of reasons, such as maintaining compliance with industry or government regulations that have not yet been updated while still obtaining post-quantum security, or for early adopters who want the potential of post-quantum security without relying solely on a newer and relatively untested algorithm. end 06

from 09 Their theory and security is well-understood and they are currently undergoing standardisation. However, since they have not been used widely before and feature unique characteristics such as the statefulness of secret keys, a gap between theory and practice remains. One of the most important use cases regarding the Internet are TLS connections, e.g. to secure traffic via the HTTP protocol. A natural question is how HBS fit this use case. While toy models for this use case do not indicate immediate issues, wider deployment raises challenges that we will discuss. Independently of whether HBS should be used for TLS, integration in cryptographic libraries is critical for a wide range of use cases. end 09

from paper 01 It is not at all clear how to securely sign operating-system updates, web-site certificates, etc. once an attacker has constructed a large quantum computer:

- RSA and ECC are perceived today as being small and fast, but they are broken in polynomial time by Shor's algorithm. The polynomial is so small that scaling up to secure parameters seems impossible. end 01

se test

Once algorithms for quantum-resistant key exchange and digital signature schemes are selected by standards bodies, adoption of post-quantum cryptography will depend on progress in integrating those algorithms into standards for communication protocols and other parts of the IT infrastructure. In this paper, we explore how two major Internet security protocols, the Transport Layer Security (TLS) and Secure Shell (SSH) protocols, can be adapted to use post-quantum cryptography. First, we examine various design considerations for integrating post-quantum and hybrid key exchange and authentication into communications protocols generally, and in TLS and SSH specifically. These include issues such as how to negotiate the use of multiple algorithms for hybrid cryptography, how to combine multiple keys, and more. Subsequently, we report on several implementations of post-quantum and hybrid key exchange in TLS 1.2, TLS 1.3, and SSHv2. We also report on work to add hybrid authentication in TLS 1.3 and SSHv2. These integrations are in Amazon s2n and forks of OpenSSL and OpenSSH; the latter two rely on the liboqs library from the Open Quantum Safe project. as said in [8]

Along these lines, US NIST has initiated a public project to standardize post-quantum (PQ) public key encapsulation and signature mechanisms, while ETSI has formed a Quantum-Safe Working Group [13] to make real-world deployment proposals. In accordance, the IETF has received proposals that introduce, and study PQ schemes in protocols [14, 19, 32, 42, 44]. Currently, the NIST project is in its second round where nine PQ signature algorithms, and 17 key exchange schemes are studied for possible standardization [1]. However, the actual integration of such schemes in PKI protocols and use-cases can be proven challenging for today's Internet due to the key size overhead, and significant latency related to the heavier computational performance of these schemes.

from 06 In this paper, we report on case studies exploring how two major Internet security protocols, Transport Layer Security (TLS) and Secure Shell (SSH), can be adapted to use post-quantum cryptography, both for confidentiality (via post-quantum key exchange) and authentication (via post-quantum digital signatures). Each of our case studies includes an evaluation of design options in the context of the protocol, selection of one or two instantiations of those design options and an implementation thereof, accompanied by observations and lessons learned from the implementation. As of this writing, the case studies include results on more than half of the KEMs and signature scheme families submitted to the NIST Round 2 submission. 1 end 06

from 06 Our specific case studies are as follows: - TLS 1.2: Post-quantum and hybrid key exchange, in OpenSSL 1.0.2s and Amazon s2n. - TLS 1.3: Post-quantum and hybrid key exchange, and post-quantum and hybrid authentica- tion, in OpenSSL 1.1.1c. - SSH 2: Post-quantum and hybrid key exchange, and post-quantum and hybrid authentication, in OpenSSH 7.9. The OpenSSL and OpenSSH implementations rely on the liboqs library from the Open Quantum Safe project, which is a C library that provides implementations of post-quantum KEMs and signatures schemes in a common interface based on implementations from NIST submission packages; some of the implementations in liboqs are based on implementations in the PQClean project [25]. The s2n implementation relies on implementations directly from NIST submission packages. Tables 1 and 2 list the KEM and signature schemes tested in the case studies we examine, and whether each scheme's use was successful in these prototypes. The failures encountered were in general due to large message sizes (public keys / ciphertexts for KEMs, public keys / signatures for signature schemes): - Some of the failures involved sizes that were bigger than the protocol specification allowed; these might be fixable by changing the

