Master thesis
Computer Science

Radboud University

Post-quantum blockchain using one-time signature chains

Author:
Wouter van der Linde
wlinde@science.ru.nl

First supervisor/assessor:
dr. Peter Schwabe
peter@cryptojedi.org

External supervisors:
dr. Andreas Hülsing
andreas@huelsing.net

dr. Yuval Yarom
yval@cs.adelaide.edu.au

Second assessor:
prof. dr. Lejla Batina
lejla@cs.ru.nl

August 27, 2018

Abstract

Recent advances in quantum computing seem to suggest it is only a matter of time before general quantum computers become a reality. Because all widely used cryptographic constructions rely on the hardness of problems that can be solved efficiently using known quantum algorithms, quantum computers will have a profound impact on the field of cryptography. One such construction that will be broken by quantum computers is elliptic curve cryptography, which is used in blockchain applications such as bitcoin for digital signatures. Hash-based signature schemes are a promising post-quantum secure alternative, but existing schemes such as XMSS and SPHINCS are impractical for blockchain applications because of their performance characteristics. We construct a quantum secure signature scheme for use in blockchain technology by combining a hash-based one-time signature scheme with Naor-Yung chaining. By exploiting the structure and properties of a blockchain we achieve smaller signatures and better performance than existing hash-based signature schemes. The proposed scheme supports both one-time and many-time key pairs, and is designed to be easily adopted into existing blockchain implementations.

Acknowledgements

I would like to thank my supervisors Peter Schwabe, Andreas Hülsing and Yuval Yarom, for their guidance and support while I was working on this thesis. Our discussions always inspired me to improve, and to try again when I was stuck. I am also very grateful to Piotr Narewski, whose implementation of bitcoin I used, for his time and effort to answer my many questions, and to Lejla Batina for taking the time to act as second reader. Finally, I thank my girlfriend, family, and friends for reading various drafts, providing valuable feedback, helping me clear my head from time to time, and for supporting me during my studies. Thank you all.

Contents

Chapter 1

Introduction

Many of the world's largest and most influential technology companies have been actively developing quantum processors, racing to be the first to achieve quantum supremacy [1]. On January 8 2018, Intel announced their 49-qubit quantum test chip [26], which would allow researchers to further examine and improve error correction techniques and simulate computational problems. On March 5, Google presented Bristlecone, their new 72-qubit quantum processor, stating they are 'cautiously optimistic that quantum supremacy can be achieved with Bristlecone' [27]. The existence of a powerful quantum computer has serious consequences for cryptography: already in 1994 Shor showed that a sufficiently powerful quantum computer can break RSA, Diffie-Hellman key exchange, and Elliptic Curve cryptography [41], which currently form the cryptographic foundations of the internet. While there is a large gap between reaching quantum supremacy and breaking RSA (which would require thousands of qubits [4]), it is important to prepare for a world where quantum computers are a reality; we need time to design post-quantum secure schemes, to improve their efficiency and usability, and to build confidence in their security [6].

Quantum computing also has significant consequences for blockchain technology, which includes crypto-currencies such as Bitcoin, Ethereum and Monero. This field has rapidly gained popularity and mainstream attention: on December 17 2017, a bitcoin's value reached an all-time high of $\$19,783.06$ (USD), while only a year before a bitcoin was worth less than a thousand dollars [19]. However, since elliptic curve cryptography is used to prove ownership of funds, the original bitcoin design is not post-quantum secure. It is crucial for the longevity of bitcoin and other blockchain technologies to adopt post-quantum secure signature schemes.

An example of a post-quantum secure crypto-currency is the Quantum Resistant Ledger (QRL) [36], which uses the hash-based signature scheme

[1] Quantum supremacy denotes the ability of a quantum computer to perform tasks surpassing what can be done with classical computers.

XMSS. XMSS however has some downsides. One is its long key generation time: generating a public key that can be used for 2^{16} signatures already takes about 20 seconds, and having more signatures available only increases this. Although 2^{16} signatures will be enough for most users, a party like Wikipedia that depends on donations will (hopefully) need far more. However, XMSS' main problem when used in blockchain applications is its signature size, which is about 2.5KB with the parameters that the QRL uses. When signatures grow larger, less transactions fit in a block, which slows down the already low rate at which transactions are being processed.

Our goal. In this thesis we develop a quantum secure signature scheme for use in blockchain technologies, based on one-time signatures combined with Naor-Yung chaining. Compared to XMSS, we intend this scheme to have smaller signatures, more efficient key generation with higher signature capacity, and faster signing and verification times, while (or as a consequence of) being conceptually easier. Finally, it should be easy to integrate into existing blockchain implementations. The signature scheme we propose is designed for blockchain applications in general, but to limit the scope of this thesis we focus on its application in bitcoin specifically.

Chapter 2

Cryptographic Background

In Chapter 4 we present a new hash-based signature scheme for use in blockchain technologies. This chapter provides the necessary cryptographic background, starting with cryptographic hash functions, pseudo-random functions and digital signature schemes. We then take a brief look at post-quantum cryptography, and discuss the construction and development of hash-based signature schemes. Finally we discuss one such scheme, W-OTS+, in detail; it is used as a building block for the scheme we propose in this thesis.

In the following, $x \xleftarrow{\$} X$ means that x is randomly chosen from X.

2.1 Cryptographic Hash Functions

Hash functions map inputs of arbitrary size to smaller fixed-size outputs. They are often used to allow efficient lookups in unsorted data, by using the output of a hash function as a key to the data. Such a data structure is called a hash table or hash map, and is available in most programming languages. Cryptographic hash functions combine this mapping property with the concept of a one-way function (OWF), which is efficient to compute but hard to invert: when given an output y of a OWF f, it is practically infeasible to find an input x (a pre-image) such that $f(x) = y$. With 'practically infeasible' we mean that there is no known efficient algorithm. In the rest of this thesis we use the terms 'hash function' and 'cryptographic hash function' interchangeably to refer to the latter.

A cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^n$ thus maps inputs of arbitrary length to fixed-size outputs (often called *digests*) of n bytes, in a way that is practically infeasible to invert. Informally, cryptographic hash functions adhere to the following properties:

- Given an output $y = H(x)$, it is practically infeasible to find the pre-image x.

- Given an input x_1, it is practically infeasible to find a second pre-image x_2 such that $x_1 \neq x_2$ and $H(x_1) = H(x_2)$.
- It is practically infeasible to find a collision: two arbitrary inputs x_1 and x_2 such that $H(x_1) = H(x_2)$ and $x_1 \neq x_2$.

Formally, these properties hold if the chance that any probabilistic polynomial algorithm outputs a pre-image, second pre-image or collision respectively is negligible. However, the above description of a cryptographic hash function does not satisfy this; if we create an algorithm that chooses random outputs and simply returns these, there will be an instance that always produces a collision with a probability of one. To prevent this issue, cryptographic hash functions are more formally defined as follows (we give only the most popular properties, see [39] for a comprehensive overview):

Definition 2.1.1 (Cryptographic hash function)**.** *A cryptographic hash function is a function $H : \{0,1\}^m \times \{0,1\}^* \to \{0,1\}^n$ that, given an m-byte key and an input of arbitrary size gives an n-byte output, such that*

- *(Pre-image resistance) for any key $k \xleftarrow{\$} \{0,1\}^m$ and output $y = H(k,x)$, finding x is practically infeasible,*
- *(Second pre-image resistance) for any key $k \xleftarrow{\$} \{0,1\}^m$ and input $x_1 \in \{0,1\}^*$ such that $H(k,x_1) = y$, finding an input x_2 such that $x_1 \neq x_2$ and $H(k,x_2) = y$ is practically infeasible,*
- *(Collision resistance) for any key $k \xleftarrow{\$} \{0,1\}^m$, finding two inputs x_1 and x_2 such that $x_1 \neq x_2$ and $H(k,x_1) = H(k,x_2)$ is practically infeasible.*

By varying the function key, different instantiations of the hash function are obtained. The set of all such variations is often called a function family, denoted as $\mathcal{H} : \{H : \{0,1\}^m \times \{0,1\}^* \to \{0,1\}^n\}$.

Although the differences between these properties are subtle, their distinction is vital. In a group of people, it is in fact far more likely that there is a random pair of people that share the same birthday (a collision) than that there is a person that shares your birthday (which is similar to finding a second pre-image). This is known as the birthday paradox. For a cryptographic hash function with n-bit outputs, finding a (second) pre-image requires a brute force attack with time-complexity $\mathcal{O}(2^n)$ [1], but finding a collision takes $\mathcal{O}(2^{n/2})$ using the so-called birthday attack. Therefore, an application that relies on collision resistance of a hash function would require a longer hash output than one that relies on (second) pre-image resistance to achieve the same security level.

[1] We are referring to general attacks on cryptographic hash functions. Using cryptanalysis it might be possible to find more efficient attacks for specific hash functions, as in [40].

2.2 Pseudo-Random Functions

Another class of functions that is often used in the following sections is that of pseudo-random functions (PRFs). Intuitively, a PRF is a function that cannot be distinguished from any other function with the same range and domain. Formally, an adversary gets access to a black box $\texttt{Box}$ that is initialized as a function from either a family $\mathcal{P}$ or from the set $\mathcal{G}(m,n)$ of all functions with domain $\{0,1\}^m$ and range $\{0,1\}^n$. The goal of the adversary is to distinguish both cases. The probability of success for adversary $\mathcal{A}$ is defined as:

$$\begin{aligned}\mathrm{Succ}_{\mathcal{P}}^{PRF}(\mathcal{A}) = \big|&\Pr[\texttt{Box} \xleftarrow{\$} \mathcal{P} : \mathcal{A}^{\texttt{Box}(\cdot)} = 1] \\ &-\Pr[\texttt{Box} \xleftarrow{\$} \mathcal{G}(m,n) : \mathcal{A}^{\texttt{Box}(\cdot)} = 1]\big|\end{aligned}$$

The definition of a pseudo-random function is then as follows:

Definition 2.2.1 (Pseudo-random function)**.** *We call $\mathcal{P} : \{P : \{0,1\}^m \times \{0,1\}^* \to \{0,1\}^n\}$ a pseudo-random function family if for all probabilistic polynomial time adversaries $\mathcal{A}$, running in time polynomial to n, the maximum success probability is negligible:*

$$\max\{\mathrm{Succ}_{\mathcal{P}}^{PRF}(\mathcal{A})\} = \mathrm{negl}(n)$$

2.3 Digital Signature Schemes

When signing a contract, for example to rent an apartment, the signer commits him- or herself to its contents because they are (or should be) the only one able to draw their own signature. In the digital world, digital signatures provide the same assurances by using public-key cryptography, which uses a pair of keys: a private key that is known only to the key pair's owner, and a public key that is publicly known. In digital signature schemes, the private key is used by the signer to create a digital signature for a given message, and the public key can be used by anyone to verify the resulting signature for that message. As long as the private key remains secret, a digital signature provides *authentication* (the message was created or approved by the signer), *non-repudiation* (the signer cannot later deny having signed the message, since only the signer has access to the private key), and *integrity* (the message was not altered after it was signed). The only way for an attacker, who does not own the private key, to create a valid signature would be to forge one.

Definition 2.3.1 (Digital signature scheme)**.** *A signature scheme is defined as a triple of algorithms $(Kg, Sign, Vf)$ for a message space $\mathcal{M}$ (which is often defined as the output range of a hash function), where*

- *$Kg(1^n)$ generates a key pair (sk, pk) given a security parameter n;*
- *$Sign(sk, M)$ creates a signature σ on M using the secret key sk;*
- *$Vf(pk, \sigma, M)$ returns 1 if σ is a valid signature on message M for public key pk;*

such that for all $(sk, pk) \leftarrow Kg(1^n)$ and all messages $M \in \mathcal{M}$, every signature $\sigma \leftarrow Sign(sk, M)$ can be verified with $Vf(pk, \sigma, M) = 1$.

The standard security notion for signature schemes is Existential Unforgeability under Chosen Message Attacks (EU-CMA) [21], which means that an adversary $\mathcal{A}$ successfully breaks the scheme if she manages to forge a valid signature for any message, while being able to obtain one or more valid message and signature pairs. This is formally defined using the following experiment (where $\text{Dss}(1^k)$ denotes a signature scheme with security parameter k and q the maximum number of signature queries):

Experiment $\text{Exp}^{\text{eu-cma}}_{\text{Dss}(1^k)}(\mathcal{A})$
$(sk, pk) \leftarrow Kg(1^n)$
$(M^*, \sigma^*) \leftarrow \mathcal{A}^{Sign(sk,\cdot)}(pk)$
Let $\{(M_i, \sigma_i)\}_1^q$ be the results of all queries to $Sign(sk, \cdot)$
Return 1 iff $Vf(pk, \sigma^*, M^*) = 1$ and $(M^*, \sigma^*) \notin \{(M_i, \sigma_i)\}$

The success probability for an adversary $\mathcal{A}$ is then

$$\text{Succ}^{\text{eu-cma}}_{\text{Dss}(1^k)}(\mathcal{A}) = \Pr\big[\text{Exp}^{\text{eu-cma}}_{\text{Dss}(1^k)}(\mathcal{A}) = 1\big],$$

and EU-CMA security of a digital signature schemes is defined as follows:

Definition 2.3.2 (EU-CMA)**.** *Let $k \in \mathbb{N}$. A digital signature scheme $Dss(1^k)$ is EU-CMA secure if for all q, t polynomial in k the success probability of any $\mathcal{A}$ running in time $<t$ and making at most q queries to Sign is negligible in k:*

$$\max\{\text{Succ}^{\text{eu-cma}}_{\text{Dss}(1^k)}(\mathcal{A})\} = \text{negl}(k)$$

One-time signature schemes A recurring concept in this thesis is that of a One-Time Signature (OTS) scheme, which is a signature scheme that is secure when using a key pair to sign a message only once. Since creating a signature with an OTS scheme often means revealing some part of the private key, using the same key pair to sign multiple messages degrades the security guarantees, making it more likely that an attacker can forge a signature. Note that the above definition of EU-CMA holds for an OTS if the amount of queries the adversary can make to the signing oracle is limited to one.

Digital signature schemes are widely employed on the internet, for example to prove authorship of files, ownership of cryptographic public keys, and integrity of software distributions. However, as we will see in the next section, the longevity of currently used schemes is not guaranteed.

