AMOS YI CONG LU ZHENG

**z-NAF and clipping: two improvements for
post-quantum hash-based digital signatures**

São Paulo
2019

AMOS YI CONG LU ZHENG

**z-NAF and clipping: two improvements for
post-quantum hash-based digital signatures**

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do Título de Mestre em
Ciências.

São Paulo
2019

AMOS YI CONG LU ZHENG

**z-NAF and clipping: two improvements for
post-quantum hash-based digital signatures**

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do Título de Mestre em
Ciências.

Área de Concentração:
Engenharia de Computação

Orientador:
Marcos Simplicio Antonio Jr.

São Paulo
2019

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catálogo-na-publicação

Zheng, Amos

z-NAF and Clipping: Two improvements for Post-Quantum Hash-based Digital Signatures / A. Zheng -- versão corr. -- São Paulo, 2019.
101 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.CRIPTOGRAFIA 2.SEGURANÇA DE COMPUTADORES
3.ALGORITMOS 4.Criptografia pós-quântica 5.Assinaturas digitais baseadas em hash I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

ACKNOWLEDGMENTS

First, I would like to thank God, the giver of life, for sustaining, caring and protecting me, and for giving me wisdom and insights until this very day. I am where I am thanks to his unfailing love and faithfulness.

I would also like to thank my parents for all support they have given me, as they always believed in me and were proud of my work. In this regard, I want to give special thanks to my mother who, besides helping me with many household tasks, always remembered me in her prayers and encouraged me to move forward even in the darkest nights.

Thanks to my brother, who, being more than a brother, was a close friend and shared with me many of my burdens. He encouraged me and helped me to not give up when I was stuck in my research.

Besides, I want to thank my advisor, prof. Marcos A. Simplicio Jr., for giving me research ideas and reviewing my work. He also introduced me to the academic world and taught me how to navigate and be successful in it.

Last but not least, I want to also thank my friends for their company in the often lonely and dull course of graduate school, they really made my daily life much more enjoyable. Also, special thanks to my lab mates who were with me in the same journey; they advised me and helped me with all the paperwork within the university.

This work was supported by Intel Corporation and FAPESP (grants 2015/50520-6 and 13/25977-7), and in part by CNPq (research productivity grant 301198/2017-9).

RESUMO

ZHENG, Amos Yi Cong Lu. **z-NAF and clipping: two improvements for post-quantum hash-based digital signatures**. 2019. 101p. Dissertação (Mestrado em Ciências) – Escola Politécnica, Universidade de São Paulo, São Paulo, 2019.

Esquemas de assinaturas baseadas em hash são uma classe de algoritmos pós-quânticos que basicamente consistem em árvores de hash construídas em cima de soluções de assinaturas unitárias (OTS). Tais esquemas possuem tamanhos pequenos de chaves, processamento eficiente e são simples de se implementar, enquanto que a segurança desses esquemas baseia-se na resistência à pré-imagem ou à colisão das funções de hash utilizadas. Apesar dessas vantagens, eles possuem tamanhos de assinaturas relativamente grandes comparados aos algoritmos tradicionais de assinatura. Nesse caso, uma forma de lidar com essa questão é reduzir os tamanhos de assinatura das OTS utilizadas. Além disso, em aplicações em que se faz muito mais verificações de assinatura do que gerações de assinatura, é desejável que a verificação seja significativamente mais rápida do que a geração. Nesse trabalho, duas contribuições são apresentadas para mitigar os problemas acima mencionados. A primeira é uma técnica probabilística que permite a redução do tamanho das assinaturas nas OTS utilizadas com custo adicional de processamento desprezível, isto é, pode-se alcançar uma redução média de até 12.5% dependendo do valor de w escolhido para a assinatura (no entanto, para $w = 4$ a redução é de apenas 0.2%). A segunda contribuição é um esquema inovador de assinatura digital que possui todas as vantagens do W-OTS e do W-OTS+, além de possuir verificação bem mais rápida do que estes em troca de uma geração de assinatura um pouco mais lenta.

Palavras-Chave: Criptografia. Segurança de computadores. Algoritmos. Criptografia pós-quântica. Assinaturas baseadas em hash. Assinaturas unitárias.

ABSTRACT

ZHENG, Amos Yi Cong Lu. **z-NAF and clipping: two improvements for post-quantum hash-based digital signatures**. 2019. 101p. Dissertação (Mestrado em Ciências) – Escola Politécnica, Universidade de São Paulo, São Paulo, 2019.

Hash-based signature schemes are a class of post-quantum algorithms that usually consist of hash-trees built upon OTS solutions. These schemes have small key sizes, efficient processing and are simple to implement, while their security properties rely basically on the pre-image or collision resistance of the their underlying hash function. Despite such advantages, however, they have relatively large signature sizes compared to traditional signature algorithms. One way of tackling this issue is to reduce the sizes of their underlying OTS algorithms. Besides that, in applications where signature verifications are done much more frequently than signature generation (e.g. trusted software distribution), it is desirable that signature verification has less overhead than signature generation. In this work, two contributions are presented to tackle these issues. The first one is a probabilistic technique that, with negligible processing overhead, allows reductions in the underlying OTS signature sizes; namely, up to 12.5% average size reduction can be achieved depending on the w parameter chosen for the signature (however, for $w = 4$ the reduction is only 0.2%). The second contribution is a novel OTS scheme which has all advantages of W-OTS and W-OTS+ and yet has much faster signature verification times at the cost of slightly slower signature generation times.

Keywords: Cryptography. Computer security. Algorithms. Post-quantum cryptography. Hash-based signature schemes. One-time signature schemes.

CONTENTS

1	Introduction	13
1.1	Goals	15
1.2	Contributions	15
1.3	Outline	16
2	Background	17
2.1	Notation	17
2.2	Mathematical Concepts	17
2.2.1	One-way functions	18
2.2.2	z-non-adjacent form (z-NAF)	18
2.3	Hash functions	19
2.4	Public-key Cryptography	20
2.4.1	Public-key encryption	20
2.4.2	Digital signature schemes	22
2.5	Hash-based signature schemes	23
2.6	The Winternitz One-time Signature Scheme (W-OTS)	26
2.7	The Merkle Tree and Hash-based Many-times Signature Schemes	29
2.8	Optimizations for hash-based signature schemes	32
3	The clipping technique for W-OTS scheme and its variants	35
3.1	Set up	36
3.2	Key Generation	36
3.3	Signature Generation	36
3.4	Signature Verification	37

3.5	Correctness of the algorithm	38
3.6	Example	39
4	Analysis of the clipping technique	41
4.1	Performance Analysis	41
4.1.1	Signature Size	41
4.1.2	Processing Time	43
4.1.3	Storage Requirements	43
4.2	Security Considerations	44
4.3	Proof of Concept	44
5	Using hash chains with z-NAF: the z-NAF OTS scheme	47
5.1	Setup	48
5.2	Key Generation	49
5.3	Signature Generation	49
5.4	Signature Verification	53
5.5	Correctness of the algorithm	54
5.6	Example	55
5.6.1	Setup	55
5.6.2	Key Generation	55
5.6.3	Signature Generation	56
5.6.4	Signature verification	57
6	Analysis of the z-NAF OTS scheme	59
6.1	Security Analysis	59
6.2	Performance Analysis	62
6.2.1	Signature Size	63
6.2.2	Processing Time	65

6.2.2.1	Key generation	65
6.2.2.2	Signature generation	65
6.2.2.3	Signature verification	66
6.2.3	Signature size - processing time trade-off	67
6.2.4	Storage Requirements	69
6.3	Proof of Concept	70
6.3.1	Searching for appropriate values for L_1 and t_{max}	72
6.3.2	Simulations	77
6.3.3	Performance comparison against W-OTS	82
6.3.3.1	z-NAF OTS vs. W-OTS: initial parameters	84
6.3.3.2	z-NAF OTS vs. W-OTS: optimized parameters	85
7	Conclusion	87
7.1	Publications	88
7.2	Future Work	88
	References	89
	Appendix A – Additional examples of the z-NAF OTS scheme	93
A.1	Example with $z = 2$	93
A.1.1	Setup	93
A.1.2	Key Generation	93
A.1.3	Signature Generation	94
A.1.4	Signature verification	95
A.2	Example with $z = 4$	95
A.2.1	Setup	95
A.2.2	Key Generation	95
A.2.3	Signature Generation	96

A.2.4	Signature verification	97
-------	----------------------------------	----

1 INTRODUCTION

The area of post-quantum cryptography refers to algorithms that can be executed on classic computers and, at the same time, resist known attacks made by quantum computers (BERNSTEIN; BUCHMANN; DAHMEN, 2008). This includes five main families of post-quantum algorithms and protocols: lattices (GOLDREICH; GOLDWASSER; HALEVI, 1997), error correcting codes (MCELIECE, 1978), multivariate quadratic systems (DING; SCHMIDT, 2005), isogenies on supersingular elliptic curves (JAO; FEO, 2011), and schemes based on symmetric cryptography primitives in general, such as hash functions (DODS; SMART; STAM, 2005).