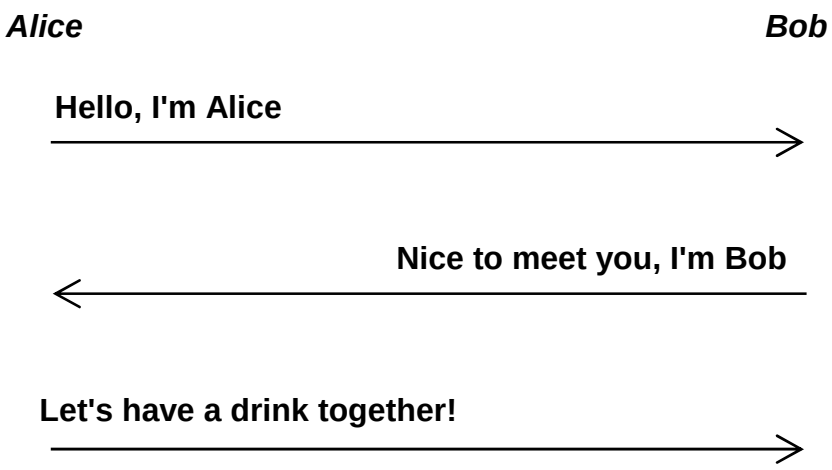

Figure 1: Handshake protocol.

protocol specification, such as increasing 2-byte length fields to 3-byte length fields; but this increases the risk of incompatibilities with existing implementations. These are denoted by in Tables 1 and 2. - Some of the failures involved sizes that were within protocol specification tolerances, but where the implementation in question had internal buffers or parameters set smaller than the maximum size permitted by the specification. In these cases, we were able to increase the implementations' buffers or parameters and get the algorithm working. These are denoted by in Tables 1 and 2. end 06

from 06 The Open Quantum Safe project implemented post-quantum and hybrid key exchange in TLS 1.2 in a fork of OpenSSL 1.0.2 [37] using KEMs from liboqs. That implementation added both PQ-only and hybrid key exchange. 3 The rest of this subsection explains how hybrid key exchange is implemented in TLS 1.2, since PQ-only is implemented just by adding another algorithm. Only ECDH is supported as the traditional algorithm in the hybrid key exchange. end 06

from 06 We now turn to case studies of adding post-quantum and hybrid authentication to the TLS and SSH protocols. Authentication has an additional complication compared to key exchange: there is a long-term credential that must be stored and distributed. In TLS, X.509 certificates are used for long-term credentials; in SSHv2, the usual format is a raw public key (there are some proposals for use of X.509 [23] or other certificates, but raw public keys remain dominant).

end 06

6.1 TLS protocol overview (sender authentication, key agreement)

sd

case of Fig. 1,

6.2 SPHINCS+ integration ()

DA QUALCHE PARTE DOVE RIPORTERO I TEST DOVRO BREVEMENTE SPIEGARE LA DIFFERENZA TRA ROBUST SIMPLE 128S 128F 256 ECC

p

from 06 Authentication in TLS 1.3 Although there have been several Internet-Drafts and experimental implementations of PQ and/or hybrid key exchange in TLS as noted in Section 3.1, none of those works considered PQ or hybrid authentication. This is likely to due to the general consensus that confidentiality against quantum adversaries is a more urgent need than authenticity, since quantum adversaries could retroactively attack confidentiality of any passively recorded communication sessions, but could not retroactively impersonate parties establishing a (completed) communication session. Nonetheless, the advent of a quantum computer would mean that we would eventually need to migrate to post-quantum authentication, meriting some preliminary investigation. While there will certainly a need for implementations to support both old (non-PQ) and new (PQ) algorithms for authentication and to be backwards compatible with implementations that have not yet been upgraded during a transition period, there may perhaps be a slightly weaker need for hybrid authentication than hybrid key exchange, since post-quantum authentication may not be activated until later in the PQ transition when algorithms have had more time to be studied compared to the need for quantum-resistant confidentiality well in advance of a quantum computer. Still we consider some of the issues with hybrid authentication below.

Negotiation. TLS 1.3 has two extensions to negotiate signature algorithms: the first is the signature algorithms cert extension is used to negotiate which algorithms are supported for signatures in certificates; and the second is the signature algorithms extension for which algorithms are supported in the protocol itself. Both of these extensions are a list of algorithm identifiers. Effectively the same considerations apply for each of these as for the supported groups extension for negotiating the key exchange method as described in Section 3.2.1: to negotiate hybrid components individually, additional lists could be added for each type, or delimiters could be used within the existing lists; to negotiate as a combination, new identifiers for each combination could be defined without internal structure, or with internal structure. Conveying public keys. In TLS 1.3, public keys for authentication are usually conveyed via X.509 certificates.

Size limits. The maximum size of an X.509 certificate (or raw public key) in TLS 1.3 is 2 24 - 1 bytes, which is large enough for all Round 2 submissions. Signature size in TLS 1.3 is limited to 2 16 -1 bytes, which is too small for some Round 2 signature schemes, for example Picnic-L3,L5-FS,UR. 1

s. For the experimental results on authentication in TLS 1.3 using OpenSSL 1.1.1 reported in Table 2, the classical signature algorithm used was RSA-3072 and the key exchange method used was ECDH with the nistp256 curve. A single certificate was sent from the server to the client, representing either the scenario where a self-signed certificate is used, or the scenario where the certificate authority is pre-installed in the client, and only the end-entity certificate needs to be sent. In the certificates, the signing algorithm and subject public key algorithm were the same. Longer certificate chains with intermediate CAs were not tested end 06