2.4 Post-Quantum Cryptography

Already in 1994 Shor showed that by using the unique properties of quantum computers it is possible to construct algorithms that solve discrete logarithms and integer factorizations in polynomial time [41]. Shor's algorithms will significantly change the field of cryptography: once a sufficiently powerful quantum computer is developed, it renders all currently widely used schemes based on public key cryptography (such as RSA, ECDSA, Diffie-Hellman key exchange etc.) insecure. Luckily we are still years away of actually building such a machine, and nothing has been broken yet. Cryptographers have been working hard on developing alternatives which include lattice-based cryptography [1][38][42], code-based cryptography [31][28][9][7], multivariate public key cryptography [15], and hash-based signatures [11][24][8][25]. All these alternatives rely on different problems, which are believed to resist attacks by quantum computers as well.

In the case of hash-based signatures, that problem is breaking one or more of the three properties of cryptographic hash functions explained in Section 2.1. Although there exists a quantum algorithm, called Grover's algorithm, that can be used to find pre-images of hash functions faster than classical computers can [22], the speedup is far less severe: Grover's algorithm finds a pre-image of an n-bit hash with time complexity $\mathcal{O}(\sqrt{2^n})$, providing a quadratic speedup compared to $\mathcal{O}(2^n)$ for a classical computer. Since for an n-bit hash output Grover's algorithm on average needs $\sqrt{2^n} = 2^{n/2}$ tries, it can be counteracted by simply doubling the output length of the used hash function.

In this thesis we expand the literature on hash-based signature schemes by proposing a new scheme, designed specifically for use in blockchain applications. For those unfamiliar with hash-based signature schemes we give a short historic overview in the next section.

2.5 Hash-Based Signature Schemes

Almost 40 years ago the first hash-based signature scheme was proposed, but only recently they have become practical enough to be used in general applications. In this section we give a short (and incomplete) historic overview of hash-based signature schemes [2].

[2]For a more complete history, see https://pqcrypto.org/hash.html and https://huelsing.wordpress.com/hash-based-signature-schemes/literature/.

Lamport one-time signature scheme. The first hash-based signature scheme was published in 1979 by Lamport [30]. His idea was to selectively reveal pre-images of the outputs of a one-way function f (which can be instantiated with a hash function), depending on the bits of the message to sign. To sign n-bit messages we generate n pairs of random values (k_{i_0}, k_{i_1}) to get the sequence $(k_{1_0}, k_{1_1}, \ldots, k_{n_0}, k_{n_1})$; this sequence is our private key. The public key is created by applying f to each of the private key values, $(f(k_{1_0}), f(k_{1_1}), \ldots, f(k_{n_0}), f(k_{n_1}))$, resulting in the sequence $(p_{1_0}, p_{1_1}, \ldots, p_{n_0}, p_{n_1})$.

To sign an n-bit message M, we divide it into a string of individual bits $(m_1, m_2, \ldots, m_n)$ (note that M can be the output of a hash function with n-bit outputs). After generating a key pair and distributing the public key, M can be signed by selectively revealing parts of the private key. For every message bit m_i, we reveal k_{i_0} if $m_i = 0$, and k_{i_1} if $m_i = 1$. The resulting signature $(s_1, s_2, \ldots, s_n)$ thus consists of exactly half of the private key values. To verify such a signature, we start by picking one value of every pair in the public key, using the message bits to determine which: if m_i is 0, we pick p_{i_0}, and p_{i_1} otherwise. We then apply f to every value in the signature, and check whether the resulting list matches the public key values we picked.

While this scheme is very fast, it has some drawbacks that make it impractical for general use. The first is that it is an OTS scheme. If the same key pair is used to sign two different messages, both parts of one or more private key pairs will be revealed, which allows an attacker to forge a signature. Another issue is that signatures and keys are very large. The private key values must be large enough to prevent an attacker from simply iterating over all possible values. Assuming we make these 256 bits long and use a one-way function with 256-bit outputs (to allow many different messages to be signed), a signature would be $256 \cdot 256 \approx 8.2$ KB, and our private and public keys $2 \cdot 256 \cdot 256 \approx 16.4$ KB.

Merkle trees. In a 1982 patent [32] Merkle describes a structure that allows many one-time signatures to be associated with one public key. He called this structure an 'authentication tree', but it is commonly referred to as a Merkle tree or hash tree, and the resulting scheme as the Merkle Signature Scheme (MSS).

To create a Merkle tree, a signer first generates N OTS key pairs, where N is a power of two. Assuming a hash function F [3], all N public keys are 'compressed' into one by building a binary tree, starting at the leaf nodes. For every OTS public key pk_i, the signer creates a leaf node $h_i = F(pk_i)$. The value of a parent node is then obtained by applying F to the concatenation

[3] In [33], a paper published several years after his patent, Merkle actually describes F as an efficient, compressing, and collision resistant one-way function.

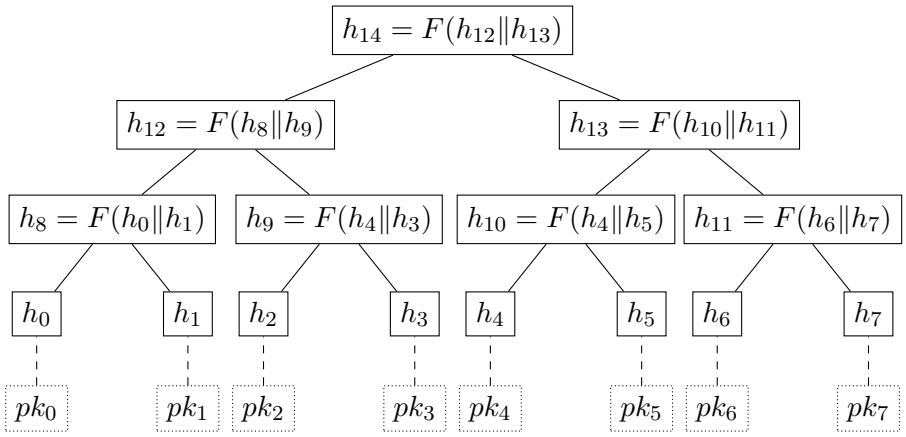

Figure 2.1: A Merkle tree with 2^3 OTS key pairs.

of that parent's child nodes. By applying this rule recursively the signer ends up with one root node, which they make public and use as a 'many-time' public key. See Figure 2.1 for an example.

A signer creates an MSS signature by choosing a leaf node that has not been used before, and creates a signature using the underlying OTS scheme with the key pair of that leaf. To prove that the key pair used for the signature is part of a Merkle tree with the long-term public key as root node, the signer includes an *authentication path* along with the signature, which contains the index of the used leaf node, and the shortest list of nodes that enables a verifier to compute the root node of the tree. An example of such an authentication path is illustrated in Figure 2.2. When using the key pair with public key pk_1 to create the signature sig, the path thus includes the index 1 (the left-most leaf having index 0). To allow a verifier to compute the root node of the tree, we must also include h_0, h_9 and h_{13}: the signer thus publishes $(sig, pk_1, 1, h_0, h_9, h_{13})$. A verifier can now compute $h_1 = F(pk_1)$, $h_8 = F(h_0 \| h_1)$, $h_{12} = F(h_8 \| h_9)$ and $h_{14} = F(h_{12} \| h_{13})$, and finally checks whether the result matches our many-time public key.

Merkle already recognized that a large tree takes a long time to generate. This is because all key pairs and nodes in the tree must be computed to obtain the many-time public key. The more signatures we want available, the bigger our tree has to be.

A large tree would also take a lot of storage space, but fortunately it is not necessary to store the entire tree: we can deterministically generate the private key values using a pseudo-random number generator and a short seed value, reducing the required storage to one short seed value at the cost of some computation time.

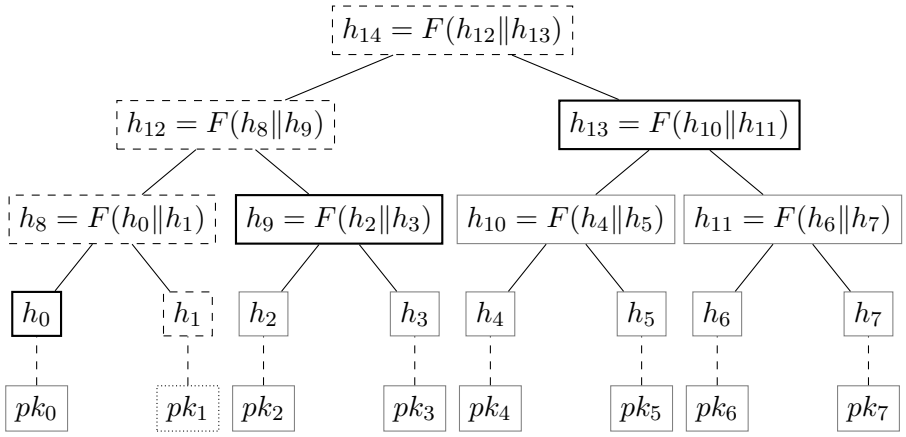

Figure 2.2: A Merkle tree authentication path example. When creating a signature with the key pair of pk_1, the authentication path consists of the nodes with black borders. The nodes with dashed borders are computed by a verifier, and those with gray borders remain known only to the signer.

Winternitz OTS. Following his patent, Merkle published his authentication tree in 1989 [33]. In the same paper he describes an idea by Winternitz for an improved one-time signature scheme. Winternitz proposed to iterate a hash function f, creating function chains such as $f(f(f(x))) = f^3(x)$ (note that $f^0(x) = x$), using chunks of message bits as iteration counts. The size of these chunks determines the maximum chain length: if we split a message into 4-bit chunks, we get a maximum chain length w (called the Winternitz parameter) of $2^4 = 16$. This allows us to sign multiple message bits with a single private-key value, as opposed to Lamport's scheme where we sign one message bit with one private key value.

A Winternitz OTS (W-OTS) private key is a list of (pseudo-)randomly generated values (not pairs). The corresponding public key values pk_i are obtained by iterating f on every private key value k_i $w - 1$ times, such that $pk_i = f^{w-1}(k_i)$. To create a signature, we reveal intermediate chain values: if the first four message bits were 1001 (thus using $w = 16$), we would reveal $f^9(k_0)$ as the first part of our signature. A verifier checks this part of the signature by evaluating f another $15 - 9 = 6$ times to obtain $f^6(f^9(k_0)) = f^{15}(k_0)$, and making sure the result matches pk_i.

This scheme trades signature size for increased computational effort. Longer chains result in shorter signatures (since we split m into fewer chunks) but require more evaluations of f. In fact, a linear decrease in signature size results in an exponential increase in evaluations of f. However, since f is (often) designed to be fast and easy to compute this is still a worthwhile trade-off.

There is one obvious vulnerability in this scheme, which is that when given for example the signature value $f^5(k_i)$, which signs the message chunk

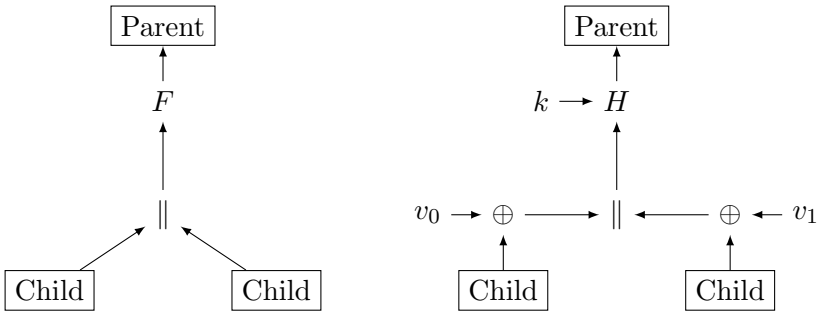

Figure 2.3: An illustration of the differences between MSS (on the left) and SPR-MSS (on the right, where H is a keyed hashed function, k the key, and v_0 and v_1 bit-masks).

0011, an attacker can easily compute $f(f^5(k_i))$, thus creating a signature for the chunk 0100. This is solved by also signing a checksum that becomes invalid when the signature is changed.

SPR-MSS. In 2005, Garcia proved that the security of Merkle's signature scheme depends on collision resistance of the hash function F and EU-CMA of the used OTS [18]. In the same year however, attacks were published on the collision resistance of MD5 [45] and SHA1 [44], which were quite popular at the time. Thus, Dahmen, Okeya, Takagi and Vuillaume set out to adapt MSS to rely on second pre-image resistance of a hash function instead of collision resistance [13]. The resulting scheme changes the hash function F in MSS to a keyed cryptographic hash function H_k, and computes parent nodes a bit differently: before appending child node values and computing H_k on the result, the child node values are XOR'ed with bit-masks (see Figure 2.3 for an illustration). Both the key k and the bit-masks are fixed during key generation and become part of the public key. The result is a scheme that has a higher security level and shorter signature size than MSS, but larger public keys.

W-OTS$^{\mathbf{PRF}}$. A slightly adapted version of the Winternitz OTS was proposed by Buchmann, Dahmen, Ehreth, Hülsing and Rückert in 2011 [10]. Their version changes the hash function f to a keyed PRF f_k that is also used slightly different. When iterating f_k, they use the output of the previous iteration as the key k for the current one. Every iteration thus uses a different key, but the same input. The original Winternitz scheme was proven to be secure if the hash function f is collision resistant and undetectable (its output indistinguishable from a purely random output) [16]. Buchmann et al. prove that their scheme is secure without having to rely on collision resistance. This allows W-OTS$^{\text{PRF}}$ to have significantly shorter signatures than the original W-OTS at the same level of security (unfortu-

nately, the given proof turned out to be flawed in 2017, making W-OTS$^{\text{PRF}}$ insecure in practice [29]).

XMSS. By combining the ideas from SPR-MSS and W-OTS$^{\text{PRF}}$, Buchmann, Dahmen and Hülsing construct the eXtended Merkle Signature Scheme (XMSS) [11]. By using SPR-MSS, the tree construction of XMSS relies on a second pre-image resistant keyed function H_k, and by using W-OTS$^{\text{PRF}}$ the only other security requirement is a PRF. The authors prove that these two requirements are minimal (that theoretically, these are the only two requirements for the existence of a secure signature scheme), and that if both the hash function and PRF are efficient, XMSS is as well. Furthermore, they prove that XMSS is forward secure: if an attacker at some point finds the private key, they are still not able to create forgeries for signatures that were created before they found the private key. With a signature size of less than 25% of SPR-MSS and a slightly higher security level, XMSS can be considered the first practical hash-based signature scheme.