The goal of researching such cryptosystems is to face the threat posed by quantum computing, an area that is evolving increasingly fast (ANTHONY, 2017). More precisely, if large-enough quantum computers become available, the security of many classical cryptographic solutions (e.g., RSA (RIVEST; SHAMIR; ADELMAN, 1977) and ECC (KOBLOITZ, 1987; MILLER, 1986)) can be bypassed by attackers using Shor (SHOR, 1999) or Grover (GROVER, 1996) algorithms. In a smaller scale, symmetric primitives are also affected by the Grover algorithm: for example, the security level of symmetric ciphers drop by half, so brute-force attacks against k -bit keys take $2^{k/2}$ operations instead of 2^k ; similarly, the security of k -bit hash functions against pre-image attacks drop from k to $k/2$, and against collisions the fall is from $k/2$ to $k/3$ (BERNSTEIN; LANGE, 2017). Therefore, efficient schemes that can become drop-in replacements for conventional solutions are required. For symmetric primitives, there are already natural alternatives, such as replacing AES-128 with AES-256 (NIST, 2001), or SHA-3-256 with SHA-3-512 (NIST, 2015a). For asymmetric schemes, however, this is a challenging issue, in particular because the memory, bandwidth and/or processing requirements of asymmetric post-quantum cryptosystems are usually higher than what is observed in currently standardized solutions (BERNSTEIN; BUCHMANN; DAHMEN, 2008).

Among the aforementioned families of post-quantum algorithms, a particularly promising area are hash-based signatures (BERNSTEIN; BUCHMANN; DAHMEN, 2008). Al-

gorithms in this category are built using one-way hash functions as underlying cryptographic primitive, so their (quantum and classical) security relies basically on well-known properties of those primitives, such as collision and pre-image resistance (DODS; SMART; STAM, 2005). That is because these algorithms all share the same idea of generating a list of random bit strings as the private key, then hash those strings one or more times to get the public key. After that, depending on the message to be signed, the corresponding pre-images are published with the signature. In addition, hash-based signatures are also quite efficient: besides leading to small public key sizes, their computational costs depend basically on performance of the underlying hashing functions, which are known to be processing and memory-efficient algorithms.

Despite this interest, hash-based signatures still face some challenges. One of its main limitations, common in many schemes in the literature, is the reasonably large signature sizes when compared to classical digital signature schemes. For example, W-OTS (MERKLE, 1990), a one-time signature scheme proposed by Ralph Merkle in 1979, had signature sizes around 4 KiB, which is 8x the size of ECC-based multi-signature schemes such as ECDSA (JOHNSON; MENEZES; VANSTONE, 2001). A more recent one-time signature proposal, W-OTS+ (HÜLSING, 2013), reduces this size in approximately half by requiring simply pre-image resistant hash functions rather than collision-resistant ones, but even that is still large compared to ECDSA. On the other hand, many-times hash-based signature proposals built upon one-time schemes, such as XMSS (BUCHMANN; DAHMEN; HÜLSING, 2011) and SPHINCS (BERNSTEIN et al., 2015), display signature sizes of 10 KiB and 41 KiB respectively, which remains much larger than even not too compact classical schemes, such as RSA, for an equivalent security level against non-quantum computers.

Besides that, many applications require many more signature verifications than signature generations. This is the case of e.g. software stores and trusted boot images, where the software or image is signed once and then, every time it is used and for every user that uses it, a signature verification is done to make sure the content wasn't tampered with. Because of this, it is desired that the verification overhead be small, whereas a larger signing overhead could be more easily tolerated. In these scenarios, a signature scheme with fast verification is ideal, even if its signature generations are comparatively slower.

1.1 Goals

Given the limitations of hash-based signature schemes, the goal of this work is to research ways to improve their signature sizes and reduce the signature verification overhead, thus promoting the adoption of hash-based schemes as a replacement for conventional digital signature schemes that are quantum-vulnerable.

1.2 Contributions

This work proposes two contributions to current hash-based signature schemes.

The first is a probabilistic technique that can be combined with one-time schemes such as W-OTS and W-OTS+ for reducing the signature size under certain conditions. The proposed approach, called *clipping*, relies on pseudo-random function (PRF) chaining, so the scheme’s private keys are generated in two layers instead of a single one. As a result, two or more hash pre-images that need to be revealed for composing the signature do not need to be explicitly send: since they can be derived from a single seed, it suffices to reveal this seed. Even though this is a probabilistic approach whose efficiency depends on the data to be signed, its overhead in terms of processing cost or memory usage is negligible, so the (potentially small) gains obtained when it is employed come basically for free. To the best of our knowledge, nothing has been done in the literature for hash-based signature schemes that resembles this first contribution.

The second is a novel hash-based one-time signature scheme, called *z-NAF OTS*, which has all advantages of W-OTS, namely, that a trade-off between signature size and processing time is possible and the public key can be derived from the signature, but has much faster verification times than W-OTS for similar signature sizes at the cost of slightly longer signature generation times. This scheme leverages the use of randomized hashing with the generalized non-adjacent form encoding to achieve verification speed-up with only a small relative slow down in signature generation. When compared to related work, this second contribution shares some goals with the randomized hashing technique proposed in (STEINWANDT; VILLÁNYI, 2008), since it also focus on faster verification at the cost of slower signature generation. However, that work does not use non-adjacent form encoding and a trade-off between signature size and processing time is not possible. More recently, Roh et al. proposed using non-adjacent form with the W-OTS+ scheme to speed up signature generation (ROH; JUNG; KWON, 2018), but they use it in a different way and use only 2-NAF instead of the generalized version z-NAF. Besides, their

scheme have smaller reductions in processing time (around 8% in signature generation algorithm) compared to ours (more than 50% in signature verification time, depending on the parameters chosen).

1.3 Outline

The outline of this work is as follows: Chapter 1 is a quick introduction about hash-based signature schemes where the motivation, goals and contributions, as well as the related works are presented.

Chapter 2 contains all background concepts needed to understand hash-based signatures such as mathematical concepts, hash functions, public-key cryptography and digital signature schemes, as well as the most fundamental hash-based constructions.

Chapter 3 presents the first contribution, a technique that can be applied to a wide range of hash-based signature schemes to reduce signature size, while Chapter 4 analyzes the presented technique.

Chapter 5 presents the second contribution which is a novel one-time signature scheme that has better verification times while retaining all advantages of the widely used one-time signature schemes W-OTS and W-OTS+, and Chapter 6 analyzes the z-NAF OTS scheme.

Finally, Chapter 7 concludes with final remarks as well as all publications resulting from this work and possible future work, then there is Appendix A with some additional numerical examples for the z-NAF OTS scheme.

2 BACKGROUND

In this chapter some background concepts related to hash-based signatures are presented. These concepts are necessary to understand how hash-based signature schemes work and how improvements could be made on them. They are divided in the following sections: notation, basic mathematical concepts, hash functions, public-key cryptography, hash-based signature schemes, the Winternitz one-time signature scheme, and the Merkle tree construction which enables many-times signature schemes.

2.1 Notation

The following notation is used throughout this work. Given a non-negative real number x and two bit strings A and B , both n bits long,

- $|A|$ denotes the length of bit string A in bits (in this case $|A| = n$);
- $A \oplus B$ denotes the bitwise exclusive OR (XOR) operation between A and B ;
- $A||B$ denotes the concatenation operation of A and B , producing a single bit string $2n$ bits long;
- $\lceil x \rceil$ denotes the ceil operation, that is, returns the smallest integer that is equal to or larger than x .

Besides, $(\cdot)_{10}$ denotes an integer in decimal representation, $(\cdot)_2$ denotes an integer in binary representation, and $(\cdot)_{z-NAF}$ denotes an integer in z-NAF representation (z-NAF representation is introduced in Subsection 2.2.2). In z-NAF representation, a digit with a bar denotes a negative digit, for example, $\bar{2}$ denotes the digit -2 .

2.2 Mathematical Concepts

This section describes the main mathematical concepts used in hash-based signatures.

2.2.1 One-way functions

One important concept in cryptography is the *one-way function*. A one-way function f is a function that is easy to compute but hard to invert. Here, "easy to compute" means that, given any element x in the domain of f , an algorithm is known which can compute $y = f(x)$ in polynomial time. Similarly, "hard to invert" means that, given a random element y in the image of f , no algorithm is known that can compute a pre-image x such that $f(x) = y$ in polynomial time (MENEZES; VANSTONE; OORSCHOT, 1996). Therefore, "easy" and "hard" are defined by the availability of a polynomial-time algorithm that performs the specified operation.

One-way functions are the building blocks of public-key cryptography in general. Most public-key cryptographic schemes (such as RSA and ECDSA) use a special kind of one-way function called *trapdoor functions*, which are one-way functions with an additional property that, given another special piece of information (the trapdoor information), inverting f becomes "easy" in the sense described above. Section 2.4 describes in more detail the use of one-way functions in public-key cryptography. For more information about one-way functions and trapdoor functions, please refer to (MENEZES; VANSTONE; OORSCHOT, 1996).