from 06 Ease of implementation. As noted above, supporting post-quantum authentication requires support in more places through the codebase since certificates come into play. 11 For hybrid, the concatenation approach of making combined algorithms allowed for a simpler implementation, since the hybrid signatures would be treated monolithically within the existing APIs, rather than needing to adapt every API to handle two certificates, two public keys, two signatures, etc. Large PQ sizes. The OQS team encountered several different problems due to large PQ public key or signature sizes, some of which they were able to overcome, and some of which they were not. Recall that the maximum X.509 certificate size in TLS 1.3 is 2 24 - 1 bytes, while the maximum signature size (in the CertificateVerify message) is 2 16 - 1 bytes. However, OpenSSL 1.1.1's code base imposes some additional constraints: - OpenSSL 1.1.1

limits the default maximum size of the Certificate message, which includes the chain of all X.509 certificates except the root, to 102,400 bytes, 12 though this value can be set by a calling application at runtime. 13 Raising this value to 2 24 - 1 bytes allowed all remaining Rainbow schemes to work (though Rainbow-Ia-Cyclic and Ia-Cyclic-Compressed worked even without this fix). - OpenSSL 1.1.1 limits the maximum size of a signature in the CertificateVerify message to 2 14 bytes. This is too small for many post-quantum signatures. Raising this value to 2 16 - 1, 14 allowed the following schemes to work: Picnic-L1-FS,UR, Picnic2-L3,L5-FS, and all SPHINCS+ variants (though SPHINCS+ 128s variants worked even without this fix). The Picnic-L3,L5-FS,UR schemes have signatures that exceed 2 16 - 1 bytes, and thus cannot be be accommodated within the current TLS 1.3 specification. This is also the case for TLS 1.2. An earlier experimental version of the OQS fork of OpenSSL 1.0.2 included post-quantum authentication in TLS 1.2, and the OQS team successfully patched the code to allow a larger signature size (2 24 - 1 rather than 2 16 - 1 by increasing a 2-byte length field to a 3-byte length field), and were subsequently able to use larger signatures successfully. This suggests that the TLS 1.3 specification could be altered to allow larger signatures, although some flag must then be used to communicate that larger length fields are being used. This may however affect compatibility with middle boxes, for example. end 06

from 09 In particular, we introduce support for EVP, ASN.1 and X.509 formats in OpenSSL and for the widely-deployed TLS and S/MIME protocols. Since OpenSSL is sparsely documented, our account can be used as a guide to integrating new signature schemes into the library. Beyond this core integration, we analyse real-world constraints for these protocols, taking into account scheme specificities. Finally, we introduce a strategy for deeper integration and optimised performance.

One of the most important use cases regarding the Internet are TLS connections, e.g. to secure traffic via the HTTP protocol. A natural question is how HBS fit this use case. While toy models for this use case do not indicate immediate issues, wider deployment raises challenges that we will discuss. Independently of whether HBS should be used for TLS, integration in cryptographic libraries is critical for a wide range of use cases.

sphincs, which relies on the use of a few-time signature scheme: instead of yielding a complete break in security as in one-time signature schemes, multiple signing with a given key pair results in a progressive shrinking of security other major difference with stateful schemes is that indexes are selected randomly rather than sequentially. On the downside, signature sizes are significantly higher than for stateful schemes, and SPHINCS has not yet benefited from the extensive scrutiny that XMSS undergoes as part of its ongoing standardization process Our test setup for TLS in OpenSSL, using XMSS for authentication. T end 09

from 05 Next, we focus on the TLS 1.3 handshake pieces affected by PQ algorithms. We assume a classic web scenario where the client is not authenticated. The TLS 1.3 ClientHello message will negotiate the desired PQ signature algorithm using the signature algorithms extensions. The PQ X.509 certificate/chain transmission by the server with the ServerCertificate message will now include PQ certificates. The server will also sign the transcripts of the handshake and transmit a PQ CertificateVerify message that contains a PQ signature which the client will verify along with the signatures in the certificate chain. As usual, when a certificate chain exceeds 16 KB of length, TLS will utilize Record Fragmentation to split packets before sending them [37]. A transition to PQ authentication will require new AlgorithmIdentifiers [9, 35] for X.509. A PQ certificate will carry the subject's PQ public key and the specific PQ signature algorithm used to create the signature. The certificate will be signed by the issuer using his PQ private key and the PQ signature is appended in the Signature field. The addition of the PQ public key and PQ signature to the X.509 certificate will increase the size of the certificate and thus the size of the related certificate chains (see Table 1). In this section we present the TLS 1.3 performance