W-OTS+. SPR-MSS replaced the need for a collision resistant hash function with a second pre-image resistant one by using keyed hash functions and bit-masks. Hülsing applied a comparable approach to W-OTS to achieve shorter signatures for the same (pre-quantum) security level (compared to W-OTS$^{\text{PRF}}$). Starting from the original W-OTS, we replace the hash function f with a keyed hash function f_k that is second pre-image resistant, one-way and undetectable. During key generation we fix a key k and randomization elements $r = (r_1, \ldots, r_j)$. Now, every time we apply the function f_k, we first compute the bitwise XOR of the input of the function and one of the randomization elements, $f_k(x \oplus r_i)$ for some i. By using W-OTS+ in XMSS, the signature size can be further reduced by 50% (compared to W-OTS$^{\text{PRF}}$) at a security level of 80 bits.

SPHINCS. One major drawback of XMSS is that it is stateful: it is vital to remember which one-time key pairs were already used to create a signature. Because of this, it is not always possible to replace RSA or ECDSA with XMSS. Also, when a previous backup of the key state is restored, or when the key state is lost, the security of stateful schemes deteriorates quickly. Because of this state, using the same key pair on different devices is also difficult to manage. In 2015, SPHINCS was created to 'eliminate the state' [8] and to be a drop-in replacement for current signature schemes such as RSA and ECDSA. This is achieved by generating a variant of a Merkle tree so that by randomly selecting leaf nodes to sign with, the chance of using the same leaf node more than once is sufficiently small to be practically non-existent. Already in 1986 Goldreich [20] proposed a stateless hash-based signature scheme by using a very large tree where every node

was not the result of hashing two child nodes, but instead an OTS signature on those children. As a result, Goldreich's scheme produces extremely large signatures since the authentication paths consisted of many OTS signatures. SPHINCS solves this issue by constructing a tree where every node is in fact a Merkle tree itself: the leaf nodes contain OTS key pairs that are used to sign the root of other sub-trees. SPHINCS introduces several more improvements, but for the sake of brevity we refer those interested to the original paper [8]. While SPHINCS does not rely on private key state, it has a large signature size (41 KB) and long computation times. Two improvements on SPHINCS, called SPHINCS+ and Gravity-SPHINCS, have been submitted to the Post-Quantum Cryptography project by NIST (https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions).

WOTS-T & XMSS-T. In 2016 Hülsing, Rijneveld and Song identified the issue of multi-target attacks on hash functions in relation to XMSS and W-OTS(+). These schemes base their security on pre-image resistance of a cryptographic hash function, for which the attack complexity is $\mathcal{O}(2^n)$ for n-bit outputs. However, this assumes the hash function is used only once, while for XMSS and W-OTS(+) an attacker can obtain many hash function outputs, and can possibly break the security of these schemes if they succeed in finding just one pre-image. When such a multi-target attack targets d outputs, the attack complexity degrades to $\mathcal{O}(\frac{2^n}{d})$, which has serious consequences in practice when deciding on parameters to use.

To prevent such attacks on XMSS and W-OTS+, the authors propose to use different keys and bit-masks for every hash function call, making each one unique. To avoid making public keys even larger, these keys and bit-masks are computed deterministically using a PRF. The resulting schemes XMSS-T and WOTS-T thus resist multi-target attacks (both in the classical and post-quantum settings), at a minor performance cost, and were standardized in RFC 8391 in May 2018 [23].

2.6 W-OTS+ in Detail

The signature scheme we propose in Section 4 uses an OTS as a building block. We chose to use W-OTS+ (including the adaptations from WOTS-T to prevent multi-target attacks) as its security is well understood and because it allows us to control the trade-off between performance and signature size. At the time of writing it is also the only standardized OTS scheme [23]. This section describes W-OTS+ in detail, as described in RFC 8391. As we use W-OTS+ as a sub-routine rather than a standalone signature scheme, the key generation and signing algorithms use different inputs than specified in Defition 2.3.1. Finally, we provide an algorithm that computes a W-OTS+ public key from a signature rather than a verification algorithm.

In the following log denotes the base-2 logarithm.

Parameters and functions. W-OTS+ uses two parameters. The security parameter n determines the size of private key, public key and signature elements, and is usually (but not necessarily) also used as the length of messages that can be signed (which are usually hash digests). The Winternitz parameter w determines the trade-off between size and computation effort. These parameters are used to calculate the variables len, len_1 and len_2. Here len denotes the amount of function chains used, which when multiplied with n gives the total length of a private key, public key and signature. len_1 determines the number of chains gathered from an input message, and len_2 the number gathered from a checksum. They are determined as follows:

$$\begin{aligned} len_1 &= \lceil 8n / \log w \rceil \\ len_2 &= \lfloor \log (len_1 \cdot (w-1)) / \log w \rfloor + 1 \\ len &= len_1 + len_2 \end{aligned}$$

Recommended values for w are 4 and 16 as these provide good trade-offs between size and performance, but other values are possible as well. We also need a second pre-image resistant hash function F, which takes an n-byte input and n-byte key to compute an n-byte output, and a pseudo-random function PRF that maps an n-byte key and 32-byte address value (which is defined in the next paragraph) to an n-byte output. Both can be instantiated using a cryptographic hash function such as SHA256 or SHA3-256[4].

To protect against multi-target attacks, all hash function calls are randomized using pseudo-random keys and bit-masks. We first describe how these are generated, then define the W-OTS+ chaining function and the algorithms for key generation, signing and verification.

Addresses, keys and bit-masks. Since we need the same keys and bit-masks used to create a signature to verify it, they must be generated deterministically. This is achieved by using the addressing structure depicted in Figure 2.4, which we from now on refer to as an OTS address. The 'layer', 'tree' and 'OTS' fields specify the position of an OTS key pair in a larger tree structure (as used by XMSS), and are all set to 0 when no such structure is used. The type field is also set to 0, indicating that the address belongs to an OTS hash [5]. The 'Chain', 'Hash' and 'KeyAndMask' fields are used internally in W-OTS+. We use dot notation to refer to a specific field, i.e. a.Chain where a is an OTS address.

[4]For more information on the security requirements of these functions we refer to Section 9 of RFC 8391.

[5]RFC 8391 specifies two other address structures with different type values. These are only used in XMSS and thus not relevant for W-OTS+.

Layer address (32 bits)
Tree address (64 bits)
Type = 0 (32 bits)
OTS address (32 bits)
Chain address (32 bits)
Hash address (32 bits)
KeyAndMask (32 bits)

Figure 2.4: A W-OTS+ OTS address.

Keys and bit-masks are generated with the function PRF, using an OTS address as the 32-byte address input and a pseudo-randomly generated n-byte public seed as the n-byte key. While the OTS address guarantees different keys and bit-masks for every call to the hash function F, the public seed guarantees a different set of keys and bit-masks for every W-OTS+ key pair. Depending on the value of the 'KeyAndMask' field of the used address, PRF produces a hash function key or bit-mask.

Chaining function. We define the chaining function $C^{i,j}(x, a, ps)$ with an n-byte input x, an amount of iterations i, a starting index j, an OTS address a and a public seed value ps, as follows. If $i = 0$, C returns x. Otherwise, given that a_h denotes address a with a.Hash $= j + i$, we compute a key k and bit-mask r as

$$k = PRF(ps, a_h) \text{ with } a_h.\text{KeyAndMask} = 0,$$
$$r = PRF(ps, a_h) \text{ with } a_h.\text{KeyAndMask} = 1,$$

and define C recursively it as

$$C^{i,j}(x, a, seed) = F(k, C^{i-1,j}(x, a_h, ps) \oplus r).$$

The chaining function thus first generates a key k and bit-mask r, then takes the bitwise XOR of the previous iteration $C^{i-1,j}(x, a, ps)$ and r, and computes F using key k on the result.

Key generation. The algorithm $Kg(s, a, ps)$, given an n-byte secret seed s, an OTS address a and an n-byte public seed ps, generates a key pair (sk, pk). The private key $sk = (sk_1, \ldots, sk_{len})$ consists of n-byte values that are pseudo-randomly generated from s. To reduce the amount of storage required it is regenerated from s when needed. A public key pk of $n \cdot len$ bytes is then computed as

$$pk = (pk_1, \ldots, pk_{len}) = (C^{w-1,0}(sk_1, a_{c_1}, ps), \ldots, C^{w-1,0}(sk_{len}, a_{c_{len}} ps))$$

where a_{c_i} denotes an address a with a.Chain $= i$.

Signing. The signing algorithm $Sign(s, M, a, ps)$ computes a signature σ for an n-byte message M, given an n-byte secret seed s, an OTS address a and an n-byte public seed ps. First, the message is mapped to len_1 chain lengths by dividing its binary representation into len_1 groups of bits $(m_1, \ldots, m_{len_1}) = m$. Each m_i is then interpreted as a natural number between 0 and $w - 1$. The idea is to treat M as the binary representation of a natural number, and transform that to a base-w representation. This results in a list of natural numbers between 0 and $w - 1$, which can be used as chain lengths. A checksum c is computed as $c = \sum_{i=1}^{len_1}(w - 1 - m_i)$, and its binary representation similarly interpreted as len_2 chain lengths $(c_1, \ldots, c_{len_2}) = c$. The full list of len chain lengths b is then obtained by concatenating these two lists so that $b = (b_1, \ldots, b_{len}) = m \| c$.

After computing the private key $sk = (sk_1, \ldots, sk_{len})$ from s, a signature σ of $n \cdot len$ bytes is computed as

$$\sigma = (\sigma_1, \ldots, \sigma_{len}) = (C^{b_1,0}(sk_1, a_{c_1}, ps), \ldots, C^{b_{len},0}(sk_{len}, a_{c_{len}}, ps))$$

where a_{c_i} again denotes an address a where $a.\text{Chain} = i$. A signature thus consists of partially evaluated chains on the private key.

Verification. Lastly we define the algorithm $PkFromSig(\sigma, M, a, ps)$ that computes a W-OTS+ public key from signature σ of $n \cdot len$ bytes on an n-byte message M, given an OTS address a and public seed ps. First, *PkFromSig* computes the chain lengths in exactly the same way as when creating a signature: M is mapped to len_1 chain lengths, the message checksum is computed and converted to len_2 chain lengths, and the two are concatenated to obtain $b = (b_1, \ldots, b_{len})$. Then a public key pk is computed from σ by completing the partial chains in the signature:

$$pk = (pk_1, \ldots, pk_{len}) = (C^{w-1-b_1,b_1}(\sigma_1, a_{c_1}), \ldots, C^{w-1-b_{len},b_{len}}(\sigma_{len}, a_{c_{len}})).$$

Finally, pk is returned. Checking the validity of pk is thus delegated to the caller.

Chapter 3

Background on Bitcoin

Having discussed the cryptographic background, we now turn to the second topic of this thesis. In the following sections we provide an introduction to the design and security of bitcoin, and describe some implementation details that are relevant to our work. We then discuss the consequences of quantum computing on bitcoin's security.

3.1 Design and Security

Bitcoin was designed by Nakamoto in 2008 [34] as a decentralized electronic currency. There is no trusted third party that processes every transaction; instead it uses a peer-to-peer network where the transaction history is determined by decree of the majority. Transactions are recorded in a public ledger, called a blockchain, which is constructed in such a way that it is practically infeasible for malicious users to make changes. Every user maintains a copy of the blockchain, and communicates changes to its peers. A blockchain is essentially a database maintained by an online community that determines the database's contents by consensus among users.

Transactions. Nakamoto defined an electronic coin as a chain of digital signatures. Each signature signs the hash of the previous link and the public key of the new owner of the coin. Each link can thus be seen as a transaction of the coin. To transfer a coin to someone else, the current owner adds a link to this chain, thus creating a new transaction: the owner signs the hash of the current signature and the public key of the payee. See Figure 3.1 for an illustration of such a coin.

Such a transaction however would only be able to transfer a single 'coin', and using a separate transaction for every cent of a larger payment would not be practical. To solve this a transaction can be further broken down into inputs and outputs. Inputs 'collect' funds from outputs of previous transactions, and outputs make funds available to subsequent inputs, speci-

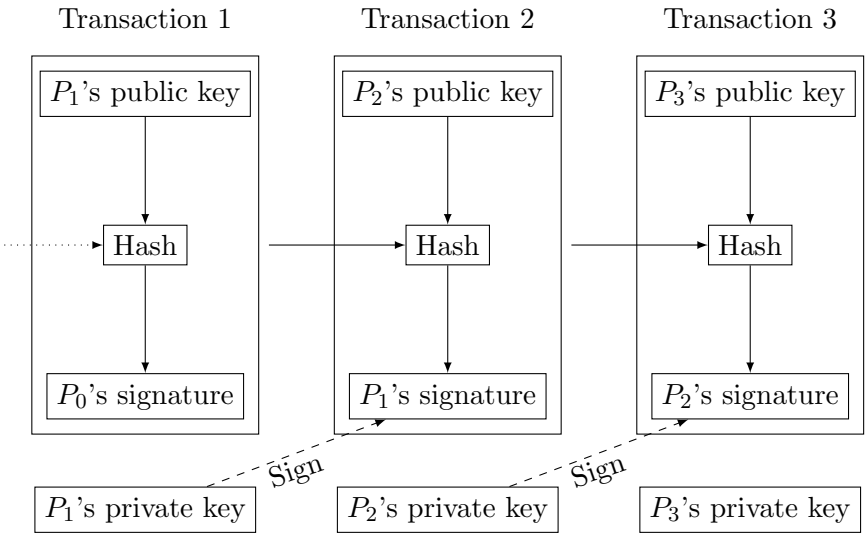

Figure 3.1: An illustration of a series of transactions as defined in [34]. After Transaction 1, P_1 is the owner of the coin. P_1 sends the coin to P_2 by using their private key to sign the hash of Transaction 1 and P_2's public key.

fying certain conditions on how they can be claimed (Nakamoto did not go into detail on how to construct inputs and outputs; see Section 3.2 for more details).

The hardest part of developing an electronic currency system is to prevent double spending. While it is impossible in the physical world to give away the same coin twice, a digital bit string is not lost after it is used to pay. As argued in [34], the only way to know whether a transaction already exists is to be aware of all transactions. Since we want to avoid having to trust a third party, this means all transactions must be publicly announced, and that all participants in the network must agree on a single transaction history.