2.2.2 z-non-adjacent form (z-NAF)

Another important concept present in the z-NAF OTS scheme (described in Chapter 5) is the z-non-adjacent form, abbreviated as z-NAF.¹ While most integer representations use unsigned digit representations (e.g. the decimal integer $(3971)_{10}$ or the binary integer $(100110001)_2$), z-NAF is a signed digit representation system, meaning that the representation system can have both positive and negative digits.

The definition of the z-NAF representation is provided below.

Given an integer n and parameter $z \geq 2$, the z-NAF representation of n , written $n = \sum_{j=0}^l n_j 2^j$, is a signed integer representation that satisfies the following properties (AVANZI, 2004):

- Either $n_j = 0$ or n_j is odd and $|n_j| < 2^{z-1}$;
- If $n_j \neq 0$, then $n_{j+1} = \dots = n_{j+z-1} = 0$.

¹In the literature, this encoding is called w-NAF, which stands for *w-width non-adjacent form*. In this dissertation as well as in the derived OTS scheme, the "w" in w-NAF was exchanged for "z" in order to avoid confusion with the Winternitz parameter w .

For example, the integer $(50)_{10}$, which in binary form is $(110010)_2$, has the 2-NAF representation $(10\bar{1}0010)_{2-NAF}$, 3-NAF representation $(30010)_{3-NAF}$, and 4-NAF representation $(1000\bar{7}0)_{4-NAF}$ (where a digit with a bar $\bar{d}$ represents the negative digit $-d$), since $1 \times 2^6 - 1 \times 2^4 + 1 \times 2^1 = 3 \times 2^4 + 1 \times 2^1 = 1 \times 2^6 - 7 \times 2^1 = 50$.

The z-NAF representation has been independently introduced by Cohen et al. (MIYAJI; ONO; COHEN, 1997) and Solinas (SOLINAS, 1997). Then, it was explored by many authors, mainly to speed up exponentiations and elliptic curve point multiplications.

The z-NAF representation has some very useful properties that are leveraged in the z-NAF OTS scheme, namely:

- Every integer has a unique representation in z-NAF form (AVANZI, 2004; HEUBERGER; KRENN, 2013; MUIR; STINSON, 2006);
- Given a l -bit integer in ordinary binary form, the z-NAF form of that integer has no more than $l + 1$ digits in z-NAF form (MUIR; STINSON, 2006);
- An average z-NAF representation has a non-zero digit density of about $1/(z + 1)$ (MUIR; STINSON, 2006; COHEN, 2001).

Avanzi (AVANZI, 2004) provides a linear time algorithm to convert any integer in ordinary binary representation to z-NAF representation. Therefore, the conversion to z-NAF representation is a fast operation.

2.3 Hash functions

A *hash function* H is a polynomial-time function which maps inputs of arbitrary size to outputs of fixed size (MENEZES; VANSTONE; OORSCHOT, 1996). Besides being one of the most important primitives of modern cryptography, hash functions are also widely used in data structures, image and audio processing.

A *cryptographic hash function*, which is the kind of hash function used in cryptography, possess these additional properties:

- Pre-image resistance: given an image y of H , it is computationally infeasible to find a pre-image x such that $H(x) = y$;
- Second pre-image resistance: given the pair (x, y) such that $H(x) = y$, it is computationally infeasible to find a second pre-image x' such that $H(x') = H(x) = y$;

- Collision resistance: it is computationally infeasible to find x and x' such that $H(x) = H(x')$.

As shown in the following sections, these properties are fundamental for the security of most public-key cryptographic schemes, including the hash-based signature schemes.

Examples of popular modern hash functions include SHA-2(NIST, 2015b), SHA-3 (NIST, 2015a) and BLAKE (SAARINEN; AUMASSON, 2015).

2.4 Public-key Cryptography

Public-key cryptography, also called asymmetric cryptography, is a field of cryptography that, in opposition to symmetric cryptography, uses different keys for encryption and decryption. One of the keys is called *public key*, which is to be publicly shared, while the other is called *private key*, which should be kept secret. Even though both keys are mathematically related, it should be computationally infeasible to derive the private key from the public key.

There are two important areas in public-key cryptography: public-key encryption and digital signature schemes. The first one uses the public key to encrypt messages such that only the owner of the private key can decrypt them. The second one uses the private key to generate a mathematical signature from the message, such that anyone can verify the authenticity of the message using the public key.

Some algorithms, such as RSA and DSA, are used both for public-key encryption and digital signature schemes, while others, such as hash-based schemes, are only used for digital signature schemes.

2.4.1 Public-key encryption

The main purpose of public-key encryption is to send private messages to the owner of the private key without knowing the private key itself. Here, the public key is used for encryption and the private key for decryption.

Suppose Alice wants to send a private message to Bob. The whole process works as follows:

- a) Bob generates a public-private key pair (if he doesn't already have one) and shares the public key with Alice (and anyone else he wish to receive messages from) while

keeping the private key for himself;

- b) Alice encrypts the message using Bob's public key;
- c) Alice sends the encrypted message to Bob. Since the public key can only be used to encrypt (and not decrypt) messages, and the private key cannot be derived from the public key, no one else can read Alice's message besides Bob;
- d) Bob uses his private key to decrypt Alice's message.

Thus, Alice can send a private message to Bob even without his private key. More generally, anyone could send a message to Bob with Bob's public key and be sure that only Bob (the owner of the associated private key) can decrypt those messages. Hence, public key encryption schemes possess the property of *recipient authentication*.

When compared to symmetric encryption schemes, public key encryption has as advantage the fact that it does not require exchanging a different symmetric key with each different sender a user wishes to receive messages from. For example, suppose a user U wants to receive a message from n different users. Instead of exchanging a different symmetric key with each of them and keeping track of n different keys, U could just send a single public key to all n users. As a result, every one of them can start sending messages to U without worrying about others being able to read the messages they send.

Another advantage of public key encryption schemes is that they can be used for digital signatures. This enables the construction of public key infrastructure (PKI), in which trust is delegated to a third party, the certificate authority (CA). In this way, if two users trust the same CA, they can start sending messages to each other securely even if they never met before, and without the need to publish their private keys to the CA (and thus avoiding the risk of being impersonated by CAs). For this reason, PKIs are widely used in the Internet today, and are among the most important building blocks of essential services such as Internet banking, e-commerce, e-mail, social networks, and so forth.

Despite of their advantages, public key encryption schemes are generally much slower than symmetric key schemes. Because of this, public key encryption is usually employed to exchange a symmetric key (e.g., a the session key) securely, and after that the two parties start communicating using the shared symmetric key instead of public keys.

2.4.2 Digital signature schemes

Contrary to public-key encryption, the purpose of a digital signature scheme is not to hide information, but instead make information verifiable. In this application, the role of public key and private key are usually the opposite of their roles in public-key encryption: the sender of the message is the one possessing the private key and uses it to sign the message, while the recipients of the message are those using the public key to verify the message.

As such, any digital signature scheme should possess the following properties:

- *Authenticity of the Sender*: the sender of the message can be authenticated;
- *Integrity*: the recipients can verify that the message has not be tampered with while in transit;
- *Non-repudiation*: if someone signed a message with his private key, he/she cannot deny later on that he signed that message (that is, if the message was signed before that key was revoked).

Those are the properties that traditional hand-made signatures also aim to provide, yet digital signatures are a much more secure replacement with security based on strong mathematical assumptions instead of human ingenuity.

Digital signature schemes typically has algorithms for the following steps: key generation (KEYGEN), signature generation (SIGGEN) and signature verification (VER). Each step is further detailed below:

- **KEYGEN**: The signer of the message generates a public-private key pair (just like in public-key encryption), and keeps the private key to himself while sharing the public key;
- **SIGGEN**: The signer of the message generates a signature for a message of his choice using the private key (through a specific mathematical algorithm) and send out both the message and the signature;
- **VER**: The recipients of the message verify that the signature is valid for that message by using another appropriate mathematical algorithm. If the signature is valid the message is accepted; otherwise it is rejected.

The set of algorithms for KEYGEN, SIGGEN and VER constitutes a public-key signature algorithm.

One important generic attack against any kind of digital signature scheme is signature forgery, where the attacker can generate valid signatures for messages that the legit signer did not sign. Three categories of signature forgery attacks can be identified and are presented from the least severe to the most severe:

- a) *Existential forgery*: The attacker can generate a valid signature for at least one message, but he has no control over the content of message he can generate valid signatures for;
- b) *Selective forgery*: In this kind of forgery, the attacker can select messages he wants to forge a signature for. This implies the capability of existential forgery;
- c) *Universal forgery*: The most severe kind of forgery. The attacker can generate a valid signature for any message in the message space; implies selective forgery capability.

Ideally, any digital signature scheme should be secure against existential forgery, at least for the group of messages defined as valid (that is, messages that satisfies a certain predefined structure restriction).

2.5 Hash-based signature schemes

Hash-based signature schemes are a class of digital signature schemes that relies only on hash functions as its cryptographic primitive. By not relying on unproven hardness assumptions of number-theoretic problems (such as integer factorization problem or the discrete logarithm problem), they can remain secure even if those problems are later solved computationally. In fact, Peter Shor has shown how a large enough quantum computer could be used to break the integer factorization and discrete logarithm problems (SHOR, 1999). Therefore, hash-based signatures remain promising candidates for post-quantum signature schemes (that is, schemes that could resist a quantum computer attack).