of each PQ signature scheme. To integrate PQ signatures into X.509 and TLS we utilized the OQS OpenSSL [35] library, which is a fork of OpenSSL that introduces post-quantum algorithms from the liboqs library [34, 41]. The OQS OpenSSL version we used was based on OpenSSL version 1.1.1c. Readers should note that in our evaluation we test PQ certificates but we do not consider the impact of PQ signatures employed in OCSP staples and SCT Timestamps. PQ SCT and OCSP signatures would add significant data to the handshake and requires further testing, but simple caching techniques at the client could significantly alleviate their impact. Our experiments assume a classic web scenario where X.509 certificates are used to authenticate the server. We utilize only full 1-RTT mode TLS 1.3 handshakes with no PSK resumption and a classic X25519 elliptic curve Diffie-Hellman (ECDH) key exchange. The PQ authentication in TLS is compared to currently used algorithms, and specifically against 3072-bit RSA. RSA operates at 128 bits of classical security, but would offer 0 bits of security in a post-quantum setting. Based on our experiments, even RSA2048 or ECDSA certificates perform very similarly to RSA3072 because a few hundred extra bytes in the certificate chain and less than 1ms signing and verification do not add noticeable difference in the total handshake time. The experiments involved one local machine and several cloud-based instances [39]. The local host machine was equipped with 16 GBs of RAM, and an Intel i5-8350U processor, and the tests were implemented on a virtual machine running on four cores (1.7 GHz each) and utilizing 8GBs of RAM. Google Cloud Platform (GCP) instances were used as our remote servers, running on an Intel Xeon processor (2 cores, 2 GHz each) with 8 GBs of RAM. All instances were running Ubuntu 18.04 in an x86 64 architecture. We first measured TLS 1.3 handshake performance. Fig. 1 shows the TLS 1.3 handshake times for classical RSA3072, ECDSA384 and post-quantum NIST Level 1 Falcon and Dilithium with one intermediate CA in the cert chain. We show the handshake time for different distances between client and server. We can see that Dilithium and Falcon perform similarly to classic signature algorithms. Falcon's performance is worse for close distances between client and server primarily because the signing time for Falcon is circ6ms which is noticeable for small round-trips. For higher hop counts both Falcon and Dilithium perform circa5% slower than RSA3072 which is acceptable. Next, we implemented a server that utilizes PQ certificates in TLS 1.3 to service multiple clients that attempt to establish secure connections. The goal was to measure the performance of the server under heavy load. To do so, an Nginx web server [31] was set up on one of the aforementioned cloud instances. We used the Siege tool [16] to simulate multiple PQ authenticated TLS connections from different clients. The client instances were located closely to each other in an attempt to emulate location based content hosting and services. PQ signatures would have an impact on authenticated tunnels like (D)TLS and IKEv2/IPSec. These protocols provide fragmentation mechanisms to allow for lengthy certificate chains, but as we showed above, larger chains and potentially slower algorithms will impact the tunnel establishment. Applications with lower connection rates and tunnels that stay up longer will be less impacted than applications that establish short and quick connections. Thus, per connection overhead is important to be considered for the migration to PQ signature algorithms. end 05

from 05 In this work, we evaluated the impact of using post-quantum hash-based signatures for software signing. We introduced parameter sets at different security levels that would be useful for different software signing use-cases and experimentally showed that the impact of switching to such signatures will be insignificant for the verifier compared to conventional RSA used today. We also showed that the signer will be more impacted but still at an acceptable level and we discussed practical issues with migrating the HBS signatures and their alternatives. We also integrated PQ signature algorithms into TLS 1.3 and evaluated the TLS handshake latency observed by a client, along with the throughput of a PQ authenticated server by considering realistic network conditions. We proved that signature and certificate chain size have some impact on the total handshake time. Our results reveal that for time-sensitive protocols like

HTTPS, the PQ algorithms that present the best performance are Dilithium and Falcon. Other protocols that do not require frequent connection establishments could use one of the other PQ signature algorithms as well. We showed that even slightly slower sign operations could significantly impact the total throughput of a server. However, as optimizations and hardware accelerations improve signing performance, we expect signature and key size to have the most impact on the handshake and total server throughput. end 05

6.2.1 Sender authentication in TLS with SPHINCS+

p

6.2.2 Workflow

p

6.2.3 ...

p

6.3 The Open quantum-safe project

from wiki Open Quantum Safe[56][57] (OQS) project was started in late 2016 and has the goal of developing and prototyping quantum-resistant cryptography. It aims to integrate current post-quantum schemes in one library: liboqs.[58] liboqs is an open source C library for quantum-resistant cryptographic algorithms. liboqs initially focuses on key exchange algorithms. liboqs provides a common API suitable for post-quantum key exchange algorithms, and will collect together various implementations. liboqs will also include a test harness and benchmarking routines to compare performance of post-quantum implementations. Furthermore, OQS also provides integration of liboqs into OpenSSL.[59] end wiki

PAPER 10 section: Open Quantum Safe: a software framework for post-quantum cryptography

from 10 Designing public key cryptosystems that resist attacks by quantum computers is an important area of current cryptographic research and standardization. To retain confidentiality of today's communications against future quantum computers, applications and protocols must begin exploring the use of quantum-resistant key exchange and encryption. In this paper, we explore post-quantum cryptography in general and key exchange specifically. We review two protocols for quantum-resistant key exchange based on lattice problems: BCNS15, based on the ring learning with errors problem, and Frodo, based on the learning with errors problem. We discuss their security and performance characteristics, both on their own and in the context of the Transport Layer Security (TLS) protocol. We introduce the Open Quantum Safe project, an open-source software project for prototyping quantum-resistant cryptography, which includes liboqs, a C library of quantum-resistant algorithms, and our integrations of liboqs into popular open-source applications and protocols, including the widely used OpenSSL library.