Blocks and proof-of-work. In bitcoin the transaction history is defined as a chain of blocks, where each block contains the hash of the previous block in the chain, a list of transactions, and a random nonce. To add a block to the chain, a user must provide a proof of work: only blocks with hashes that start with a certain amount of zeros are accepted by the network. Users are thus asked to find a pre-image for a partial hash output, which (for a pre-image resistant hash function) should require a brute-force search. The amount of zero's required, often called the *difficulty*, can be varied depending on the total hashing power of the network.

To add a block to the chain (given the hash of the last block in the chain and a list of transactions to include) a user repeatedly increments

the new block's nonce until they find one such that the block's hash has enough leading zeros. For a secure hash function, the chance of finding a number that results in a hash with n leading zeros is on average $\frac{1}{2^n}$. If n is large enough, finding a suitable nonce using a single computer could take days, months, or even years. However, if many users are searching for such a nonce, the chance that one of them will find one quickly is significantly higher.

To give users an incentive to create new valid blocks, the first transaction in every block is a special one that 'creates' a new coin. While this may seem like free money, users have to expend resources to create new blocks. In bitcoin, these resources are CPU/GPU time and thus electricity. Another incentive is the transaction fee: part of the funds collected by a transaction's inputs is reserved as payment to the user that includes it in a new block. Once a predetermined amount of coins exist, finding a block no longer generates new coins and the transaction fee becomes the only incentive to find new blocks.

Consensus. Occasionally multiple users might find a new block (roughly) simultaneously. In this scenario, the blockchain forks into multiple branches. Each user continues with whichever of these conflicting blocks they received first, but will switch to the other branch if it becomes longer earlier. The bitcoin consensus rule thus says that the longest chain is always right. Long forks, where for multiple branches new blocks are found simultaneously multiple times in a row, are very unlikely; and if they do occur, they will eventually resolve to one chain. However, when receiving payment via bitcoin, it is a good idea to wait for the transaction to be confirmed by several blocks (3 confirmations is considered highly reliable [1]) before acting on it to avoid a situation where the transaction was included in a branch that is later abandoned.

Malicious users that want to double spend their coins have to change or undo a transaction in the blockchain after receiving what they initially paid for. Every block that is added to the chain makes it harder for a malicious user to make changes to previous blocks. Because every block in the chain contains the hash of the previous one, a change in one block affects all blocks after it as well. Therefore a malicious user must redo not only the proof of work for the changed block, but also for every block that came after it. While an adversary is re-doing all this work, the honest users are adding blocks to the 'honest' chain. As long as more than half of the processing power in the network is controlled by honest users, the 'honest' chain will most likely grow faster than a fork created by any adversary.

[1] See https://bitcoin.org/en/you-need-to-know#instant

3.2 Relevant Implementation Details

Although Nakamoto mentioned that transactions could be built using inputs and outputs, he did not describe how these would work in detail. Below we discuss how these are implemented, and how to pay or receive payment using bitcoin addresses. We also discuss the different software components of the bitcoin network, as well as a protocol that was recently added to bitcoin, *segregated witness*, which makes the scheme we propose in Chapter 4 significantly more practical. We only describe those details that are relevant to our work; for more information on the implementation of the P2P network, smart contracts, and other details we refer to bitcoin's developer guide (https://bitcoin.org/en/developer-guide).

Bitcoin addresses. An address can be seen as the bitcoin equivalent of a bank account: if Alice wants to send some bitcoins to Bob, she needs his bitcoin address, which is a base58-encoded string $<version \parallel hash \parallel csum>$ consisting of a version number, a hash, and a checksum. The version number describes how the hash should be used (more on this in the following paragraphs), and the checksum is there to avoid transmission errors.

Transactions, inputs and outputs. A transaction is a tuple $(ins, outs)$ of a list of inputs ins and a list of outputs $outs$ with at least one each (we exclude some fields that are not relevant for this thesis), and is identified by a transaction identifier (txid) defined as its hash. An output is a tuple $(val, script_{\mathcal{PK}})$ of a bitcoin value val and a public key script $script_{\mathcal{PK}}$. The public key script specifies conditions that must be met to spend the output's value, which are described using a simple scripting language. An input, which is a tuple $(txid, idx, script_{\mathcal{S}})$ claims funds of the output with index idx in a transaction with txid $txid$, by providing a signature script $script_{\mathcal{S}}$ that contains data that satisfies the conditions described in the specified output's $script_{\mathcal{PK}}$. A signature script is verified by pushing its data onto a stack, then executing the corresponding public key script. We describe several script types below.

Until an output is spent, it is marked as an *Unspent Transaction Output* (UTXO). All UTXOs are kept in a UTXO database until they are spent, so that checking whether a specific output has already been spent is trivial: if an input tries to spend funds of an output that is not in the database, the corresponding transaction is rejected.

P2PKH script. An output with a Pay-to-Public-Key-Hash (P2PKH) public key script requires a claimant to prove they own a key pair of which the public key hashes to a certain value. Let us assume that Alice wants to pay Bob half of a bitcoin for trimming her backyard, and that Bob would

like to collect his payment using a P2PKH script (shown in Listing 3.1). Bob first generates a key pair and computes a cryptograhic hash of his public key $H(pk_{Bob})$. He then constructs a bitcoin address using this public key hash and a version prefix of 1 $<1 \parallel H(pk_{Bob}) \parallel csum_{Bob}>$, and sends it to Alice. She decodes the address to extract Bob's public key hash, creates a P2PKH script $script_{\mathcal{PK}}^{Bob}$, and uses it to construct a transaction output $out_A = (0.5, script_{\mathcal{PK}}^{Bob})$. Finally, she includes out_A in a transaction tx_A (such that $tx_A.outs[i] = out_A$ for some index i) with txid $txid_A$, and broadcasts it over the network.

```
OP_DUP OP_HASH160 <pubkey hash> OP_EQUALVERIFY OP_CHECKSIG
```

Listing 3.1: A P2PKH pubkey script

```
<signature> <public key>
```

Listing 3.2: A P2PKH signature script

To claim his coins, Bob must create an input with a valid signature script. For a P2PKH script, it consists of a signature σ_{Bob} created with Bob's private key and his full public key (see Listing 3.2). By default the signed data consists of all inputs (excluding signature scripts) and outputs of the transaction. Bob thus creates the input $(txid_A, i, \sigma_{Bob}\ pk_{Bob})$. Verifying Bob's signature script starts by pushing both the signature and full public key it contains onto the stack. We then execute the operations of the public key script (illustrated in Table 3.1), starting with `OP_DUP` which duplicates the item currently on top of the stack. `OP_HASH160` computes a hash of the top stack item, consuming the original value. After this a public key hash (the one Alice extracted from Bob's address) is pushed onto the stack. `OP_EQUALVERIFY` then checks whether the two topmost items on the stack are equal, removing both from the stack. If not, script verification fails. Finally `OP_CHECKSIG` is executed, which interprets the two topmost items on the stack as a signature and full public key (in that order, consuming both), then checks whether the signature is valid for the given public key.

P2SH script. The bitcoin scripting language can be used to create complex scripts, but it would be impractical to distribute custom scripts to spenders instead of a short, well defined bitcoin address. Moreover, public key scripts are created by spenders who do not necessarily care about what a script does, while receivers often do. Pay-to-Script-Hash scripts (P2SH) were created to solve these problems. A receiver creates a P2SH address by creating the custom script they want to use, which is called the *redeem script*, hashing it, and including that hash in a standard bitcoin address with the version prefix 3. A spender that receives such an address creates a P2SH public key script as follows:

Stack	Script
sig - pk_A	OP_DUP OP_HASH160 <pk_B hash> OP_EQUALVERIFY OP_CHECKSIG
sig - pk_A - pk_A	OP_HASH160 <pk_B hash> OP_EQUALVERIFY OP_CHECKSIG
sig - pk_A - pk_A hash	<pk_B hash> OP_EQUALVERIFY OP_CHECKSIG
sig - pk_A - pk_A hash - pk_B hash	OP_EQUALVERIFY OP_CHECKSIG
sig - pk_A	OP_CHECKSIG
1	-

Table 3.1: An example of a successful P2PKH script evaluation, where the signature script elements have already been pushed to the stack.

```
OP_HASH160 <redeem script hash> OP_EQUAL
```

Listing 3.3: A P2SH public key script

A receiver then claims their funds by creating a signature script that includes the redeem script and any data required to verify it. An example of a more complex script for which P2SH addresses are used (and one we refer to later in this thesis), is a *multisig script*.

Multisig Script. Multisig scripts can be used to require signatures from multiple parties to spend an output. A multisig redeem script has the following structure, where m denotes the amount of required signatures and n the number of given public keys:

```
<m> <pubkey A> [pubkey B] [pubkey C...] <n> OP_CHECKMULTISIG
```

Listing 3.4: Multisig redeem script

To spend an output with a P2SH multisig script, a spender must provide m signatures that match m of n given full public keys along with the original redeem script. A full multisig signature script, including the data required for verification, thus looks like the following, where the OP_0 is included because of an off-by-one error in the original bitcoin code that needed to be preserved for backwards compatibility:

```
OP_0 <sig A> [sig B] [sig C...] <m> <pubkey A> [pubkey B] [
    pubkey C...] <n> OP_CHECKMULTISIG
```

Listing 3.5: Multisig redeem script with data

Stack	Script
OP_0 - sig A - OP_1 pk_A pk_b OP_2 OP_CHECKMULTISIG	OP_HASH160 <hash> OP_EQUAL
OP_0 - sig A - redeem hash	<hash> OP_EQUAL
OP_0 - sig A - redeem hash - hash	OP_EQUAL
OP_0 - sig A - 1	-
OP_0 - sig A	OP_1 <pk_A> <pk_b> OP_2 OP_CHECKMULTISIG
...	...
OP_0 - sig A - 1 - pk_A - pk_b - 2	OP_CHECKMULTISIG
1	-

Table 3.2: An example of a successful 1-of-2 P2SH multisig script evaluation. After restoring the redeem script as public key script, we skip a few pushes to the stack for the sake of brevity.

A P2SH multisig script is then verified as illustrated in Table 3.2. First the multisig redeem script is hashed and compared to the script hash in the public key script. Then script verification is performed a second time, using the redeem script as public key script.

Clients, wallets and miners. Creating a bitcoin address is done using a *wallet program*. Wallets store the private key for all addresses they create, and are thus also used to sign transactions; or in other words to create valid signature scripts for public key scripts that send funds to an address controlled by the wallet. Since wallet programs do not need to be connected to the bitcoin network to generate addresses or sign transactions, and because they store private keys, they can and should be kept on a system that is not directly connected to the internet.

The networking components of bitcoin are performed by *clients* (sometimes called bitcoin nodes). When running a bitcoin client for the first time, it will download and verify all blocks in the blockchain from other active clients, after which it can be used to check the balance of a given address, distribute new blocks or transactions signed by a wallet, etc.

Mining software is used to find new blocks as quickly as possible. A mining program asks a bitcoin client for a block template, which contains a number of transactions that can be included in a new block, the current difficulty (how many leading zeros the next block must have), as well as other data required to create a new block, and then starts iterating a random nonce until it finds a block of which the hash satisfies the difficulty requirement (hash enough leading zeros). Once a block is found, the mining

software sends it to a client so it can be distributed through the network. To obtain their reward, miners include a special transaction called a coinbase transaction in every block they mine. A coinbase transaction contains one input (the content of which is completely ignored), and one output that sends the mining reward and any mining fees payed by other transactions included in the mined block to an address of the miner's choice.

To optimize their profits, miners should find the optimal combination of transactions in terms of size and fees; large transactions thus need higher fees than smaller ones to give miners an incentive to include them in new blocks. Thus, one of the obstacles for using hash-based signatures in bitcoin is that because of their large signature sizes the required transaction fees could become exorbitantly high. One of the goals of this thesis is thus to create a hash-based signature scheme with a smaller signature size than existing ones.

Segregated Witness. To conclude this section we take a brief look at *segragated witness* (segwit), a protocol upgrade designed to increase the amount of transactions that fit in a block and to solve the issue of transaction malleability. Changes in certain data, such as the signature script, do not change the outcome of a transaction (are malleable) but do change its txid: it is possible for a malicious miner to change an input's signature script, without invalidating the transaction but changing its txid. This can make it hard to monitor the status of a transaction by tracking its txid.

Segwit solves this malleability by moving signature scripts from inputs to a separate data field, called the *segregated witness* or *witness*, that is not used when computing the txid; a segwit transaction thus looks like $(ins, outs, witness)$. Since these segwit signature scripts are not part of the txid, they must be added to the block hash when a miner includes such transactions in a block. To do this the miner collects all segwit signature scripts in the block by concatenating the *witness* fields of all segwit transactions, computes a Merkle tree of their hashes, and puts the root of that tree $root_{wts}$ in the $script_{\mathcal{PK}}$ of a second output of the coinbase transaction $cbtx$ [2]:

$$cbtx.outs[1] = (0, root_{wts}).$$

As this output does not have any value and its script contains only data, it cannot be spend with future inputs.

Segwit also introduces a virtual block size, computed as $\lceil(\text{base size} \cdot 3 + \text{full size})/4\rceil$ where base size denotes the block size without witness data, and full size the block size including witness data. The $(\text{base size} \cdot 3 + \text{full size})$ part is called the block weight, and has to be smaller or equal to 4 MB. As a

[2] This is again a simplified explanation. For more details we refer to https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki#Commitment_structure.

result, a 71-byte (DER-encoded) bitcoin signature included as witness data only counts as 18 bytes for the virtual block size. This increases the amount of transactions that fit in a block, thus increasing the amount of transactions bitcoin can process per second and lowering the required transaction fee. Most importantly for us, segwit ameliorates the large signature sizes of hash-based signature schemes, making them much more practical in bitcoin: a typical W-OTS+ signature of 2144 bytes would count as only 536 bytes for the virtual block size when included as witness data.

3.3 Post-Quantum Secure Blockchain

The security of payments in bitcoin relies on the prevention of double spending and controlling who can spend what coins. The first is achieved using the hash-based proof-of-work described above, which relies on the assumption that finding a pre-image of a (partial) output of a cryptographic hash function is much more costly than the potential gains. As we discussed in Section 2.4, a sufficiently powerful quantum computer could use Grover's algorithm to find a pre-image of an n-bit hash function after $\mathcal{O}(2^{n/2})$ tries (on average). Therefore miners using a quantum computer would have an advantage because they can find pre-images faster than miners using a non-quantum computer. While the same can be said for miners using ASICs, the speedup that such specialized hardware provides is linear, while quantum computers (using Grover's algorithm) achieve a quadratic speedup. Solving this issue is an important part of making bitcoin post-quantum secure, but is out of scope for this thesis.