The basic idea behind hash-based signature schemes is to commit values by hashing them and publishing the outputs of the hash function as the public key, then reveal some of the committed values later on during signature generation. Since no one could invert the hash function, if someone first comes up with a value matching an output of the hash function previously published, that value could only be a committed value and it could

only come from the authentic signer. For example, to sign one bit, the signer could do the following:

- a) Choose two long enough integers at random, say, x and y of 256 bits;
- b) Hash both integers, and publish the resulting integers $H(x)$ and $H(y)$, both of 256 bits, as the public key;
- c) If the signer wants to sign the message 0, he publishes x as the signature; if he wants to sign the message 1, he publishes y as the signature.

Since no one could have calculated x or y (or both) from $H(x)$ and $H(y)$, if the verifier receives the message 0 with x or the message 1 with y , he knows that the signature is valid and the message is authentic.

As an important observation, the signer can only use the aforementioned signature scheme once (that is, it can only be used to sign either 0 or 1, but not both), otherwise both x and y would be publicly known and the messages would no longer be authentic (anyone could claim that the signer signed 0 or 1 at will). On the other hand, if this signature scheme is used only once, then no one can forge signatures for any message not signed by the signer. For example, suppose the signer signed only the message 0 and thus only published x . Since no one else knows y (and calculating y from $H(y)$ is computationally infeasible), no one could claim the signer signed the message 1. In the same way, if the signer only signed the message 1, no one could claim that he signed the message 0.

Being able to sign a single bit is not very useful, therefore if one wishes to sign n bits, he could choose n pairs of random values $(x_1, y_1, x_2, y_2, \dots, x_n, y_n)$ and publish the hashed values $(H(x_1), H(y_1), H(x_2), H(y_2), \dots, H(x_n), H(y_n))$ as the public key. Then, he could sign a n -bit message by publishing x_i if the i -th bit is zero and y_i otherwise. For example, to sign the four-bit message 1011 he would publish (y_1, x_2, y_3, y_4) . The described n -bit digital signature scheme is the original Lamport-Diffie one-time signature scheme (MERKLE, 1990) and was first proposed by Leslie Lamport in 1979 (LAMPOR, 1979). Similarly to the one-bit scheme, this scheme should also be used only once, otherwise existential forgery attacks would be possible.

Many hash-based signature schemes are built from this idea and can be used to sign only one message, thus they are called *One-Time Signature* (OTS) schemes. These include the aforementioned Lamport-Diffie OTS scheme, the Winternitz OTS scheme (W-OTS,

explained in Section 2.6) and many others. In order to sign more than one message with the same private key, a tree structure called the Merkle Tree (presented in Section 2.7) is frequently used. When a one-time signature scheme is used together with a Merkle Tree, the resulting scheme is called a *Many-Times Signature* (MTS) scheme, which can sign many (but a limited number) of messages with the same private key.

Hash-based signatures are fast compared to conventional digital signature schemes such as RSA and DSA and are secure even against quantum computers. Besides that, they are simple in their design and doesn't rely on any unproven number-theoretic assumptions, which increases the confidence in their security. Furthermore, even though they can sign only a limited number of messages (in contrast to RSA and DSA, which in principle can sign an unlimited number of messages), that limit can be set to a very high value (such as 2^{50}) without too much performance impact, therefore this limitation is only theoretical.

Early hash-based signature schemes had very large private and public keys (e.g. LD-OTS capable of signing 256-bit messages had public and private keys 32KB long) compared to RSA and DSA (which have keys less than 1KB long). However, this problem was solved by later designs which employed hash trees and pseudo-random number generators. Modern hash-based signature schemes can have key sizes as small as 256 bits for the same 256 bits of classic security.

Among the main challenges of hash-based signature schemes are their signature sizes, which are much larger than that of conventional digital signature schemes. For example, RSA-2048 has 256 Bytes signatures and ECDSA-256 has 64 Bytes signatures, whereas W-OTS (presented in the next section) and its improved versions with practical parameters have signature sizes not much smaller than 2KB (and W-OTS can only sign one message per key pair!). XMSS (BUCHMANN; DAHMEN; HÜLSING, 2011) and SPHINCS (BERNSTEIN et al., 2015), two modern hash-based many-times signature schemes, has signatures sizes around 10KB and 40KB, respectively. For this reason, much of the effort in hash-based signature schemes is devoted to decrease signature size or increase performance. The latter is a goal for schemes such as W-OTS(+), where increasing performance allows smaller signatures to be used with the same security level.

2.6 The Winternitz One-time Signature Scheme (W-OTS)

The Winternitz One-time Signature Scheme (W-OTS), which was proposed by Ralph Merkle (MERKLE, 1990), is a hash-based signature scheme that can be used to sign a single message per key pair. It is a generalization of LD-OTS with the introduction of hash-chains as a time-space trade-off; more specifically, with W-OTS it is possible to have smaller signature sizes at the cost of longer signature generation and signature verification times.

Consider a message m to be signed which is n bits long and a desired security level of b bits. Consider also a one-way function $f : \{0, 1\}^b \rightarrow \{0, 1\}^b$ (in practice, hash functions are used in the place of the ideal one-way functions). The Winternitz OTS has a adjustable parameter, w , which controls the tradeoff between signature size and processing time. With this tradeoff, messages are signed in w -bit blocks instead of one bit at a time (as was in LD-OTS). The scheme works as follows:

- **KEYGEN:** The signer generates L b -bit integers at random, $X = (x_0, x_1, \dots, x_{L-1})$, hashes each of them $2^w - 1$ times, producing $Y = (y_0, y_1, \dots, y_{L-1})$ where $y_i = f^{2^w - 1}(x_i)$. The total number of w -bit blocks, L , is calculated as follows:

$$L_1 = \left\lceil \frac{n}{w} \right\rceil, \quad L_2 = \left\lceil \frac{\lfloor \log_2 L_1 \rfloor + w + 1}{w} \right\rceil, \quad L = L_1 + L_2.$$

where L_1 is the number of w -bit blocks required to hold the message bits and L_2 is the number of w -bit blocks required to hold the checksum.

X is the private key and Y is the public key to be shared.

- **SIGGEN:** The message m is divided into L_1 blocks $(m_0, m_1, \dots, m_{L_1-1})$, each of w bits. Each one of these blocks are treated as integers, and the signer calculates the checksum c from the message as follows:

$$c = \sum_{i=0}^{L_1-1} 2^w - m_i$$

He then concatenates the L_1 blocks from the message with the L_2 blocks from the checksum, and obtains L w -bit blocks, denoted by $(B_0, B_1, \dots, B_{L-1})$. After that, he calculates the W-OTS signature

$$\sigma = (f^{B_0}(x_0), f^{B_1}(x_1), \dots, f^{B_{L-1}}(x_{L-1}))$$

- **VER:** The verifier, upon receiving m and $\sigma = (\sigma_0, \sigma_1, \dots, \sigma_{L-1})$, repeats the calculations done by the signer to generate the L w -bit blocks: he divides m into L_1 blocks of w bits each, then calculates the checksum c just like the signer did. He then concatenates the L_1 blocks from the message with the L_2 blocks from the checksum and obtains L w -bit blocks to be verified, $(B'_0, B'_1, \dots, B'_{L-1})$. Lastly, the verifier calculates

$$Y' = (y'_0, y'_1, \dots, y'_{L-1})$$

where

$$y'_i = f^{2^w - B'_i - 1}(\sigma_i)$$

(i.e. he "completes" the hash chains). If $Y' = Y$ then the signature is valid, otherwise it is invalid.

Figure 1 shows the W-OTS hash chains when used together with a pseudo-random generator to shorten the private key.

Figure 1 – The conventional W-OTS with PRF.

Source: Author

The correctness of the algorithm can be easily verified by observing a simple fact: if neither the message nor the signature has been corrupted or tampered with, then $B_i = B'_i$ (since both the signer and the verifier uses the same calculations from the message to get them) and $\sigma_i = f^{B_i}(x_i)$, therefore $y'_i = f^{2^w - B'_i - 1}(\sigma_i) = f^{2^w - B_i - 1}(f^{B_i}(x_i)) = f^{2^w - 1}(x_i) = y_i$ and the signature is valid.

It can be seen clearly that the security of W-OTS depends basically on the security of the one-way function f used. If f is second pre-image resistant (and thus pre-image resistant), the attacker cannot derive any piece of the private key from the public key.

However, note that even though an attacker cannot "go back" in the hash chains because of the one-wayness of f , he could "go forward". Because of this, given a pair of valid m and σ , if no checksum was used, he could forge new signatures from a previous legit signature by increasing the value of any M_i in the original message and going forward in the corresponding hash chains. For example, if $m = 0110$, $w = 2$ and $\sigma = (\sigma_0, \sigma_1)$, the attacker could forge a signature for $m' = 1010$ by providing the signature $\sigma' = (f(\sigma_0), \sigma_1)$. In this way, the attacker would be able to launch selective forgery attacks (but not universal forgery attacks). The checksum exists exactly to prevent this attack: with it, any increase in any message block m_i would inevitably cause a decrease in the checksum (and vice-versa), therefore with the checksum the attacker can only go forward in any hash chain if he goes back in at least one other chain, which he cannot do.