This implies that information that needs to remain confidential for many years needs to be protected with quantum-resistant cryptography even before quantum computers exist. In communication protocols like TLS, digital signatures are used to authenticate the parties and key exchange is used to establish a shared secret, which can then be used in symmetric cryptography. This means that, for security against a future quantum adversary, authentication in

today's secure channel establishment protocols can still rely on traditional primitives (such as RSA or elliptic curve signatures), but we should incorporate post-quantum key exchange to provide quantum-resistant long-term confidentiality. This has the benefit of allowing us to introduce new post-quantum ciphersuites in TLS while relying on the existing RSA-based public key infrastructure for certificate authorities. However, applications which require long-term integrity, such as signing contracts and archiving documents, will need to begin considering quantum-resistant signature schemes. end 10

from 10 The goal of our Open Quantum Safe (OQS) project (https://openquantumsafe.org) is to support the development and prototyping of quantum-resistant cryptography. OQS consists of two main lines of work: liboqs, an open source C library for quantum-resistant cryptographic algorithms; and prototype integrations into protocols and applications, including the widely used OpenSSL library. As an example of where the OQS framework can assist with the grand challenge of moving quantum-resistant cryptography towards reliable widespread deployment, consider a small- or medium-sized enterprise that understands the need to integrate quantum-resistant cryptography into its products. Perhaps their products protect information that requires long-term confidentiality. Perhaps their products will be deployed in the field for many years with no easy opportunity for changing the cryptographic algorithms later. Or perhaps they or their customers are worried about the small but non-negligible chance that today's algorithms will be broken, by quantum computers or otherwise, much earlier than expected. Whatever their reason for wishing to integrate quantum-safe cryptography into their products sooner rather than later, this would not be an easy path for them to take. In-house implementation of quantum-safe primitives requires advanced specialized expertise in order to understand the research literature, choose a suitable scheme, digest the new mathematics, choose suitable parameters, and develop robust software or hardware implementations. This is an enormous, expensive, and risky endeavour to undertake on one's own, especially for a small- or medium-sized enterprise. Commercially available alternatives, especially back in 2014 when this project started taking shape, were few, and also potentially problematic from a variety of perspectives: cost, patents, transparency, maintenance, degree of external scrutiny, etc. Companies who would like to offer a quantum-safe option today do not have an easy or robust path for doing so. OQS gives such organizations the option of prototyping an available quantum-resistant algorithm in their applications. Since these are still largely experimental algorithms that have not yet received the intense scrutiny of the global cryptographic community, our recommendation is to use one of the available post-quantum algorithms in a "hybrid" fashion with a standard algorithm that has received intense scrutiny with respect to classical cryptanalysis and robust implementation. Since we fully expect that ongoing developments and improvements in the design, cryptanalysis, and implementation of quantum-safe algorithms, OQS is designed so improvements and changes in the post-quantum algorithm can be adopted without major changes to application software. Organizations who do not wish or need to use open source in their products can still benefit from: - reference implementations that will guide them in their own implementations - benchmark information that will guide their choice of algorithm - the ability to test alternatives in their products before deciding which algorithms to choose. OQS was thus designed with the goal of both facilitating the prototyping and testing of quantum-resistant algorithms in a range of applications, and of driving forward the implementation, testing, and benchmarking of quantum-resistant primitives themselves. The high-level architecture of the OQS software project is shown in Figure 3. The OQS project also includes prototype integrations into protocols and applications. Our first integration is into the OpenSSL library, 3 which is an open source cryptographic library that provides both cryptographic functions (libcrypto) and an SSL/TLS implementation (libssl). OpenSSL is used by many network applications, including the popular Apache httpd web server and the OpenVPN virtual private networking software. Our OpenSSL 1.0.2 fork (https://github.com/open-quantum-safe/openssl) integrates

post- quantum key exchange algorithms from liboqs into OpenSSL's speed command, and provides TLS 1.2 ciphersuites using post-quantum key exchange based on primitives from liboqs. For each post-quantum key exchange primitive supported by liboqs, there are ciphersuites with AES-128 or AES-256 encryption in GCM mode (with either SHA-256 or SHA-384, respectively), and authentication using either RSA or ECDSA certificates. (We use experimental ciphersuite numbers.) Each of these four ciphersuites is also mirrored by another four hybrid ciphersuites which use both elliptic curve Diffie-Hellman (ECDHE) key exchange and the post-quantum key exchange primitive. Our OpenSSL integration also includes generic ciphersuites. liboqs includes interfaces for each key exchange algorithm so it can be selected by the caller at runtime, but it also includes a generic interface that can be configured at compile time. Our OpenSSL integration does include ciphersuites for each individual key exchange algorithm in liboqs, but it also includes a set of ciphersuites that call the generic interface, which will then use whatever key exchange algorithm was specified at compile time. This means that a developer can add a new algorithm to liboqs and immediately prototype its use in SSL/TLS without changing a single line of code in OpenSSL, simply by using the generic OQS ciphersuites in OpenSSL and compiling liboqs to use the desired algorithm. end 10