Access to coins is controlled using digital signatures. Currently bitcoin uses the elliptic curve algorithm secp256k1 [37], which can be broken by quantum computers using Shor's algorithm (again see Section 2.4). It must be replaced with a post-quantum secure signature scheme to make bitcoin post-quantum secure. Hash-based signature schemes are a good option because their security is well understood and they provide practical performance.

W-OTS+, XMSS and SPHINCS The obvious hash-based candidates are W-OTS+, XMSS and SPHINCS. However, W-OTS+ is not an ideal candidate because it is an OTS scheme: as users can only create one signature for any public key, only one-time addresses would be possible. This is impractical for many use cases (for example for parties relying on donations or have a high volume of transactions), and can lead to loss of funds if a signed transaction does not end up in the blockchain as creating another signature with the same key pair increases the chances that an adversary successfully creates a forgery. XMSS signatures are quite large: using Winternitz parameter $w = 16$, a tree height $h = 10$, and a 256-bit hash function,

a signature is 2500 bytes. This results in large transactions, which require large fees and reduce the amount of transactions that fit in a block. Furthermore, with these parameters we get $2^h = 2^{10} = 1024$ signatures for one public key, and larger trees result in longer key generation times and even bigger signatures. When using XMSS, users must remember to move their funds to another address before they run out of signatures. Another possible drawback is that XMSS is stateful. SPHINCS is stateless, but pays for it with even larger signatures of about 41 KB. As none of them are ideal candidates for use in blockchain applications, we propose a new hash-based signature scheme that is specifically designed for use in such applications.

Chapter 4

XNYSS

We now propose a new post-quantum secure signature scheme that is designed specifically for use in blockchain-based applications such as bitcoin: the eXtended Naor-Yung Signature Scheme (XNYSS). Before we define the algorithms for key generation, signing and verification, we describe how the scheme is designed by discussing the issues that arise when using an OTS scheme in the context of a cryptographic currency, and our solutions to these issues.

4.1 One-time addresses

All information stored in a blockchain is public. For blockchain-based cryptographic currencies this means that if someone uses one address for all of their transactions, everyone knows exactly how much money they have and how they are spending it. To gain some financial privacy it is recommended to use an address only once to receive payment, and once to spend it. This makes it significantly harder (but not impossible [2]) for an adversary to track your balance and spending behavior.

If a public key is to be used only once to collect payment, using a one-time signature scheme seems logical: just generate a key pair, send an address based on the public key to the spender, and later create a signature using the private key to spend the funds. Transactions, however, are not guaranteed to be adopted into the blockchain: this can take a long time if the transaction fee is too low, and can even fail if an invalid transaction is signed. If a one-time private key was used to sign an invalid transaction, that key cannot be used again securely because using an OTS private key more than once makes it easier for an attacker to forge a signature and thus potentially steal funds.

Multiple key pairs per address. To alleviate this problem we associate a bitcoin address with multiple OTS key pairs instead of one (see Figure

4.1). A signature for such an address would then have to match any of the given public keys. Thus, when using k key pairs to construct an address, a user has k chances to claim their funds. In practice this approach allows such a one-time address to be used for k transactions if all goes well. However, their purpose is to be used just once, with $k - 1$ backup options in case things go awry.

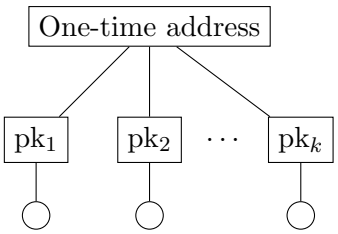

Figure 4.1: A one-time address based on k public keys. The white circles represent key pairs that can be used to sign.

Key state. It is vital that a wallet keeps track of which key pairs of an address were already used: if we use the same key pair twice (for example because the signed transaction was not adopted into the blockchain the first time), an attacker could succeed in forging a signature and attempt to claim our funds before we do. Thus after using a key pair to sign an input, it should be removed from the key state. Similarly, after a one-time address is successfully used to claim funds, all key state for that address can (and should) be removed, preventing it from being used again in the future.

4.2 Long-term addresses

There are situations where using a one-time address is not practical, and a long-term one is preferable. For example, it is not practical for parties that rely on donations, such as Wikipedia, to change their public address after every donation. Thus our scheme must also support long-term addresses, which is not possible using just an OTS scheme.

Naor-Yung chaining. To support long-term public keys we use Naor-Yung signature chains [35]: whenever a message m is signed, we also sign the hash of the public key we will use for our next signature, thus creating a chain of related signatures. The public key of the chain's start node is then used as a long-term public key and can be used to create a long-term bitcoin address. To verify a signature for a long-term public key, we check whether it is part of the corresponding signature chain, in other words whether its OTS public key was signed in a previous link of the chain. Naor-Yung

chaining is best known for providing *forward-security* to signature schemes, which means that if a private key is compromised it is still impossible to forge previously created signatures. This can be achieved when signing with Naor-Yung by deleting the current private key after creating a new signature, and replacing it with the one corresponding to the next public key.

Forking chains. When using signature chains in the context of a blockchain, there are still significant problems. For example, since bitcoin uses transaction fees as incentive for miners it can take a long time for a low fee transaction to be included in the blockchain. When using a single chain, no inputs can be signed until the previous signature in the chain is confirmed in the blockchain: clients that are not aware of the previous signature will reject the new one since they are missing a link in the chain. It is even worse if a signed transaction does not make it into the blockchain at all, which for example can happen when the fee is too small or the transaction is invalid: we would not be able to add a new node to our chain since we cannot use the last used key pair again.

To solve these problems, we fork the chain when creating a signature by signing b new public key hashes instead of one. The result is essentially a tree of signatures, where we can fall back to a previous fork whenever we break our current chain ('trimming' a branch of our tree). The signed public key hashes must be sent along with the signature for it to be verified: we denote such a signature for a long-term address σ_{lt} as a tuple of a one-time signature and a number of public key hashes:

$$\sigma_{lt} = (\sigma_{ots}, pkh_0, \ldots, pkh_{b-1})$$

Of course this scheme only works if we can assume that failed transactions are sufficiently rare. Every time a signed input is included in the blockchain, we gain $b-1$ key pairs to fall back on. Thus, if nine transactions succeed but the tenth fails, we would have $9 \cdot (b-1)$ key pairs to fall back on.

Slow start. As is, this construction suffers from a 'slow start': after creating an address, a user can create one transaction and then has to wait for it to be recorded in the blockchain before being able to sign another. We can alleviate this by using a similar approach as for our one-time addresses: instead of using one forked chain for a long-term address, we use k, so that after generating an address a user can already sign k transactions (see Figure 4.2 for an illustration of this construction). This also increases the amount of subsequent transaction failures after which a user runs out of available signatures; when one tree is exhausted they can continue signing with any of the others.

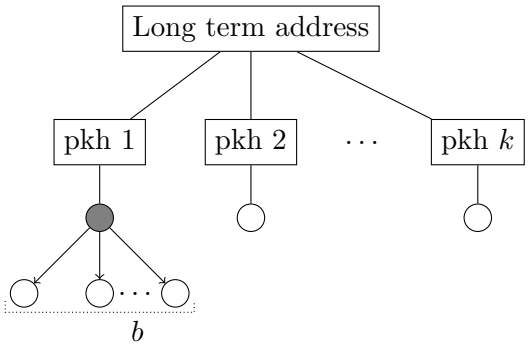

Figure 4.2: A long-term address using a multisig script that includes k public key hashes and creates b forks for every signature. The white circles represent unused key pairs, the gray circle a key pair that has already been used for a signature.

The slow start problem is further mitigated by the fact that we can sign multiple inputs in one transaction while consuming only one key pair in our tree. For the first input in a transaction we choose an available key pair and create a signature σ_{lt} that signs b new public key hashes along with the transaction. For any other inputs in the same transaction we can actually use those b new key pairs signed by σ_{lt} directly: since they are included in the same transaction, either all of the new public key hashes are confirmed when it is included in a new block, or they are all rejected. In the second case, we only used one available node; but if the transaction is adopted into the blockchain and we signed i inputs we gain $b + (b-1) \cdot (i-1)$ usable key pairs (see Table 4.1 for results using varying input counts and branching factors). Because of this property the slow start period is very short for users that collect funds in bulk.

#inputs	$b = 2$	$b = 3$	$b = 4$	$b = 5$
1	2	3	4	5
2	3	5	7	9
4	5	9	13	17
6	7	13	19	25
8	9	17	25	33

Table 4.1: Comparison of the amount of advertised public key hashes in a single transaction for various amounts of inputs and branching factors b.

Of course this does not solve the slow start issue completely. We could provide 'missing links' in signatures by including hashes and signatures for failed or unconfirmed transactions. However this would significantly increase

the signature size, decreasing the amount of transactions that fit in a block. Because the size of a W-OTS+ signature (which is 2144 bytes using $w = 16$ and $n = 32$) dominates the transaction size, each 'missing link' would take the place of a transaction with a single input. An attacker could try to abuse this in order to slow down the transaction rate, by including multiple missing links in every signature. Furthermore, including a 'missing link' effectively doubles the required transaction fee, making such transactions very expensive. In our opinion the benefits of including missing links do not justify the costs.

Signature verification. To verify a signature for a long-term address, we have to trace the used public key to one of the long-term public keys that are part of that address. Since the entire chain is stored in the blockchain, we can look up every previous link in the signature chain until we find one that was created with one of the root public keys. But because we are working with a blockchain, we only have to verify that the previous link in the chain (which advertised the public key of the current signature) is adopted into the blockchain and was used for the same address. If so, that previous link has already been verified as being valid for that address, which allows us to assume that the public key signed in that link is indeed part of a chain that can be traced back to the given long-term address. This makes signature verification in our scheme very efficient.

Stateful wallet. Because we sign with an OTS scheme, we need an up-to-date key state whenever we sign a new input. Despite the fact that we can only use key pairs of which the public key hash has been confirmed in the blockchain, we cannot derive our key state from the public ledger. This is because only accepted transactions are stored; rejected or pending ones are not. Therefore, the wallet needs to store the key state itself.

In the following we assume the key state to only contain information of unused OTS key pairs. Thus each state entry contains all information necessary to create a signature, which for W-OTS+ includes a private and public seed, as well as a confirmation status that indicates whether the public key hash of this key pair has been accepted into the blockchain. To update this confirmation status, the owner of a wallet has to periodically poll a bitcoin client.

Wallet backups and multi-device support. If a user loses their key state (for example because of hardware issues or file corruption), they cannot recover that state reliably: if at any time the user signed a failed transaction, any key state recovered from the blockchain would be incomplete and could result in a key pair being used more than once. However, restoring a full system backup with previous key state is very insecure because that could

restore previously used private keys. Since loss of key state likely results in loss of funds, we need a secure and reliable way to create key backups.

Fortunately we can easily create a backup of the key state by moving one or more state entries to a separate storage device (removing them from the original state). As these nodes are no longer in the active key state, they cannot be used to create signatures until that backup is restored. Of course this is only possible after a number transactions were successfully adopted into the blockchain. Thus when a user loses or corrupts their key state, they must restore such a key state backup with unused state entries before signing new transactions.

These backups can also be used to provide multi-device support: the user can create a 'backup' for every additional device they want to use as a wallet. These devices can then be used to sign transactions for the same long-term address simultaneously and independent of each other, since they do not share any state.

Preventing double key usage. After signing a number of transactions for the same address, multiple combinations of public keys and signatures will be publicly available. If an attacker finds a private key for any of them (which might happen many years from now), they can create a valid signature and claim any UTXO of the corresponding address. A similar issue occurs when a user signs two transactions with the same private key: this could allow an attacker to forge a signature for the corresponding key pair.

To prevent key reuse we propose that all bitcoin clients build and maintain an Unused Public Key Hash (UPKH) database, similar to the UTXO database. Assume a block is accepted into the blockchain that includes a transaction input $(txid, idx, script_{\mathcal{S}})$ where the signature script $script_{\mathcal{S}}$ contains a long-term signature $\sigma_{lt} = (\sigma_{ots}, pkh_0, \ldots, pkh_{b-1})$. The client then adds advertised public key hashes $pkh_0, \ldots, pkh_{b-1}$ to the database, such that when queried for one of them it returns the public key hash of the start node of the chain for which σ_{lt} is a signature. The client also removes the hash of the public key corresponding to σ_{ots} from the database. Finally, whenever the client verifies a transaction it checks whether the public key hashes corresponding to given one-time signatures are present in the UPKH database, except for those corresponding to the start of a chain which are included in their respective bitcoin addresses.

Another advantage of using such a UPKH database is that it reduces signature verification time: instead of searching through the blockchain for a certain UPKH, we look it up in a database (which can be done in constant time when using hash maps).

Revoking a UPKH. At some point there might be a UPKH in the database which should not be used for future signatures, for example because its private key was used to sign a failed transaction. The UPKH entries that would be removed by a failed transaction would remain in the database forever because the signer cannot use the corresponding private keys again. Therefore, it should be possible to revoke UPKHs from the database to remove redundant data from storage and to prevent an attacker that actually finds a private key from creating valid signatures. This could be done by including a revocation section in transactions where we list UPKHs to remove. Since UPKHs are hashes, their size does not significantly impact the total size of transactions, which is dominated by the signature size in our scheme. Of course the verifying client must check that revoked UPKHs belong to a chain owned by the signer: thus a UPKH can only be revoked if the transaction includes a signature for the same chain.

4.3 Full Scheme

Below we specify algorithms for key generation, signing and verification. Note that the algorithms below operate on signature chains, not on bitcoin addresses (which consist of k such chains). We define a cryptographic hash function H with output length n, and use the notation *wotsp.alg* to refer to an algorithm *alg* of W-OTS+ (see Section 2.6). We use the notation a_0 to denote an all-zero W-OTS+ address.

4.3.1 Parameters and Key State

The parameters of our scheme are a security parameter n in bytes (equal to the W-OTS+ parameter n) also used as the message digest length that can be signed, and a branching factor b that determines the amount of child nodes generated when creating a signature for a long-term public key. We also inherit the Winternitz parameter w from W-OTS+.