When compared to LD-OTS, W-OTS leads to smaller signature sizes at the cost of extra processing, using $(2^w - 1)$ -long hash chains to sign w bits simultaneously. For example, for $w = 2$, the scheme signs pairs of bits, so processing each pair takes 0, 1, 2 or 3 hash operations depending on whether the bits to be signed are 11, 10, 01 or 00, respectively; for $w = 3$, signing 3 bits would then take 0 to 7 hash operations for the bit strings 111 to 000, and so on.

The W-OTS scheme is relatively fast if reasonably large signature sizes are not a concern. The signature size is approximately

$$|\sigma| = \left\lceil \frac{n}{w} \right\rceil$$

while the number of hashes needed can be approximated by

$$\#(f) = \left\lceil \frac{n}{w} \right\rceil (2^w - 1)$$

when ignoring the checksum-related signature size and processing overhead (which grow only logarithmically with message length). One can easily see that, while the signature size is roughly inversely proportional to w , the processing time is exponential with w . For this reason, only small values of w (i.e. not much larger than 4) is used in practice, and that still yields signature sizes around 2KB for $n = 256$ bits.

Since the publishing of W-OTS scheme, many other OTS schemes were proposed, some of them as an improvement over W-OTS (e.g. W-OTS+ (HÜLSING, 2013)), while others are based upon different ideas to sign a single message (e.g. BiBa (PERRIG, 2001), HORS (REYZIN; REYZIN, 2002), and many others). These are explored in Section 2.8.

The next section explains how one may sign many messages with a construction based

on one-time signatures.

2.7 The Merkle Tree and Hash-based Many-times Signature Schemes

The Merkle tree, proposed by Ralph Merkle in the same paper as W-OTS (MERKLE, 1990), is a tree construct that allows the use of multiple one-time signatures (OTS) schemes as a single many-times signature (MTS) scheme.

Consider a one-time signature scheme with its public and private key pair. The Merkle tree works as follows: generate 2^{HEIGHT} OTS key pairs, where HEIGHT is the desired tree height. Then, hash the public keys two by two in a tree-like fashion. The collection of all OTS private keys are the private key of the MTS, and the root of the tree is the public key of the MTS. Now, in addition to the OTS signature, a set of values called the *authentication path* needs to be included in the MTS signature as well in order to verify the OTS signature using the MTS public key. These values are all the sibling nodes in the path from the current OTS public key node to the tree root. Figure 2 is an example with $\text{HEIGHT} = 3$.

As discussed in Section 2.6, reuse of an OTS key-pair would allow signature forgery, therefore some mechanism is needed to keep track of which OTS key pair was used. The simplest way to do this is to assign an index for each OTS key pair, e.g. from zero (leftmost OTS in the tree) to $2^{\text{HEIGHT}} - 1$ (rightmost OTS). In this way, a single integer is enough for the task.

Now, given a tree height HEIGHT and the underlying OTS scheme, the three steps KEYGEN, SIGGEN and VER of the MTS scheme works as follows:

- **KEYGEN:** Generate 2^{HEIGHT} OTS key pairs. The collection of all OTS private keys is the MTS private key. Then, hash the public keys in pairs in a binary tree structure. The root of the tree is the MTS public key.
- **SIGGEN:** Pick the current unused OTS key pair at index i , generate an OTS signature from that, and publish the MTS signature $\text{SIG} = (i, \sigma, \text{AUTH})$ where i is the key pair index, σ the OTS signature and $\text{AUTH} = (\text{AUTH}_0, \text{AUTH}_1, \dots, \text{AUTH}_{\text{HEIGHT}-1})$ the authentication path. Then, advance the OTS index by one unit to prevent OTS reuse.
- **VER:** With the message and the MTS signature, derive the OTS public key from

Figure 2 – A Merkle tree with HEIGHT = 3 as a MTS, where the individual OTS public keys are hashed once, then the hash results are hashed two-by-two in a binary tree structure until only a single value remains (the root of the tree). The "H" near the edges means the hashing operation. The highlighted gray boxes are the values in the authentication path corresponding to the highlighted OTS public key, and are needed in order to reconstruct the MTS public key from the signature.

Source: Author

the message and the OTS signature within the MTS signature. Then, use the index and the authentication path values to compute the path $(p_0, p_1, \dots, p_{\text{HEIGHT}})$ to the root of the Merkle tree, where p_0 is the public key of the OTS. To compute the other elements of the path, consider the index i in its binary form as a bit string $(i_{\text{HEIGHT}-1}, i_{\text{HEIGHT}-2}, \dots, i_1, i_0)$. Then,

$$p_j = \begin{cases} H(p_{j-1} || \text{AUTH}_{j-1}), & \text{if } i_{j-1} = 0 \\ H(\text{AUTH}_{j-1} || p_{j-1}), & \text{if } i_{j-1} = 1 \end{cases}$$

If p_{HEIGHT} matches the signer's authentic public key, accept the signature as valid; reject it otherwise.

MTS schemes built with Merkle trees are a simple way to sign many messages with a single associated public key, which was not possible with OTS schemes alone. Moreover, they are fast with reasonable values of HEIGHT, very simple to implement and, because they rely solely on one-way functions, their security is well understood.

However, these MTS schemes are not without disadvantages. One disadvantage is

their large private keys, which grows with HEIGHT. A simple solution that is adopted in most recent MTS schemes is to use a pseudo-random number generator to generate all OTS private keys when needed, so the seed of the pseudo-random number generator becomes effectively the private key of the MTS.

Even though a single Merkle-tree construction can be employed to allow the generation of an arbitrarily high number of signatures, the resulting scheme would not scale well because increasing the tree height increases the number of underlying OTS exponentially, which results in a costly private key generation process for the entire MTS. For example, an electronic messaging application that requires generating millions of signatures every day would require an MTS with many millions of underlying OTS. All those OTS public keys would have to be built during the key generation phase, since the root of the Merkle tree can only be computed if all of its leaves are provided. Hence, the cost of generating the MTS public key supporting 2^n signatures would be $O(2^n)$: it corresponds to the cost of generating keys for 2^n individual OTS, plus the cost of approximately 2^n hash operations for building the Merkle-tree. In practice, that could be a huge performance hit.

To remedy this, CMSS (BUCHMANN et al., 2006) proposes to decompose the original Merkle-tree into many layers of smaller trees, which are joined by intermediate OTS. In other words, the leaves of upper layer trees contain OTS for signing roots of lower layer trees, similarly to certificate chains in a Public Key Infrastructure (PKI). This starts from the single tree in the topmost layer and goes down until the bottom layer trees, whose leaves are the OTS for signing actual messages. This construction has the advantage of requiring only the topmost tree to be generated during the key generation phase. As a drawback, it increases the overall signature size, because the signature for messages needs to contain the signatures that authenticate the intermediate tree roots as well. This approach is incorporated in many modern MTS designs aimed at practical applications, such as XMSS^{MT} (HÜLSING; RAUSCH; BUCHMANN, 2013).

Yet another disadvantage of these kind of MTS schemes is the need to keep track of unused OTS schemes (they are called *stateful* schemes in the literature) (BERNSTEIN; BUCHMANN; DAHMEN, 2008). There exists hash-based schemes that address this problem, such as SPHINCS (BERNSTEIN et al., 2015). Nevertheless, the cost of eliminating statefulness is yet another increase in signature sizes. For example, an instantiation of SPHINCS, SPHINCS-256, has signature sizes as large as 41KB (compared to ≈ 10 KB for XMSS signatures).

Finally, the somewhat larger signature sizes of hash-based MTS schemes compared

to conventional signature schemes (such as RSA and ECDSA) remains. Much research in hash-based signature schemes is devoted to reduce signature sizes and/or processing time, nevertheless the current signature sizes are already practical in most applications with current technology, even with applications involving microcontrollers with limited processing power and memory (BUCHMANN; DAHMEN; HÜLSING, 2011) (HÜLSING; BUSOLD; BUCHMANN, 2013; HÜLSING; RIJNEVELD; SCHWABE, 2016). Moreover, newer schemes such as SPHINCS+ can already reduce the SPHINCS signature size from 41KB to around 30KB. More advancements are expected in the coming years.

2.8 Optimizations for hash-based signature schemes

As mentioned in Section 2.5, one of the main problems with hash-based signature schemes is their signature sizes, which are usually much larger than ECC-based schemes even when we consider only the OTS schemes (i.e., without taking into account the overhead resulting from revealing intermediate nodes of the Merkle tree in MTS schemes). One proposal for addressing this issue is to eliminate the need for collision-resistant hash functions, but requiring only pre-image resistance. In practice, to attain a security level of k -bits, this approach allows the adoption of k -bit hash functions (instead of $2k$ bits) in a classical setting, or $2k$ -bit hash functions (instead of $3k$ bits) in a quantum setting, thus leading to more compact signatures. This is the case of W-OTS+ scheme (HÜLSING, 2013), an improvement over W-OTS that uses different random bit-masks for each hash function call, thus eliminating the need for collision-resistant hash functions. To accomplish this, the hash functions in the hash chains linking private and public keys are replaced by a "chaining function", which consists basically in XORing the values with a random chain-level mask then feeding the result into a hash function.