from 10 As mentioned earlier, one of the goals of the Open Quantum Safe project is to make it easier to prototype post-quantum cryptography. It should be easy to add a new algorithm to liboqs, and then easy to use that algorithm in an application or protocol that already supports liboqs. Recently, we added the NewHope ring-LWE-based key exchange to liboqs and our OpenSSL fork. It is interesting to compare the amount of work required to add NewHope to liboqs and our OpenSSL fork with the figures in Table 3 on adding BCNS15 directly to OpenSSL. In liboqs, the wrapper around NewHope is 2 new files, totalling 163 lines of code, and requires 5 lines of code to be changed in 2 other files (plus changes in the Makefile). As noted above, liboqs includes a "generic" key exchange method which can be hard-coded at compile time to any one of its implementations, and our OpenSSL fork already includes a "generic OQS" key exchange ciphersuite that calls liboqs' generic key exchange method. Thus, once NewHope has been added to liboqs, it is possible to test NewHope in OpenSSL with zero changes to the OpenSSL fork via the generic key exchange method and recompiling. However, to explicitly add named NewHope ciphersuites to OpenSSL, we are able to reuse existing data structures, resulting in a diff that touches 10 files and totals 222 lines of code. Moreover, the additions can very easily follow the pattern from previous diffs, 4 making adding a new OQS-based ciphersuite a 15-minute job. end 10

from 10 The next few years will be an exciting time in the area of post-quantum cryptography. With the forthcoming NIST post-quantum project, and with continuing advances in quantum computing research, there will be increasing interest from government, industry, and standards bodies in understanding and using quantum-resistant cryptography. Lattice-based cryptography, in the form of the learning with errors and the ring-LWE problems, is particularly promising for quantum-resistant public key encryption and key exchange, offering high computation efficiency with reasonable key sizes. More cryptanalytic research will be essential to increase confidence in any standardized primitive. Since each post-quantum candidate to date has trade-offs between computational efficiency and communication sizes compared to existing primitives, it is also important to understand the how applications and network protocols behave when using different post-quantum algorithms. The Open Quantum Safe project can help rapidly compare post-quantum algorithms and prototype their use in existing protocols and applications, and experiments like Google's use of NewHope in its Chrome Canary browser will give valuable information about how post-quantum cryptosystems behave in real-world deployments. For cryptographers interested in designing new public key encryption, digital signature schemes, and key exchange protocols-for cryptanalysts looking to study new mathematical problems- for cryptographic engineers building new systems-and for standards bodies

preparing for the future-exciting times lie ahead! end 10

6.4 Testing

from 10 In the BCNS15 work on ring-LWE-based key exchange, we did a performance evaluation at two levels: the standalone cryptographic operations of the ring-LWE key exchange protocol, and its performance when run in the TLS protocol. The first is a fairly common practice in cryptographic research: implement your algorithms in C, then use some cycle counting or microsecond-accurate timing code to determine the runtime of your algorithms. end 10

paper 07 spiega differenze tra i vari blake, araka, simpira, ecc

o

6.4.1 Testbed design

l

6.4.2 RSA/DSA + DH

k

6.4.3 SPHINCS+ + DH

k

6.4.4 ...

j

6.4.5 Comparison

k

6.5 Discussion

IPSEC SSH ???

7 Conclusions

from 06 The case studies we explored provide a preliminary investigation into approaches for implementing post-quantum and hybrid key exchange and authentication in TLS 1.2, TLS 1.3, and SSH, but they are certainly not exhaustive. Standards bodies employing hybrid cryptography will have to make choices for the various design considerations discussed in this document, and may make different choices depending on their scenarios. The case studies revealed some challenges in TLS and SSH standards and implementations with respect to limits on message