Because our scheme is stateful, it is vital to keep track of the key pairs that have already been used to sign. We store all nodes that can be used for new signatures, being nodes with public keys that were advertised in previous signatures. Our key generation creates a new key state, which consists of a flag *ots* that signals whether the given state belongs to a one-time (if true) or long-term (if false) public key, and a list of signature nodes. Each node contains

- an n-byte secret seed s,
- an n-byte public seed ps,
- a 32-byte txid *txid* of the transaction that advertises the node's public key, and

- a boolean confirmation status *confirm* that indicates whether the node can be used to create a signature.

The signing algorithm uses a key state to find a suitable node it can use to create a signature. By deleting a node from the key state after using it, we make sure that private keys are never used more than once. The size of a signature node in our key state is calculated as follows (assuming one byte of storage for confirmation status and $n = 32$):

$$\begin{aligned} &len(s \parallel ps \parallel txid \parallel confirm) \\ &\quad = 32 + 32 + 32 + 1 \\ &\quad = 97 \text{ B} \end{aligned}$$

Since the key state is quite small (99.3 KB for a state with 1024 available nodes) it should not be a problem for any modern device.

4.3.2 Algorithms

Below we define the key generation, signing and verification algorithms of XNYSS in detail. We deviate from our definition of a digital scheme (see Definition 2.3.1) for the following reasons.

Because our scheme has two modes of operation (one-time and long-term), *Kg* takes an additional argument *ots* that signals whether a one-time or long-term secret key state should be generated.

The *Sign* algorithm has an additional parameter *txid* which enables us to create signatures for inputs in the same transaction without waiting for confirmation. It also returns an updated secret key state *sk* in addition to the created signature. As such, our scheme formally classifies as a *Key Evolving Signature Scheme* [5], which uses a key update function after signing to update the secret key. We capture this behavior by removing nodes used to create a signature from the key state, and adding any generated child nodes.

To verify a signature, we do not need to follow a signature chain all the way to the root (as mentioned in 4.2). A W-OTS+ signature is verified by computing a public key from a signature and checking whether it matches a public key that was distributed earlier. In our case, the public key computed from a signature may not match the public key that it was created for, because a signature for a long-term public key is part of a chain of signatures. A signature is valid if the corresponding public key is either 1) advertised in a previously accepted transaction, 2) advertised in the same transaction, or 3) the root of a new chain. As there are multiple candidates a regenerated public key can match to be valid, we would have to call the verification algorithm multiple times. It is more practical and efficient to compute a public key from a given signature (which is possible when using W-OTS+), and let the blockchain application verify it for a given address. Because of this, we do not provide a full verification algorithm, but instead define

a function *PkFromSig* that computes the public key of a given signature. This public key is then checked against the three cases described above in the blockchain application using our scheme.

Key generation. We define the key generation algorithm $Kg(1^n, ots)$ (see Algorithm 1), given a security parameter n and boolean ots, to generate a key pair (sk, pk) where sk holds the state information as described above. The ots parameter determines whether a one-time (when ots is true) or long-term (when it is false) public key is created. The returned key state saves the value of ots, and contains one node (the start node of a new forked chain) that contains a random n-byte secret seed s, a random n-byte public seed ps, no txid, and a confirmation status set to true; it can thus be used to sign a transaction input immediately after key generation. We obtain pk by evaluating $wotsp.Kg(s, a_0, ps)$. The corresponding private key is pseudo-randomly generated from s when it is needed, and is thus not returned.

Algorithm 1 The XNYSS key generation algorithm

1: **procedure** $(sk, pk) \leftarrow Kg(1^n, ots)$
2: $\quad s, ps \xleftarrow{\$} \{0,1\}^n$
3: $\quad \_, pk \leftarrow wotsp.Kg(s, a_0, ps)$ ▷ with a_0 an all-zero address
4: $\quad nodes = \{(s, ps, txid := \text{null}, confirm := \text{true})\}$
5: $\quad sk := (ots, nodes)$
6: $\quad$ **return** (sk, pk)
7: **end procedure**

Signing. The signing algorithm $Sign(sk, M, txid)$ (see Algorithm 2), given a secret key state sk, an n-byte message M and transaction identifier $txid$, creates a signature and updates sk. First it picks a node from sk's node list. As mentioned before, after creating a signature for the first input of a transaction we can sign all other inputs in the same transaction using the child nodes generated for the first signature without waiting for confirmation. Thus *Sign* first checks if there is a node with a matching $txid$; if not, it does a second pass over the node list and picks the first already confirmed node. The algorithm fails if no suitable node can be found.

Having found a suitable signature node with secret seed s and public seed ps, a W-OTS+ signature σ_{ots} is created by calling $wotsp.Sign(s, m, a_0, ps)$. The value of m depends on the value of $sk.ots$. If it is true, $m = M$. When ots is false, b child nodes are generated. Their secret and public seeds are computed as $H(s \parallel r_1)$ and $H(ps \parallel r_2)$ respectively, where r_1 and r_2 are both n bytes of (pseudo) random data. The $txid$ field of these child nodes is set to $txid$, and their confirmation status to unconfirmed. Then their public keys pk_1 through pk_n are generated using W-OTS+ key generation,

Algorithm 2 The XNYSS signing algorithm

procedure $(sk, \sigma) \leftarrow Sign(sk, M, txid)$ ▷ where $sk = (ots, nodes)$
  $node_\sigma \leftarrow$ FindNode$(sk.nodes, txid)$
  if $node_\sigma =$ Fail **then**
    return Fail
  end if
  if $ots = true$ **then**
    for $i \in \{1, \ldots, b\}$ **do**
      $r_1, r_2 \xleftarrow{\$} \{0,1\}^n$
      $s_i \leftarrow H(node_\sigma.s \parallel r_1)$
      $ps_i \leftarrow H(node_\sigma.ps \parallel r_2)$
      $\_, pk_i \leftarrow wotsp.Kg(s_i, a_0, ps_i)$
      $node_i \leftarrow (s_i, ps_i, txid, \text{false})$
    end for
    $sk.nodes \leftarrow sk.nodes \cup \{node_1, \ldots, node_b\}$
    $m \leftarrow H(M \parallel H(pk_1) \parallel \ldots \parallel H(pk_b))$
    $pks \leftarrow (pk_1, \ldots, pk_b)$
  else
    $pks \leftarrow \emptyset$
    $m \leftarrow M$
  end if
  $\sigma_{ots} \leftarrow wotsp.Sign(node_\sigma.s, m, a_0, node_\sigma.ps)$
  $sk.nodes \leftarrow sk.nodes \setminus node_\sigma$
  return $(sk, (\sigma_{ots}, ps, pks))$
end procedure

function $node \leftarrow$ FindNode$(nodes, txid)$
  for all $N \in nodes$ **do**
    if $N.txid = txid$ **then**
      return N
    end if
  end for
  for all $N \in nodes$ **do**
    if $N.confirm = \text{true}$ **then**
      return N
    end if
  end for
  return Fail
end function

and finally M is defined as

$$M = H(m \parallel H(pk_1) \parallel \ldots \parallel H(pk_n)).$$

In order to verify the created signature σ, a verifier will need the used public seed and any child public key hashes generated when *ots* is false. Since we always use all-zero W-OTS+ addresses, a_0 does not need to be included. The signature, including data required for verification, is then defined as

$$\sigma = (\sigma_{ots}, ps[, H(pk_1), \ldots, H(pk_b)]).$$

See Table 4.2 for the total size of one-time and long-term signatures when using W-OTS+. After creating the signature, any generated child nodes are added to the key state, and the node used to sign is removed from it so it cannot be used again.

w	one-time	long-term		
	$b = 0$	$b = 2$	$b = 3$	$b = 4$
16	2176	2240	2272	2304
256	1120	1184	1216	1248

Table 4.2: Total signature size for one-time signatures and long-term signatures for various values of the branching factor b and Winternitz parameter w.

Verification. The algorithm $PkFromSig(sig, M)$ (see Algorithm 3) thus computes a public key, given an n-bit message M and signature σ that contains a W-OTS+ signature σ_{ots}, public seed ps and optionally public key hashes $H(pk_0), \ldots, H(pk_{b-1})$. First, *PkFromSig* defines a message m, in a way similar to *Sign*: if no public key hashes are included in σ, $m = M$, otherwise $m = H(M \parallel H(pk_0) \parallel \ldots \parallel H(pk_{b-1}))$. Then it computes the signature's public key using $wotsp.PkFromSig(\sigma_{ots}, m, a_0, ps)$.

4.3.3 Limitations

Signature capacity. Unlike XMSS, we do not have a fixed tree height that limits signature capacity. The only limiting factor on the amount of available signatures for a single public key is in fact the amount of public seeds we can generate. Since public seeds consist of 32 bytes, we have 2^{256} different seeds available. To prevent multi-target attacks from degrading the security level of our scheme, we should never use a public seed more than once. If we choose them at random, we expect a collision to occur after generating 2^{128} nodes due to the birthday paradox. Therefore, a long-term public key should not be used for more than 2^{128} signatures, which can be considered infinite for all practical use cases.

Algorithm 3 The XNYSS public-key regeneration algorithm

procedure $pk \leftarrow PkFromSig(\sigma, M)$ ▷ where $\sigma = (\sigma_{ots}, ps, pks)$
  if $\sigma.pks = \emptyset$ **then**
    $m \leftarrow M$
  else ▷ $pks = pk_0, \ldots, pk_{b-1}$
    $m \leftarrow H(M \parallel H(\sigma.pks.pk_0) \parallel \ldots \parallel H(\sigma.pks.pk_{b-1}))$
  end if
  $pk \leftarrow wotsp.PkFromSig(\sigma.\sigma_{ots}, m, a_0, \sigma.ps)$
  return pk
end procedure

Slow start. By using forking chains our scheme allows signing multiple transaction inputs without having to wait for confirmations. However, when starting a new signature chain tree we do not have any forks to fall back on yet. As it might take a while for transactions to be adopted into the blockchain, a user will have to wait a while before being able to create more signatures after using the start nodes of all k chains of an address.

Failed transactions. When a signed transaction is not adopted into the blockchain, for example because a user signs an invalid transaction or the fee is too low, we cannot reuse the used signature node (nor use any of its child nodes). Our construction mitigates the impact of this issue by having many signature nodes available, but if this happens early on in a public key's lifetime, the consequences could be severe. If worse comes to worst, there will be no more nodes available to create signatures, which could result in a loss of funds as the user is unable to collect them. Users should thus be careful about transaction validity and fees when they use a new public key.

4.4 Performance

We implemented XNYSS as described above (as well as W-OTS+ since we could not find a suitable standalone implementation) in Go using $n = 32$, $b = 3$, and $w \in \{16, 256\}$; it can be found at `https://github.com/lentus/xnyss`. Table 4.3 lists performance benchmarks and the sizes of signatures and public keys of our implementation; for comparison, Tables 4.4 and 4.5 show benchmarks and sizes for different XMSS implementations. The results in Table 4.4 were obtained using the Aidos Kuneen implementation of XMSS (`https://github.com/AidosKuneen/xmss`), which uses an optimized SIMD implementation of SHA256. Westerbaan's XMSS implementation (`https://github.com/bwesterb/xmssmt`), used to obtain the results in Table 4.5, uses hash precomputation and caching to optimize signing times specifically, but does not use an optimized implementation of SHA256. A typical bitcoin

client would use the Aidos Kuneen implementation for fast block verification, while a wallet program might use Westerbaan's for faster signing. We ran all benchmarks on a machine with an Intel Core i7-4770K CPU (4 cores @ 3491.79 MHz).

Optimizations. We implemented two major performance optimizations, the first of which is hash precomputation in W-OTS+ [1]. The keys and bit-masks in W-OTS+ are computed with a PRF, which is implemented as the SHA256 hash of 32 bytes of padding, a 32-byte public seed, and a full W-OTS+ address. The padding and public seed make up exactly one SHA256 digest, and since they do not change during a sign, verify or keygen operation, W-OTS+ recomputes the same intermediate hash digest many times. By precomputing this value we save a lot of hash computations: in our implementation this reduced signing, verification and key generation times by about 42%. The second optimization is to compute W-OTS+ hash chains concurrently by dividing the work over as many CPU cores as are available. We use the standard SHA256 implementation provided by the Go `crypto` package. Since our code spends about 75% of the time computing SHA256 hashes (determined using the Go profiling tool 'pprof'), any speedup for SHA256 directly results in a speedup of W-OTS+ and thus XNYSS.

	timings (ms)				signature sizes (bytes)	
w	sign one-t.	sign long-t.	verify	keygen	one-time	long-term
16	0.21	1.17	0.19	0.32	2176	2272
256	1.59	9.44	1.70	2.63	1120	1216

Table 4.3: Timing benchmarks and signature sizes for XNYSS, using $n = 32$ and $b = 3$. For signing we give both one-time and long-term benchmarks.

Results. In blockchain applications such as bitcoin, the most important performance measures for signature schemes are verification time and signature size: slow verification increases the time it takes to download the entire blockchain on client initialization, and bigger signature size decreases the amount of transactions that fit in a block. When using Winternitz parameter $w = 16$, signature verification is about 61.2% faster with XNYSS than with Aidos Kuneen's XMSS implementation, and signatures are almost 13% smaller than when using XMSS. We also achieve significantly faster key generation, and signing with a one-time key pair is 65.6% faster than Westerbaan's XMSS implementation. Signing for a long-term key pair is however 47.9% slower, but since signing time does not affect the performance of the

[1] Credit for this optimization goes to Westerbaan, who included it in his XMSS implementation.

	timings			signature size
#signatures	sign	verify	keygen	(bytes)
2^{10}	7.13 ms	0.49 ms	291.84 ms	2500
2^{16}	6.87 ms	0.50 ms	17.83 s	2692

Table 4.4: Timing benchmarks and signature sizes for the Aidos Kuneen XMSS implementation (using SHA256 and $w = 16$).

	timings			signature size
#signatures	sign	verify	keygen	(bytes)
2^{10}	0.61 ms	0.72 ms	386.98 ms	2500
2^{16}	0.61 ms	0.70 ms	20.47 s	2692
2^{20}	0.61 ms	0.80 ms	326.78 s	2820

Table 4.5: Timing benchmarks and signature sizes for Westerbaan's XMSS implementation (using SHA256 and $w = 16$).

network and because we can still create hundreds of signature per second, this is not a big issue.

By increasing the Winternitz parameter, it is possible to achieve significantly smaller signatures at the cost of performance. Using $w = 256$, signature size is lowered by 48.5% but signing, verification, and key generation times increase with factors of about 8, 9, and 8 respectively.