Another general optimization proposed over pioneering hash-based signature schemes refers to the encoding of the messages to be signed. For example, in (STEINWANDT; VILLÁNYI, 2008) the authors propose that messages should first be hashed together with a counter $r \geq 0$, similarly to what is suggested in (PERRIG, 2001). The goal is to obtain a hash value d satisfying certain conditions, namely that there must be a maximum and minimum number of non-overlapping runs of consecutive 0's and 1's, each of which having a maximum length ℓ . Then, d is encoded into hash chains considering the run-lengths, not the integer values of each group of w bits. For example, if d starts with 11110011, the signature part corresponding to those bits would be given by $(H^{\ell-4}(x_0), H^{\ell-2}(x_1), H^{\ell-2}(x_2))$, where (x_0, x_1, x_2) are parts of the private key. The goal in this case is to speed up the

verification time at the cost of longer signature generation times. This happens for two reasons: first, the signer needs to find a suitable counter r for satisfying the scheme’s constraints; second, since short runs appear more often than long runs, the signer ends up computing a larger portion of the hash chain that links the private key x_i to the public key y_i (i.e., the number of hash calls at the signer is closer to ℓ , whereas at the verifier it is closer to 0).

Finally, a somewhat orthogonal concern with hash-based signatures is that they lead to stateful systems, i.e., the signer must keep information about previously used keys to prevent reuse (BERNSTEIN et al., 2015). After all, since most MTS schemes are built over OTS schemes, their security relies on the fact that no key is used twice, since that would fatally break the security of the system (e.g., allowing signatures to be forged). One of the few schemes in the literature that addresses this problem is SPHINCS (BERNSTEIN et al., 2015) a stateless hash-based signature solution that relies on a large hyper-tree structure with HORST, a variant of the HORS few-times signature scheme. The price paid by SPHINCS for its statelessness, however, is that the scheme produces even larger signatures than its stateful counterparts: about 41 KB for SPHINCS-256, which provides 128 bits of quantum security.

3 THE CLIPPING TECHNIQUE FOR W-OTS SCHEME AND ITS VARIANTS

Aiming to allow shorter signatures, in this section a novel technique that builds upon the manner by which private keys are usually generated from a seed in hash based schemes is described. Specifically, most OTS hash-based signature schemes use a single layer of pseudo-random numbers generated from a seed, using the resulting x_i as the private key (see Figure 1, which shows this approach for the W-OTS scheme). Then, during signature generation, x_i is sometimes revealed directly, whereas other times what is revealed is $H^j(x_i)$ for some j . The basic idea of the hereby proposed "clipping technique" is to leverage on situations where consecutive parts of the private key (say, x_i and x_{i+1}) would be naturally revealed. In such cases, a single piece of information that allow the generation of both x_i and x_{i+1} can be revealed, thus compressing the total signature size. This is accomplished by using two layers of pseudo-random functions to generate when computing the scheme's private keys. As such, the technique applies to W-OTS+, W-OTS and any other OTS that relies on pseudo-random generators for deriving secret values that may be revealed as part of the signature itself.

Figure 3 illustrates the proposed approach, using W-OTS as the base scheme (since it is simpler to explain). In a nutshell, the main modification on the scheme lies in the fact that the OTS's private values x_i are not derived directly from a seed. Instead, a master seed (denoted $seed_M$) is employed in the generation of a single layer of intermediate seeds $seed_i$, each of which is subsequently employed for generating pairs of private values x_{2i} and x_{2i+1} . Then, whenever a signature should reveal those pair of private values sharing the same lower layer seed, this seed is revealed in their place. More formally, still using W-OTS as basis, the hereby proposed clipping technique can be described as follows.

Figure 3 – The improved W-OTS scheme with PRF, where $x_{2i+1} = seed_i \oplus x_{2i}$. For example, x_1 is generated by calculating $seed_0 \oplus x_0$.

Source: Author

3.1 Set up

Given n , w , f , and PRF as defined in the W-OTS scheme, calculate L_1 , L_2 and L also like in the original W-OTS. The resulting signature scheme is the triple $SIGN = (GEN, SIG, VER)$, as described below.

3.2 Key Generation

Use two layers of pseudo-random sequences to generate the L pseudo-random integers $(x_0, \dots, x_{L-1})$. For the first layer, the user selects an n -bit integer as the master seed $seed_M$ (the private key of the OTS). Using that seed, generate the first layer of n -bit pseudo-random values $seed_i$, for $0 \leq i < L/2$. Then, for the second layer, generate the private value x_{2i} as $PRF(seed_i)$, and its companion private value $x_{2i+1} = seed_i \oplus x_{2i}$. As a result, the second layer contains the private values $X = (x_0, \dots, x_{L-1})$, from which each part of the public key $pk = Y = (y_0, \dots, y_{L-1})$ are computed as $y_i = f^{2^w-1}(x_i)$, as usual.

3.3 Signature Generation

The signer calculates

$$d||c = b_0||b_1||\dots||b_{L-2}||b_{L-1}$$

where d is the w -bit blocks from the message digest and c is the w -bit blocks from the checksum, and each block b_i is interpreted as the binary representation of an integer, just like in the W-OTS scheme. The difference to W-OTS, however, is that instead of processing each block of $d||c$ separately, they are processed two blocks at a time, either sequentially or in parallel. More precisely, for each pair of blocks b_{2i} and b_{2i+1} , where $0 \leq i \leq L/2$, if either b_{2i} or b_{2i+1} is non-zero, the usual hashing chain is used: $f^{b_{2i}}(x_{2i})$ and $f^{b_{2i+1}}(x_{2i+1})$ are published with the signature, and the signer then proceeds with the next two blocks b_{2i+2} and b_{2i+3} . However, if both b_{2i} and b_{2i+1} are sequences of zero bits, the shared seed in the lower layer is published instead of $f^{b_{2i}}(x_{2i})$ and $f^{b_{2i+1}}(x_{2i+1})$ and the signer proceeds with the next two blocks b_{2i+2} and b_{2i+3} . The signer repeats the same procedure for all pairs of blocks until the entire $d||c$ is processed. The resulting signature $\sigma = (\sigma_0, \sigma_1, \dots, \sigma_{S-2}, \sigma_{S-1})$ is sent alongside the message. Note that, with this technique, the signature can have variable length depending on the message to be signed.

3.4 Signature Verification

With the message and the signature σ , the verifier computes the digest d and the checksum c , and calculates $d||c = b_0||b_1||\dots||b_{L-2}||b_{L-1}$ just like in the original verification operation. The verifier then proceeds considering pairs of blocks, which can be processed either sequentially or in parallel. For a sequential verification, let $i = 0$ and $k = 0$ be indices for the blocks from $d||c$ and from the signature σ , respectively. For each pair of blocks b_{2i} and b_{2i+1} in $d||c$, where $0 \leq i \leq L/2$, if either block is non-zero the verification is done as usual, by evaluating $y'_{2i} = f^{2^w-1-b_{2i}}(\sigma_k)$ and $y'_{2i+1} = f^{2^w-1-b_{2i+1}}(\sigma_{k+1})$; the verifier then proceeds to the next two blocks, updating the i and k indices by making $i = i + 2$ and $k = k + 2$. However, if both b_{2i} and b_{2i+1} are zeros, the verifier first calculates $x'_{2i} = PRF(\sigma_k)$ and $x'_{2i+1} = x'_{2i} \oplus \sigma_k$, then calculates $y'_{2i} = f^{2^w-1}(x'_{2i})$ and $y'_{2i+1} = f^{2^w-1}(x'_{2i+1})$; then, i and k are updated by $i = i + 2$ and $k = k + 1$, since a single block from the signature was used to process two blocks from $d||c$. The verifier repeats this process until all blocks have been processed; if this process results in exactly L values $(y'_0, \dots, y'_{L-1})$ and $Y' = (y'_0, \dots, y'_{L-1}) = Y$, the signature is valid; otherwise, it is invalid.

A parallel verification follows an analogous process: (1) start by identifying pairs of blocks b_{2i} and b_{2i+1} from $d||c$ that are both zeros, thus learning the indices k of blocks from the signature that cover those pairs of blocks; (2) compute the corresponding $x'_{2i} = PRF(\sigma_k)$ and $x'_{2i+1} = x'_{2i} \oplus \sigma_k$; and finally (3) proceed with the usual verification process, checking whether $y'_i = f^{2^w-1-b_i}(\sigma_i) = y_i$ for the blocks not covered by the previous steps,

and whether $y'_i = f^{2^w-1}(x'_i) = y_i$ for the covered blocks.

3.5 Correctness of the algorithm

Similar to the original W-OTS(+) algorithm, the correctness of the "clipped" algorithm is straightforward. The basic idea is, both the signer and the verifier are working together to rebuild the hash chains built in the key generation phase; the message only determines where the signer stops and the verifier continues the work. If neither the message nor the signature were tampered with, then the public key reconstructed in the verification process certainly matches the genuine public key resulting from the key generation process.