sizes for key exchange and signatures that may affect some Round 2 submissions. Some size limits were implementation-imposed limits, and increasing those implementation-specific limits within standards-compliant limits enabled some additional schemes to function. Note that increasing those limits may have deleterious effects on performance on resistance to denial of service attacks, and should be investigated further. Some size limits were imposed by the standards; in some cases implementations experimented with increasing those size limits beyond the standards, and had some preliminarily positive results, but doing so would in deployment require carefully negotiating the use of larger length fields, and still risks compatibility with middle boxes and other pieces of the infrastructure. A few failures in the experimental results reported are not fully diagnosed, and may be resolvable with further engineering effort. The experimental results to date from the case studies explored the size impacts of post-quantum KEMs and post-quantum signature schemes independently. An obvious next step is to evaluate the O(n 2) combinations of KEMs and signatures for compliance with standards and implementation constraints. The case studies above omitted some Round 2 submissions that have not yet been incorporated into their underlying frameworks. The OQS team intends to extend the OpenSSL and OpenSSH implementations described in this report to include all Round 2 KEMs and signature schemes. The first step is to get all Round 2 KEMs and signature schemes into liboqs, which OQS is working towards with the help of the PQClean project [25], and we welcome external contributors. Once in liboqs, the algorithms will be enabled in the OpenSSL and OpenSSH forks. Beyond compliance with standards and implementation constraints, future steps include bench- marking network performance: - Network performance in lab conditions: Following the methodology of [8, 9, 19], how does latency and throughput behave on isolated networks in the lab? - Network performance in more realistic conditions: Attempt to develop a simulation that reflects network conditions described by Langley [29] to assess latency and throughput in more realistic network conditions. There are many other network protocols and applications also of interest for the post-quantum transition. end 06

8 Appendix: user manual?

BOH

References

[1] W.Diffie, M.E.Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, Vol. IT-22, No. 6, November 1976, pp. 644-654, DOI 10.1109/TIT.1976.1055638

[2] S.Bone, M.Castrom, "A Brief History of Quantum Computing"

[3] D.J.Bernstein, D.Hopwood, A.Hulsing, T.Lange, R.Niederhagen, L.Papachristodoulou, M.Schneider, P.Schwabe, Z.Wilcox-O'Hearn, "SPHINCS: practical stateless hash-based signatures", February 2, 2015

[4] J.P.Aumasson, D.J.Bernstein, C.Dobraunig, M.Eichlseder, S.Fluhrer, S.L.Gazdag, A.Hulsing, P.Kampanakis, S.Kolbl, T.Lange, M.M.Lauridsen, F.Mendel, R.Niederhagen, C.Rechberger, J.Rijneveld, P.Schwabe, "SPHINCS+ Submission to the NIST post-quantum project", March 14, 2019

[5] D.J.Bernstein, A.Hulsing, S.Kolbl, R.Niederhagen, J.Rijneveld, P.Schwabe, "The SPHINCS+ Signature Framework", September 23, 2019

[6] S.Kolbl, "Putting Wings on SPHINCS", DTU Compute, Technical University of Denmark, Denmark

[7] P.Kampanakis, D.Sikeridis, "Two Post-Quantum Signature Use-cases: Non-issues, Challenges and Potential Solutions", November 21, 2019

[8] E.Crockett, C.Paquin, D.Stebila, "Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH", July 19, 2019

[9] G.Endignoux, "Design and implementation of a post-quantum hash-based cryptographic signature scheme", July 2017

[10] A.Hulsing, J.Rijneveld, P.Schwabe, "ARMed SPHINCS Computing a 41 KB signature in 16 KB of RAM"

[11] D.Butin, J.Walde, J.Buchmann, "Post-Quantum Authentication in OpenSSL with Hash-Based Signatures", TU Darmstadt, Germany

[12] D.Stebila, M.Mosca, "Post-Quantum Key Exchange for the Internet and the Open Quantum Safe Project", July 28, 2017

[13] M.J.Kannwischer, A.Genet, D.Butin, J.Kramer, J.Buchmann, "Differential Power Analysis of XMSS and SPHINCS"

[14] A.Genet, M.J.Kannwischer, H.Pelletier, A.McLauchlan, "Practical Fault Injection Attacks on SPHINCS"

[15] L.Castelnovi, A.Martinelli, T.Prestm, "Grafting Trees: a Fault Attack against the SPHINCS framework"

[16] D.J.Bernstein, "Introduction to post-quantum cryptography", Department of Computer Science, University of Illinois at Chicago

[17] P.W.Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", August 30, 1995, DOI 10.1137/S0097539795293172

[18] Prashant, "A Study on the basics of Quantum Computing", December 2005

[19] ETSI. ETSI TC Cyber Working Group for Quantum-Safe Cryptography, https://portal.etsi.org/TB-SiteMap/CYBER/CYBER-QSC-ToR, Accessed April 18, 2020

[20] G.Alagic, J.M.Alperin-Sheriff, D.C.Apon, D.A.Cooper, Q.H.Dang, C.A.Miller, D.Moody, R.C.Peralta, R.A.Perlner, A.Y.Robinson, D.C.Smith-Tone, Y.K.Liu, "Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process", NIST, January 31, 2019

[21] NIST. Post-Quantum Cryptography, https://csrc.nist.gov/projects/post-quantum-cryptography, Accessed April 18, 2020

[22] B.Kommadi, Post Quantum Cryptography Algorithms, https://medium.com/datadriveninvestor/post-quantum-cryptography-algorithms-c510d31ae52c January 1, 2019

[23] J.Ding, B.Y.Yang, "Multivariate Public Key Cryptography", University of Cincinnati and Technische Universitat Darmstadt, Academia Sinica and Taiwan InfoSecurity Center, Taipei, Taiwan