We chose to use a standard branching factor of 3 because it provides a good balance between signature size and the number of available signatures (based on Table 4.1). We must also keep in mind that all advertised public key hashes have to be saved in the UPKH database, which is another reason to keep b small. However, an instantiation of our scheme is not bound to one branching factor, and a wallet can (and should) choose to lower the branching factor when enough key pairs are available for signing. Wallets can use this feature to stop the key state from growing beyond a certain limit, which is very useful for wallets running on resource constrained devices.

Chapter 5

Implementation in Bitcoin

We modified an existing bitcoin implementation in Go (gocoin, https://github.com/piotrnar/gocoin) to use XNYSS. The resulting client supports addresses based on XNYSS public keys in addition to all existing address types. Below we describe how the new addresses are created, how the UPKH database is implemented, and describe how to implement our scheme using segregated witness. Our full implementation can be found at https://github.com/lentus/wotscoin.

5.1 Addresses and Scripts

As we describe in Chapter 4 we use multiple XNYSS key pairs to create an address, for both one-time and long-term addresses. We implement this using Bitcoin's (P2SH) *multisig scripts*, or m-of-n scripts, requiring only one matching signature ($m = 1$) instead of multiple. By reusing the existing multisig code we avoid having to add a completely new address type and can reuse much existing code.

XNYSS addresses. To create an XNYSS-based address, we generate the root nodes of k new XNYSS signature chains (in our implementation we fixed $k = 3$), and compute the hashes of their public keys. While normal multisig scripts contain full public keys, this is not practical when using XNYSS because of the large public keys of W-OTS+ (which are the same size as signatures). It is possible to use just hashes in our multisig scripts because when using W-OTS+ it is possible to compute a public key from a signature. We then create a normal multisig script with these k hashes, using the opcode `OP_CHECKXNYSSMULTISIG` instead of the original `OP_CHECKMULTISIG`, and hash the resulting script to create a script hash. The new opcode tells a verifier that an XNYSS signature is required to claim funds paid to this address, without requiring changes to existing data structures. We can then create a P2SH address, and use it to receive payment. The differences

between a normal multisig script and an XNYSS one are thus the use of public key hashes instead of full public keys, and the new opcode. Note that during address generation there is no difference between one-time and long-term addresses; this distinction is only relevant when creating and verifying XNYSS signatures.

```
OP_0 <sig> <1> <pk hash A> [pk hash B] [pk hash C...] <n>
    OP_CHECKXNYSSMULTISIG
```

Listing 5.1: XNYSS multisig redeem script with data.

XNYSS multisig scripts. To claim funds paid to an XNYSS-based address, we must create a valid XNYSS signature and a multisig redeem script (using the new opcode) that matches the script hash included in the address. Since our key state contains the *ots* flag, the wallet knows whether to include new public key hashes in the signature (for long-term addresses) or not. The used wallet program must update and store the key state accordingly after signing.

Verifying an XNYSS multisig script is done in the same way as verifying a normal Bitcoin multisig (and thus can reuse the same code), except that the signature is verified by first evaluating the *PkFromSig* algorithm from Section 4.3.2 and checking whether the hash of the resulting public key 1) equals one of the k public key hashes included in the multisig script, 2) is present in the UPKH database, or 3) is included in a signature of a previous input in the same transaction. If any of these cases apply, the signature is valid and script evaluation ends successfully. See Algorithm 4 for pseudo code. Note that the given code assumes an XNYSS multisig contains only one signature (i.e. is a 1-of-k multisig).

There is however an issue when using XNYSS signatures in bitcoin, which is the maximum size of data that can be pushed onto the stack during script verification: it is fixed at 520 bytes, which is far too small for an XNYSS signature. In our implementation we simply increased this value [1], but this is not necessary when using segregated witness (as described below).

Supporting XNYSS. In summary, to add support for XNYSS multisig scripts, a bitcoin wallet must be able to create multisig scripts with public key hashes and the new `OP_CHECKXNYSSMULTISIG` opcode and to manage key state. A client must be able to verify an XNYSS signature during multisig

[1] In bitcoin terms, this would result in a 'hard fork': when the acceptance rules are relaxed, blocks that satisfy the new rules may not satisfy the old ones, and would thus be rejected by clients that have not yet updated. When the rules are made more strict, all blocks accepted under the new rules are also valid under the old ones, resulting in a 'soft fork': as long as the new rules are accepted by a majority of users, the network will eventually converge to a single chain.

Algorithm 4 The XNYSS multisig verification algorithm

```
1: procedure b ←VerifyMultisig(M, σ, script_R, tx)
2:     pkhs ← GetPublicKeys(script_R)   ▷ Get pkhs from redeem script
3:     pkh_σ ← H(xnyss.PkFromSig(σ, M))
4:     for all pkh ∈ pkhs do
                                   ▷ Check if pkh_σ corresponds to a start node
5:         if pkh_σ = pkh then
6:             return true
7:         end if
                                ▷ Check if pkh_σ was signed in a previous node
8:         pkh_lt ← upkh.GetLongTerm(pkh_σ)
9:         if pkh_lt = pkh then
10:            return true
11:        end if
                                      ▷ Check if pkh_σ was signed in the same tx
12:        for all inp ∈ tx.ins do
13:            σ_inp ← GetSignature(inp.script_S)
14:            if pkh_σ ∈ σ_inp.pks then
15:                script_R^inp ← GetRedeemScript(inp.script_S)
16:                if script_R^inp = script_R then
17:                    return true
18:                end if
19:            end if
20:        end for
21:    end for
22:    return false
23: end procedure
```

script verification, which requires an implementation of the UPKH database and handling the new opcode `OP_CHECKXNYSSMULTISIG`.

5.2 UPKH Dabatase

The UPKH database is used to store public key hashes, advertised in signatures for long-term addresses, until their corresponding key pair is used for subsequent signatures. In our implementation each UPKH record $rec = (pkh_{un}, pkh_{lt}, bh)$ contains the unused public key hash pkh_{un}, the hash of the corresponding long-term public key pkh_{lt} (the start node of a chain), and the 'block height' bh at which it was added to the blockchain. All UPKH records are saved in a hashmap to provide fast lookups. Whenever a block is successfully verified, a new UPKH record is created for all public key hashes advertised in that block's XNYSS signatures (using the new block's height), and the new records are added to the UPKH database. Note that advertised public key hashes that are used in the same transaction to sign a different input must not be added to the database because they have already been used. When the public key hash of a signature in a new block is present in the database, it is removed when that block is accepted.

UPKH database size. In our current implementation each UPKH record is 56 bytes in size (32 bytes for the unused public key hash, 20 bytes for the long-term public key hash [2], and 4 bytes for the block height). The storage footprint could be reduced by grouping UPKHs for the same long-term public key hash: the UPKH database would then consist of a hash map with (part of) the long-term public key hashes as indices, and with hash maps that map an unused public key hash to a block height as values. This approach saves 20 bytes for every UPKH entry in a chain after the first one. Unfortunately we were not able to implement this optimization during the time allotted to this thesis.

Adding a database of unused public key hashes will of course add to the total storage required for all clients. However, its size is always a fraction of the full blockchain because it does not contain any information that is not present in the blockchain itself. We should also keep in mind that most users do not need long-term addresses at all, and *should* be using one-time addresses, which do not add any data to the UPKH database, to protect their privacy.

In fact, limiting the amount of available public key hashes per address would introduce several issues. Let us assume we use a limit of m UPKHs per address. Whenever a transaction is issued, we check whether the total amount of UPKHs after applying the transaction would exceed m; if so, we

[2] Bitcoin uses the RIMP160 hash function, which has 20-byte outputs, for hashes included in addresses and multisig scripts.

reject the transaction. One minor issue is that there will always be people that want more than m UPKHs; without becoming practically infinite, m will never be large enough for everyone.

Another more important issue is that (offline) wallets do not have any information on whether an unconfirmed transaction has been rejected or not. Thus, wallets will need to keep their own count of confirmed UPKHs. Let us assume we currently have m confirmed UPKHs in the database, and the wallet just signed a transaction consuming one of them and providing a new one (keeping the total at m). If we sign another transaction without confirming the previous one, the wallet must assume that either 1) the previous one is not yet confirmed, or that 2) it was rejected. Either way, it runs into problems if it chooses wrong.

In case 1), the wallet will wait for confirmation of the transaction, having $k-1$ available signing nodes until it is confirmed. However, if that confirmation is never given because the transaction is actually rejected, we effectively reduce our available nodes *from the wallet's perspective* by one permanently. Thus the *wallet* will eventually run out of available public keys, even though not all slots in the UPKH database are in use for the corresponding address.

In case 2), the wallet would decide to revoke the public key hash used for the previous transaction. Now, if the first transaction was actually still pending it will be canceled if the second is accepted first. We thus risk having to sign and distribute a perfectly valid transaction again. If the first is accepted before the second, the latter transaction becomes invalid since it is trying to revoke an already used UPKH. If we decide to accept such transactions anyway, an attacker could try to bloat the blockchain with invalid UPKH revocations. So again we have to re-sign and redistribute a transaction. In this situation we are essentially back to using non-forking chains, since we will have to wait for previous transactions to be confirmed before being able to sign a new one.

5.3 Updating Wallet State

When a long-term address is used to sign a transaction input, the created signature advertises public keys of key pairs to be used in the future. Before those key pairs keys advertised in the signature can be used to sign for the same address, the transaction must be included in a block and added to the blockchain. The wallet thus needs to be able to poll a client about the status of advertised public keys. We implemented this process as follows:

- On request of the user the wallet generates a file listing unconfirmed public key hashes;
- The user moves the generated file to a machine running a client;

- The client looks up the confirmation status on a user request, writing it to another file;
- The user moves this confirmation file back to the wallet;
- On request of the user the wallet updates its key state accordingly.

By using the block depth as confirmation status, a user can decide how many confirmations are required before an advertised key pair can be used to create a signature. This is similar to waiting for a transaction to be confirmed by a few more blocks before spending funds received in that transaction, to make sure that the transaction was not part of a branch that is later abandoned; if a user signs with a key pair that was advertised in a block that is later abandoned, that key pair goes to waste since it cannot be used again. By setting the amount of required confirmations to a higher value (while 3 is already 'highly reliable', 6 is generally regarded as safe [3]), this issue can be avoided.

5.4 Segregated Witness

The main drawback of our scheme is its large signature size: when signatures become larger, fewer transactions fit in a block and transaction fees increase. In bitcoin we can use the segregated witness protocol to reduce the ramifications of our large signatures. We can create a native Pay-To-Witness-Script-Hash (P2WSH) address [4] by creating a public key script as described in Section 5.1, computing the corresponding script hash, prefixing it with a version byte set to 0, and encoding the result to a new base-32 format called 'bech32' [5]. When claiming funds sent to such a segwit address, we create a normal XNYSS signature and recreate the multisig redeem script, but instead of putting these in the signature script we write them to the witness structure; the signature script is left empty. Verification of the witness script and signature is then performed as normal, except that the maximum script element size of 520 bytes can be ignored since the witness structure is not restricted by existing script constraints. When using XNYSS signatures in segwit scripts it is thus not necessary to change the maximum script element size.

Unfortunately we were not able to include segwit support for XNYSS keys in our implementation due to the limited time available for this thesis.

[3] See https://bitcoin.org/en/you-need-to-know#instant

[4] The segwit protocol specifies P2SH witness addresses, which are indistinguishable from normal P2SH addresses, and 'native' witness addresses that use a new address format. For the native versions, we can leave an input's signature script empty, which is not the case for the P2SH versions; thus, using native segwit addresses results in smaller transactions (see https://bitcoincore.org/en/segwit_wallet_dev/).

[5] See https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki#Bech32

	Traditional		Segwit	
w	one-time	long-term	one-time	long-term
16	2333	2429	658	682
256	1277	1373	394	418

Table 5.1: Total transaction size (one input and one output) in bytes when using 'normal' and (native) P2WSH scripts, with 1-of-3 multisig scripts.

We can however compute the new virtual transaction size, which as mentioned in Section 3.2 is computed as (base size $\cdot 3 +$ full size)/4: the results are listed in Table 5.1. Note that a P2WSH public key script is 34 bytes long instead of 24 because the redeem script is hashed with SHA256. The reduction in transaction size, about 70%, makes our scheme much more useful in practice.

Chapter 6

Discussion and Conclusion

6.1 Conclusion

We presented the post-quantum secure signature scheme XNYSS, which is based on Naor-Yung signature chaining of an OTS scheme, which we instantiate with W-OTS+, and is specifically designed for use in blockchain technologies. Our scheme supports both one-time and many-time public keys, and provides smaller signatures and better performance than existing many-time hash-based signature schemes such as XMSS. Moreover, we showed that XNYSS can easily be integrated into existing bitcoin implementations, and thus does not require the creation of another cryptographic currency. While XNYSS signatures are still significantly larger than those currently produced in bitcoin, we showed that signature size can be reduced at the cost of performance (by increasing the W-OTS+ Winternitz parameter w), and that the segwit protocol can be used to significantly reduce the resulting transaction size.

A disadvantage of our scheme is that it is stateful. Consequently, loss of key state can lead to loss of funds, and restoring a full system backup can result in reuse of one-time key pairs which significantly reduces the security of our scheme. It is however possible to create safe key state backups using our scheme, in a way that also allows multiple devices to create signatures for the same long-term public key.

6.2 Related Work

The QRL. The Quantum-Resistant Ledger (QRL) [36] uses XMSS instead of secp256k1 to provide a post-quantum secure blockchain. It was created as an alternative to bitcoin, the primary reason being that updating bitcoin to use a post-quantum secure signature scheme would be quite hard in practice (for the full reasoning, see https://theqrl.org/faq/coinsupdate/). The QRL launched its main network in June 2018.

As the QRL uses single-tree XMSS with Winternitz parameter $w = 16$ and (by default) a tree height of 10. These parameters result at most $2^{10} = 1024$ signatures of 2500 bytes. If a user runs out of signatures for a certain address, they must take care to empty their wallet with the last available signature; otherwise all remaining funds are lost. For the same value of w, XNYSS produces signatures that are about 13% smaller, has significantly better performance (as shown in Section 4.4), and supports a practically infinite amount of signatures for every address (see Section 4.3.3).

The large signature size of XMSS is a problem in the context of blockchain technology, as it means that less transactions fit in a block. When defining a new blockchain such as the QRL, the maximum block size could simply be increased, but that would require more bandwidth and storage space. However, perhaps the biggest issue with projects such as The QRL is that there are over 1600 different cryptographic currencies, yet bitcoin alone accounts for over 40% of the market share, followed by Ethereum with about 17% [1]. Rather than introducing yet another alternate coin, we think it is more effective to integrate post-quantum secure signature schemes in existing implementations. In this thesis we have shown that this is indeed possible in the case of bitcoin, and requires relatively little effort.