Suppose neither the message nor the signature has been corrupted or tampered with. It follows that $b_i = b'_i$ for all i (since both the signer and the verifier uses the same calculations from the message to get them). Also, for each pair of values b_i and b_{i+1} (or b'_i and b'_{i+1}) there is a corresponding σ_j in the signature if both b_i and b_{i+1} were zeros, or a corresponding pair σ_j and σ_{j+1} otherwise.

If both b_i and b_{i+1} are zeros, since the signature is authentic, it follows that $\sigma_j = seed_i$, so the reconstruction

$$\begin{cases} x'_i = PRF(\sigma_j) \\ x'_{i+1} = \sigma_j \oplus x'_i \\ y'_i = f^{2^w-1}(x'_i) \\ y'_{i+1} = f^{2^w-1}(x'_{i+1}) \end{cases}$$

satisfies $y'_i = y_i$ and $y'_{i+1} = y_{i+1}$.

On the other hand, if at least one of $\{b_i, b_{i+1}\}$ is non-zero, then the proof is analogous to that of the original W-OTS: since $\sigma_i = f^{b_i}(x_i)$ and $\sigma_{i+1} = f^{b_{i+1}}(x_{i+1})$, we would have that

$$\begin{cases} y'_i = f^{2^w-b'_i-1}(\sigma_i) = f^{2^w-b_i-1}(f^{b_i}(x_i)) = f^{2^w-1}(x_i) = y_i \\ y'_{i+1} = f^{2^w-b'_{i+1}-1}(\sigma_{i+1}) = f^{2^w-b_{i+1}-1}(f^{b_{i+1}}(x_{i+1})) = f^{2^w-1}(x_{i+1}) = y_{i+1} \end{cases}$$

Therefore, we have proven that $y'_i = y_i$ for all i and, thus, the signature is valid.

3.6 Example

As a simplified example, suppose the digest of a message to be signed is $d = 00101100$ — here the checksum is omitted for the sake of simplicity, but it would need to be included in the message and signed as well. Using $w = 1$, the private key would be $X = (x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$. In the traditional W-OTS scheme, the signature would include all eight x_i hashed zero or one time, thus one would obtain the signature $\sigma = (f^0(x_0), f^0(x_1), f^1(x_2), f^0(x_3), f^1(x_4), f^1(x_5), f^0(x_6), f^0(x_7))$. With the proposed clipping technique, however, there are two pairs of blocks that are both zeros: the first (b_0 and b_1) and the last (b_6 and b_7) one. Therefore, the corresponding seeds are published, so the resulting signature becomes $\sigma = (seed_0, f^1(x_2), f^0(x_3), f^1(x_4), f^1(x_5), seed_3)$. In this particular case, the signature becomes 25% shorter compared to the original W-OTS scheme (again, without taking the checksum into account for the sake of simplicity).

4 ANALYSIS OF THE CLIPPING TECHNIQUE

In this chapter the proposed clipping technique is evaluated considering the resulting signature size, processing time, and underlying security, assuming it is applied to W-OTS+ or W-OTS. In summary, we show that the (probabilistic) reduction in signature size is attained with negligible performance overheads and no security impacts to those schemes.

4.1 Performance Analysis

In this section we analyze the signature size, processing overhead and storage requirements of W-OTS(+) with the clipping technique.

4.1.1 Signature Size

As described for W-OTS in Section 2.6, it is assumed that the hash-based signature scheme uses a PRF for generating the secret n -bit values x_i from a seed, which plays the role of the private key. The message to be signed is converted into the bit-string $d||c$, with L blocks of w bits, where each block is interpreted as an integer from zero to $2^w - 1$. In this scenario, the message digest d follows a uniform distribution, since this is a necessary condition for any secure cryptographic hash function; similarly, the checksum c obtained from the uniformly distributed blocks from d is also expected to be uniformly distributed. Therefore, the probability of one w -bit block from $d||c$ being filled with zeros is $\frac{1}{2^w}$, whereas the probability of two adjacent blocks from $d||c$ being both zero is $\frac{1}{2^{2w}}$.

In this scheme, due to the way the pseudo-random number layers are created, any two adjacent blocks b_{2i} and b_{2i+1} in $d||c$ are signed using secret values x_{2i} and x_{2i+1} that share the same seed. Therefore, the whole message to be signed can be seen as a concatenation of two-block units, and the analysis of how much data is omitted from the signature when the seed is revealed instead of each x_{2i} and x_{2i+1} gives the overall signature size reduction.

For each two-block unit, there are two possibilities: if both blocks contain a zero value (i.e., with probability $1/2^{2w}$), then the contribution to the signature is a single n bit integer, the lower layer seed; otherwise (i.e., with probability $1 - 1/2^{2w}$), two n -bit strings are published, adding $2n$ bits to the signature. The average bit-length of each two-block unit when the clipping technique is applied is, thus:

$$\text{AvgLen} = 2n \times \frac{2^{2w} - 1}{2^{2w}} + n \times \frac{1}{2^{2w}} = \frac{2^{2w+1} - 1}{2^{2w}}n$$

The savings of an average "clipped" two-block unit, when compared with a regular, non-"clipped" two block unit can be calculated as:

$$\text{SAVINGS} = \frac{2n - \text{AvgLen}}{2n} = \frac{1}{2^{2w+1}}$$

Finally, since the whole signature is just concatenations of independent two-block units, the savings in terms of signature size is equivalent to the savings for each unit.

For $n = 256$ bits and $w = 1$, for example, we have $\text{SAVINGS} = 12.5\%$, which means that, for those parameters, when applying the clipping technique, the average signature size would be 12.5% shorter than the signature size of the scheme without the technique.

Note that the savings decrease exponentially with w , so smaller values of w lead to better compression. This is illustrated in Table 1, which shows the average savings for different values of w . Given its probabilistic nature, however, in practice the clipping technique may be useful for compressing signatures even for $w \geq 3$, depending on the actual data to be signed.

Table 1 – Average signature size for the original W-OTS+ and the "clipped" W-OTS+, obtained with the proposed approach, depending on the w parameter. A 256-bit (classical) security level is assumed.

w	Signature Size		Average Savings (s)
	Original W-OTS+	"Clipped" W-OTS+	
1	8192 bytes	7168 bytes	12.50%
2	4096 bytes	3968 bytes	3.13%
3	2731 bytes	2710 bytes	0.78%
4	2048 bytes	2044 bytes	0.20%

Source: Author

4.1.2 Processing Time

In terms of processing, the proposed clipping technique affects basically the generation of the secret values x_i from a private key $seed_M$, either during the GEN and VER operations. Even though the two-layer construction may appear to lead to a less efficient GEN than the original, single-layered approach, the fact is that both perform the same number of PRF calls per x_i . More precisely, as observed in Figure 1, each x_i takes one call to the PRF, whereas the clipping technique shown in Figure 1 takes two calls to the PRF plus one XOR operation for each *pair* (x_{2i}, x_{2i+1}) . Since XOR operations have a negligible cost, both approaches end up being similar in terms of processing time.

The VER operation, on the other hand, would require one extra PRF call for each pair of blocks that can be "clipped". Assuming the same cost for PRF calls and hash function calls, the gains in terms of signature size translate to an equivalent overhead in terms of processing for $w = 1$: where two calls to the hash function would be made to compute (y'_{2i}, y'_{2i+1}) from $(\sigma_{2i}, \sigma_{2i+1})$, there is an extra call to the PRF (i.e., a 50% overhead) to recover (x'_{2i}, x'_{2i+1}) ; in this case, a pair of n -bit blocks is replaced by the corresponding n -bit seed (i.e., a 50% saving). For larger values of w , on the other hand, this extra PRF call becomes less pronounced when compared to the $2 \cdot (2^w - 1)$ hash function calls required to compute y'_{2i} and y'_{2i+1} . Hence, albeit the signature size gains obtained drop exponentially with w , so does the processing overhead, thus allowing interesting trade-offs.

As a final remark, another aspect of the proposed scheme refers to parallelism. Specifically, W-OTS+ and W-OTS is such that, with L or more cores, one could compute all values of x_i in parallel using a PRF. In comparison, in our proposal only half of the values of x_i can be computed simultaneously (the even ones), since the other half (the odd ones) is dependent on the former. Still, the performance impact is relatively minor, as it only adds a constant processing overhead (namely, one PRF call) in the processing flow when compared to the fully parallel approach.

4.1.3 Storage Requirements

The improved scheme requires the same storage as W-OTS+ or W-OTS, namely, the public and private keys for the signer and the public key for the verifier. Therefore, it has no storage overhead compared to the original signature scheme.

4.2 Security Considerations

The overall security of the clipping technique is inherited from the hash-based signature scheme to which it is applied. After all, except for the manner by which the private key $seed_M$ is processed for generating the secret x_i , our proposal does not change anything on the signature scheme itself. Therefore, the main security aspect that needs to be assessed is whether the proposed two-layer construction reveals any information it should not reveal to attackers. In what follows, it is shown that this is not the case.