[24] J.Patarin, "Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new Families of Asymmetric Algorithms", Eurocrypt, 1996

[25] A.Kipnis, J.Patarin, L.Goubin, "Unbalanced Oil and Vinegar Signature Schemes", April 15, 1999, DOI 10.1007/3-540-48910-X_15

[26] R.Lindner, C.Peikert, "Better Key Sizes (and Attacks) for LWE-Based Encryption" November 30, 2010

[27] R.Overbeck, N.Sendrier "Code-based cryptography" In: Bernstein D.J., Buchmann J., Dahmen E., "Post-Quantum Cryptography", 2009, pp. 95-145, DOI 10.1007/978-3-540-88702-7_4

[28] N.Sendrier, "Code-Based Cryptography", 2011, DOI 10.1007/978-1-4419-5906-5

[29] R.J.McEliece, "A Public-Key Cryptosystem Based on Algebraic Coding Theory", Communications Systems Research Section, February 1978

[30] H.Dinh, C.Moore, A.Russell, "McEliece and Niederreiter Cryptosystems That Resist Quantum Fourier Sampling Attacks", Advances in Cryptology - CRYPTO 2011, 2011, DOI 10.1007/978-3-642-22792-9_43

[31] N.Sendrier, "McEliece Public Key Cryptosystem", 2011, DOI 10.1007/978-1-4419-5906-5

[32] R.C.Merkle, "A Certified Digital Signature", 1979

[33] L.Lamport, "Constructing Digital Signatures from a One Way Function" SRI International Computer Science Laboratory, October 18, 1979

[34] J.Buchmann, E.Dahmen, A.Hulsing, "XMSS - A Practical Forward Secure Signature Scheme based on Minimal Security Assumptions", Second Version, November 26, 2011

[35] A.Huelsing, D.Butin, S.Gazdag, J.Rijneveld, A.Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC-8391, May 2018

[36] J.Katz, Y.Lindell, "Introduction to Modern Cryptography, Second Edition", Section 5.4.1, pp. 164-168

[37] F.Song, "A Note on Quantum Security for Post-Quantum Cryptography", pp. 246-265, 2014

[38] C.Zalka, "Grover's quantum searching algorithm is optimal", February 20, 1998, DOI 10.1103/PhysRevA.60.2746

[39] D.McGrew, M.Curcio, S.Fluhrer, "Leighton-Micali Hash-Based Signatures", RFC-8554, April 2019

[40] P.Benioff, "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines", Journal of Statistical Physics, pp. 563-591, May 1980, DOI 10.1007/BF01011339

[41] IBM Quantum, https://www.ibm.com/quantum-computing/ Accessed April 23, 2020

[42] S.Kundu, R.Kundu, S.Kundu, A.Bhattachaijee, S.Gupta, S.Ghosh, I.Basu, "Quantum computation: From Church-Turing thesis to Qubits", IEEE, October 2016, DOI 10.1109/UEMCON.2016.7777805

[43] L.M.K.Vandersypen, M.Steffen, G.Breyta, C.S.Yannoni, M.H.Sherwood, I.L.Chuang, "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance", pp. 883-887, 2001, DOI 10.1038/414883a

[44] C.Y.Lu, D.E.Browne, T.Yang, J.W.Pan, "Demonstration of a Compiled Version of Shor's Quantum Factoring Algorithm Using Photonic Qubits", 2007, DOI 10.1103/PhysRevLett.99.250504

[45] B.P.Lanyon, T.J.Weinhold, N.K.Langford, M.Barbieri, D.F.V.James, A.Gilchrist, A.G.White, "Experimental Demonstration of a Compiled Version of Shor's Algorithm with Quantum Entanglement", 2007, DOI 10.1103/PhysRevLett.99.250505

[46] E.M.Lopez, A.Laing, T.Lawson, R.Alvarez, X.Q.Zhou, J.L.O'Brien, "Experimental realization of Shor's quantum factoring algorithm using qubit recycling" October 21, 2012, DOI 10.1038/nphoton.2012.259

[47] G.M.Zaverucha, D.R.Stinson, "Short One-Time Signatures", July 26, 2010

[48] J.Buchmann, E.Dahmen, S.Ereth, A.Hulsing, M.Ruckert, "On the Security of the Winternitz One-Time Signature Scheme", 2011

[49] J.Pieprzyk, H.Wang, C.Xing, "Multiple-Time Signature Schemes against Adaptive Chosen Message Attacks"

[50] A.Hulsing "WOTS+ - Shorter Signatures for Hash-Based Signature Schemes", September 28, 2017, DOI 10.1007/978-3-642-38553-7

[51] L.Reyzin, N.Reyzin, "Better than BiBa: Short One-time Signatures with Fast Signing and Verifying", April 30, 2002

[52] O.Goldreich, "Foundations of Cryptography: Volume 2 Basic Applications", Cambridge University Press, 2004