IOTA. Another project that provides a post-quantum secure distributed ledger is IOTA [17], which focuses on the application of blockchain technology for the Internet-of-Things by providing fee-less micro-transactions. IOTA actually replaces the blockchain with a new structure called the tangle, a discussion of which is out of scope for this thesis.

IOTA achieves post-quantum secure signatures by using the Lamport OTS scheme [30], which means that an address in IOTA can be used to create only one signature. As such, long-term addresses are not possible in IOTA. This may be sufficient for some applications, but as we have seen in the above there are situations where being able to create multiple signatures for the same address is more practical and can reduce the risk of losing funds due to failed transactions. IOTA does not have bitcoin's issue where a transaction with a low fee may take a very long time to be adopted into the blockchain, as IOTA is fee-less. However, if a user signs and distributes an invalid transaction, the corresponding funds are either at risk of being stolen if a second signature is created and distributed, or lost if not. When using XNYSS, after signing and distributing an invalid transaction the corresponding funds can still be claimed securely because of the available backup nodes.

Depending on the preferred security level, the length of a signature in IOTA is either 1300 bytes ('low' security level of 129 bits), 2600 bytes ('medium' security level of 257 bits), or 3900 bytes ('high' security level

[1] As of July 2nd, data retrieved from https://coinmarketcap.com/.

of 386 bits). At roughly the same 'medium' security level of 256 bits for W-OTS+, XNYSS provides smaller one-time signatures (roughly 16% using $w = 16$), and provides long-term addresses as well.

BPQS. While we were working on this thesis, Chalkias, Brown, Hearn, Lillehagen, Nitto and Schroeter published an idea similar to ours called Blockchained Post-Quantum Signatures (BPQS) [12]. BPQS, instead of creating one (or multiple) big tree(s), chains together very small Merkle trees consisting of only two leaves, a left and right leaf both containing an OTS public key. The right leaf can be used to create a signature while the left is used to sign the root of another small Merkle tree. Using this construction, the authors define BPQS-FEW, which consists of a fixed tree where all keys are precomputed, and BPQS-EXT, where only the keys for two leaves are computed and the left leaf can be used to sign another small Merkle tree when required. These two constructions can be mixed, for example by starting with BPQS-FEW but with the last left leaf signing a BPQS-EXT root, thus starting with a fixed number of available signatures but allowing an arbitrary number if required. It could even be used to fall back to a different signature scheme such as SPHINCS. And since all signatures are included in blocks in the blockchain, it is not necessary for a verifier to check the entire authentication path of a BPQS tree by referencing the previously signed block in a new signature.

Both BPQS and XNYSS are hash-based signature schemes designed to be used in blockchain technologies, and use a similar approach. The main difference between these two proposals is that XNYSS was designed to be easily added to existing blockchain implementations that are not yet quantum secure, while BPQS is a general improvement for blockchain technologies using hash-based signature schemes, such as the QRL. And while the authors briefly mention that verification with BPQS is efficient because part of a signature's authentication path is stored in the blockchain, we provide a practical implementation of this approach in the form of our UPKH database.

6.3 Improvements and Future Work

Practical considerations In Section 4.4 we provide benchmarks for our scheme using two values for the Winternitz parameter parameter, $w = 16$ and $w = 256$. By increasing w we decrease signature size but increase computation time, which is not a trivial trade-off. Decreasing signature size allows us to fit more transactions in a block, thus increasing the throughput of the network and decreasing required transaction fees. However, the resulting increase in computation time means that signature verification, and thus block verification, is slower. New clients that join the network have

to download and verify the entire blockchain before being able to participate [2]: an increase in signature verification time means that this initial download will take longer. More work should be done to determine whether this trade-off between transaction size and initial download time is worth it, for example by using a large-scale simulation.

We have also not covered the economic ramifications of our work. Transactions that include XNYSS signatures will require a higher fee because of their larger size. Assuming a fee of 6 satoshis per byte, a transaction with one input and one output using an XNYSS multisig script for a one-time address (using segwit and $w = 16$) requires a total fee of $6 \cdot 394 = 2364$ satoshis, as opposed to 1350 satoshis for a median transaction size of 225 bytes [3]. Whether such a fee is already practical, and what would be its effect on the current market, are interesting questions for future work.

Performance improvements. Currently our XNYSS code uses a SHA256 implementation provided by the standard Go `crypto` package. Since the XNYSS algorithms spend most of its time computing SHA256 hashes, any performance improvement in the SHA256 implementation used (for example by using AVX or NEON vector assembly instructions [14][43]) will result in a speedup of XNYSS. It might also be worthwhile to try different, more efficient hash functions such as BLAKE2 [3] to improve performance. And while Go is not slow by any means, it is not quite as fast as C. We expect an optimized C implementation of our scheme to outperform the one accompanying this thesis.

Unbalanced Merkle tree script. We use the bitcoin multisig script to base an address on multiple XNYSS chains. Each chain is represented in the multisig with a 20-byte public key hash. For one-time addresses, if all goes well only one of them will ever be used to create a signature. It is possible to reduce the total script size by using an unbalanced Merkle tree (depicted in Figure 6.1) instead of a multisig script, where the public key hashes of the XNYSS chains are all on different levels: the first public key hash is directly below the root node, the second is one level below that, etcetera. When creating a signature, we then include the authentication path required to verify this tree, which for the first key pair consists of the signature and one hash (the first non-leaf node in the unbalanced Merkle tree). However, when the signed transaction fails and we have to sign again using the same

[2] This is not entirely true as bitcoin also supports clients that only download block headers, without the full transacactions (see `https://bitcoin.org/en/developer-guide#simplified-payment-verification-spv`). However, such clients must trust the server they are downloading from to be truthful, and if SVP clients were to be massively adopted the bitcoin network would in practice be centralized around a few full nodes that do store the entire blockchain, a situation that bitcoin is designed to prevent.

[3] Data sourced from `https://bitcoinfees.earn.com/` on July 16, 2018.

address, the authentication path becomes longer since we have to use a node that is further down in the tree.

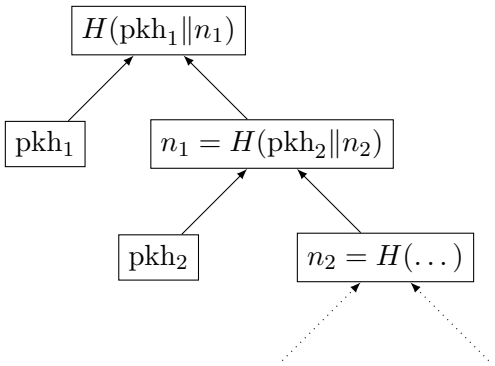

Figure 6.1: An illustration of an unbalanced Merkle tree, which could be used instead of multisig scripts to slightly reduce the total script size.

Using this approach, the first signature for a one-time address can thus be shorter than when using multisig scripts, but the script size increases with every failure. We did not implement this approach because the reduction in script size is quite small (a few 20-byte hashes) compared to the total signature size, and because it would require more changes to script evaluation than when using the existing multisig scripts. It is however an interesting optimization that might be useful in the future.

NIST PQC. Multiple signature schemes were submitted to the NIST Post-Quantum Cryptography Standardization project (https://csrc.nist.gov/Projects/Post-Quantum-Cryptography), which aims to standardize post-quantum secure alternatives to currently used cryptographic schemes. Signature schemes such as LUOV and Rainbow might be useful in blockchain applications because of their relatively small signature sizes, and since they do not require any key state to be kept they could be used as drop-in replacements for currently used schemes.

Bibliography

[1] Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 284–293. ACM, 1997. 10

[2] Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 34–51. Springer, 2013. 30

[3] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013. 54

[4] Stephane Beauregard. Circuit for shor's algorithm using 2n+ 3 qubits. *arXiv preprint quant-ph/0205095*, 2002. 4

[5] Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999. 37

[6] Daniel J Bernstein. Introduction to post-quantum cryptography. In *Post-quantum cryptography*, pages 1–14. Springer, 2009. 4

[7] Daniel J Bernstein. Simplified high-speed high-distance list decoding for alternant codes. In *International Workshop on Post-Quantum Cryptography*, pages 200–216. Springer, 2011. 10

[8] Daniel J Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 368–397. Springer, 2015. 10, 15, 16

[9] Bhaskar Biswas and Nicolas Sendrier. McEliece cryptosystem implementation: Theory and practice. In *International Workshop on Post-Quantum Cryptography*, pages 47–62. Springer, 2008. 10

[10] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the security of the winternitz one-time signature scheme. 2011. https://eprint.iacr.org/2011/191. 14

[11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS-a practical forward secure signature scheme based on minimal security assumptions. In *International Workshop on Post-Quantum Cryptography*, pages 117–129. Springer, 2011. 10, 15

[12] Konstantinos Chalkias, James Brown, Mike Hearn, Tommy Lillehagen, Igor Nitto, and Thomas Schroeter. Blockchained Post-Quantum Signatures. Cryptology ePrint Archive, Report 2018/658, 2018. https://eprint.iacr.org/2018/658. 53

[13] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital signatures out of second-preimage resistant hash functions. In *International Workshop on Post-Quantum Cryptography*, pages 109–123. Springer, 2008. 14

[14] Ana Karina DS de Oliveira and Julio López. An efficient software implementation of the hash-based signature scheme mss and its variants. In *International Conference on Cryptology and Information Security in Latin America*, pages 366–383. Springer, 2015. 54

[15] Jintai Ding and Bo-Yin Yang. Multivariate public key cryptography. In *Post-quantum cryptography*, pages 193–241. Springer, 2009. 10

[16] Chris Dods, Nigel P Smart, and Martijn Stam. Hash based digital signature schemes. In *IMA International Conference on Cryptography and Coding*, pages 96–115. Springer, 2005. 14

[17] IOTA Foundation. The Next Generation of Distributed Ledger Technology — IOTA. https://www.iota.org/. Accessed: July 12 2018. 52

[18] LC Coronado Garcıa. On the security and the efficiency of the Merkle signature scheme. Technical Report 192, Cryptology ePrint Archive, 2005. Available at http://eprint. iacr. org/2005/192, 2005. 14

[19] Omkar Godbole. Bitcoin Price Primed to Test $20k Ahead of CME Launch. https://www.coindesk.com/sell-news-bitcoin-price-tests-20k-ahead-cmes-futures-launch/. Accessed: 16 March 2018. 4

[20] Oded Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 104–110. Springer, 1986. 15

[21] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988. 9

[22] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996. 10

[23] A. Hülsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, RFC Editor, May 2018. 16

[24] Andreas Hülsing. W-OTS+–shorter signatures for hash-based signature schemes. In *International Conference on Cryptology in Africa*, pages 173–188. Springer, 2013. 10

[25] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In *Public-Key Cryptography–PKC 2016*, pages 387–416. Springer, 2016. 10

[26] Intel. 2018 CES: Intel Advances Quantum and Neuromorphic Computing Research. https://newsroom.intel.com/news/intel-advances-quantum-neuromorphic-computing-research/. Accessed: 15 March 2018. 4

[27] Julian Kelly. A Preview of Bristlecone, Google's New Quantum Processor. https://research.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html. Accessed: 15 March 2018. 4

[28] Kazukuni Kobara and Hideki Imai. Semantically secure McEliece public-key cryptosystems-conversions for McEliece PKC. In *International Workshop on Public Key Cryptography*, pages 19–35. Springer, 2001. 10

[29] Philip Lafrance and Alfred Menezes. On the security of the wots-prf signature scheme. Cryptology ePrint Archive, Report 2017/938, 2017. https://eprint.iacr.org/2017/938. 15

[30] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report CSL-98, SRI International Palo Alto, 1979. 11, 52

[31] R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, January 1978. 10

[32] Ralph C. Merkle. Method of providing digital signatures. Patent, 01 1982. US4309569A. 11

[33] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989. 11, 13

[34] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008. Accessed: 19 June 2018. 20, 21

[35] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 33–43. ACM, 1989. 31

[36] The QRL. QRL - The Quantum Resistant Ledger. https://theqrl.org/. Accessed: July 12 2018. 4, 51

[37] Minghua Qu. Sec 2: Recommended elliptic curve domain parameters. *Certicom Res., Mississauga, ON, Canada, Tech. Rep. SEC2-Ver-0.6*, 1999. 28

[38] Oded Regev. New lattice-based cryptographic constructions. *Journal of the ACM (JACM)*, 51(6):899–942, 2004. 10

[39] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, pages 371–388. Springer, 2004. 7

[40] Yu Sasaki and Kazumaro Aoki. Finding Preimages in Full MD5 Faster Than Exhaustive Search. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 134–152, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 7

[41] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. Ieee, 1994. 4, 10

[42] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *International*

Conference on the Theory and Application of Cryptology and Information Security, pages 617–635. Springer, 2009. 10

[43] Wouter van der Linde, Peter Schwabe, and Lejla Batina. Parallel sha-256 in neon for use in hash-based signatures. BSc thesis, Radboud University Nijmegen, 2016. 54

[44] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Annual international cryptology conference*, pages 17–36. Springer, 2005. 14

[45] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer, 2005. 14

Appendix A

Using our code

Our code is available at https://github.com/lentus/wotscoin. Assuming that Go is installed correctly (see https://golang.org/doc/install for installation instructions), the code can be retrieved with the command `go get github.com/lentus/wotscoin`. To build our software, navigate to `$HOME/go/src/github.com/lentus/wotscoin`. Then execute `go build` in the `wallet` folder to build the wallet, and in the `client` folder to build the client. See the included `README` for further usage information.

Currently the client is setup to connect to a private network we used for testing which has been taken offline. To create a private network, we recommend to setup a permanent client on a server (using a random port number for the test network, or 52082 which is currently used in the code), then setup a bitcoin seeder (such as https://github.com/sipa/bitcoin-seeder, changing lines 401-405 to use the url of the server running the client and replace any occurrence of the port number 18333 with the chosen port number). Finally, in our code navigate to `lib/others/peersdb` and change lines 294-297 of `peerdb.go` to use the url of the seeder you installed and the chosen port number. After rebuilding and executing the client, it should connect to the permanent client. We used Pooler's CPU miner (https://github.com/pooler/cpuminer) to mine new blocks.