First, note that $seed_i$ is only explicitly revealed when both x_{2i} and x_{2i+1} need to be public anyway, i.e., when b_{2i} and b_{2i+1} are zeros. If only $x_{2i} = PRF(seed_i)$ is revealed, on the other hand, its companion $x_{2i+1} = seed_i \oplus x_{2i}$ cannot be obtained by attackers; after all, $seed_i$ acts as a one-time pad to protect the secrecy of x_{2i+1} , whereas computing $seed_i$ from x_{2i} requires inverting the PRF. An analogous argument applies to the task of computing x_{2i} when x_{2i+1} is disclosed, since by symmetry $seed_i$ acts as a one-time pad for $x_{2i} = x_{2i+1} \oplus seed_i$. Finally, when neither x_{2i} or x_{2i+1} are disclosed, but rather $f^\alpha(x_{2i})$ and $f^\beta(x_{2i})$ for some $\alpha, \beta > 0$, their secrecy remains protected by the pre-image resistance of the hash function f , as usual. Therefore, the clipping technique does not bring any negative impact in terms of security.

4.3 Proof of Concept

A proof of concept implementation was developed in C language using the OpenSSL library for the SHA256 hash function ($n = 256$ bits and $d = 256$ bits). It was compiled with the usual -O2 flag without any special hardware optimizations. Ten thousand KeyGen-Sign-Verify tests were performed in an Intel Core i5 750 machine with 6GB of RAM, which produced the results summarized in Figure 4. This figure shows that, for $w = 1$, signing was about 6% faster when using the technique, although the verification was about 14% slower. This occurs because, with the proposed technique, some PRF calls are moved from the signing operation to the verification operation. Other than that, all differences were well within their standard deviations, which shows that the smaller signature sizes obtained with the technique have a low impact in terms of performance.

Figure 4 – Summary of the performance comparison between W-OTS and W-OTS with the clipping technique. The standard deviation is shown in the error bars.

Source: Author

5 USING HASH CHAINS WITH Z-NAF: THE Z-NAF OTS SCHEME

In this chapter, a novel OTS is presented, called z-NAF OTS. The z-NAF OTS is a hash-based OTS conceived using the idea that, in the z-NAF form, integers have significantly higher zero-digit density than non-zero digits density. Hence, the average signature hash chain length built upon those digits is skewed towards the zero-digit end. One could take advantage of this fact to make signing faster or verification faster, depending on the chosen hash chain mapping. We take the latter approach in this work to optimize signature verification time over existing hash-based OTS schemes (such as the W-OTS family of signature schemes).

Similar to W-OTS, the z-NAF OTS uses hash chains to sign many bits at once. The difference is that, instead of signing the message directly, it first searches for a predefined pattern using randomized hashes of the message, then it encodes the message to be signed into z-NAF form (as described in Section 2.2.2), and finally it signs the message in that form by mapping the z-NAF digits into the hash chains. In this way, one obtains an OTS scheme that combines: (1) the advantages of W-OTS, namely that it is fast, has small keys and the public key can be recovered from the signature; and (2) faster verification times at the cost of a slightly slower signature generation time.

For added security, one could also replace the one-way function by a chaining function with chain level masks, in the same way that W-OTS+ does to improve the security of the traditional W-OTS scheme. Since the substitution is straightforward, in this chapter the traditional one-way function is used in the description of the algorithm for the sake of simplicity.

It is important to note that, while there exists hash-based digital signature schemes with much faster verification speeds compared to this scheme, such as BiBa (PERRIG, 2001) and HORS (REYZIN; REYZIN, 2002), they differ from the W-OTS family of OTS schemes in that they lack the ability to recover the public key from the signature alone when the signature is genuine. Because of this, if one wishes to use these schemes with

current many-times signature (MTS) schemes, one would need to send the public key along with the original OTS signature, which would make the total signature size too large to be practical. Besides, compared to the W-OTS family of OTS schemes, the BiBa and HORS OTS schemes also lack parameters that provide the ability to adjust the signature size and processing overhead trade-off, making the latter less flexible than the former schemes.

The z-NAF OTS has none of the aforementioned problems, making it an ideal candidate to serve as a drop-in replacement for the W-OTS family of hash-based OTS schemes used in current MTS schemes.

The z-NAF OTS scheme is described more formally as follows.

5.1 Setup

Let n be the length of the messages to be signed, z the z-NAF parameter, b the desired classical security level and w the Winternitz parameter. In this scheme, w is only used when signing the checksum of the message, since the checksum is not converted into a z-NAF form. Furthermore, define L_1 , the number of hash chains that are used to sign the z-NAF digits, and t_{max} , a parameter that, together with the parameter z , defines the length of those hash chains. The meaning of these two parameters is clarified in Section 5.3 and in Chapter 6; these parameters can be tweaked in the scheme to allow different signature size and performance trade-offs.

After the definition of the aforementioned parameters, calculate

$$L_2 = \left\lceil \frac{\lceil \log_2 L_1 + \log_2(1 + t_{max}) \rceil + z - 1}{w} \right\rceil$$

Also calculate the total message chain length, len , as

$$len = (1 + t_{max})2^{z-1}$$

Like in W-OTS(+), let

$$f : \{0, 1\}^b \rightarrow \{0, 1\}^b$$

be a one-way function and

$$g : \{0, 1\}^* \rightarrow \{0, 1\}^n$$

be a cryptographic hash function. In addition, let

$$h : \{0, 1\}^{b+|k|} \rightarrow \{0, 1\}^b$$

be a cryptographic hash function, where $|k|$ is the length of the nonce k used in the signature generation algorithm described later in Section 5.3 (in practice, all functions f , g and h are usually implemented using the same hash function H). Also, let PRF be a cryptographically secure pseudo random function that when given a b -bit seed SEED_{in} outputs a random number RAND and an updated seed SEED_{out} , both of bit-length b .

5.2 Key Generation

Generate a b -bit private key x randomly, then use it and the PRF to generate $L_1 + L_2$ secret b -bit strings, $((x_m)_{L_1-1}, \dots, (x_m)_0)$ and $((x_c)_{L_2-1}, \dots, (x_c)_0)$. Here, the (x_m) denotes the secret strings used to sign the message, whereas the (x_c) denotes those used to sign the checksum.

After that, calculate

$$\begin{cases} (y_m)_i = f^{len-1}((x_m)_i) \\ (y_c)_i = f^{2^w-1}((x_c)_i) \end{cases}$$

Finally, calculate the public key $y = g((y_m)_{L_1-1} || \dots || (y_m)_0 || (y_c)_{L_2-1} || \dots || (y_c)_0)$. The private-public key pair is (x, y) .

Figure 5 shows the hash chains calculated in the key generation algorithm.

5.3 Signature Generation

Given the private key x and a n -bit long message m to be signed, calculate the randomized hash $m' = h(m || k)$, where k is a nonce. In a simple implementation, this nonce could be a counter value starting at zero, where the next nonce value is simply the previous nonce value increased by one. However, if one desires to eliminate the need for collision resistance of h through the use of randomized hashing, one should generate k (pseudo-)randomly.

Next, convert m' to z-NAF form, $m'_{z-NAF} = (D_{n+1}, D_n, D_{n-1}, \dots, D_1, D_0)$. Due to the z-NAF properties discussed in Section 2.2.2, we know that no n -bit integer has a z-NAF representation with more than $n + 1$ digits. Besides, the digits D_i are either zero

Figure 5 – Hash chains of the key generation algorithm.

Source: Author

or odd and $-2^{z-1} < D_i < 2^{z-1}$. Therefore, there are $2^{z-1} + 1$ possible values for each D_i .

Now, in the z-NAF OTS scheme, we require that m'_{z-NAF} must satisfy the following conditions:

- The least significant digit D_0 must not be zero;
- The number of non-zero digits in m'_{z-NAF} must be equal to or less than L_1 ;
- The number of trailing zeros at the left of each non-zero digit must be equal to or less than $z - 1 + t_{max}$. The $z - 1$ zeros in the left necessarily appear as per the z-NAF properties; hence, we require that, besides those mandatory zeros, the additional zero length cannot exceed t_{max} .

If any of the above three conditions is not satisfied, pick another nonce value and generate a new m'_{z-NAF} following the previous steps. On the other hand, if all three conditions are met, then discard $z - 1$ zeros on the left of each non-zero digit in m'_{z-NAF} and group the remaining digits into blocks starting at D_0 , where each block has a non-zero digit at its right followed by the number of zeros at its left up to the next non-zero digit. The result of this operation has the following form: $m'_{grouped} = (D'_r, D'_{r-1}, \dots, D'_1, D'_0) = (0^{t_r} N_r, 0^{t_{r-1}} N_{r-1}, \dots, 0^{t_1} N_1, 0^{t_0} N_0)$, where

- 0^{t_i} denotes a zero-run of length t_i ;

- N_i are the non-zero digits that were present in m'_{z-NAF} ;
- $t_i \leq t_{max}$ as per our previous requirements on m'_{z-NAF} ;
- $r \leq \lceil \frac{n+1}{z} \rceil$, because there are at least $z - 1$ zeros at the left of each non-zero digit in m'_{z-NAF} , except when the non-zero digit is in one of the far left positions in m'_{z-NAF} .

The form of $m'_{grouped}$ shows why the condition of non-zero D_0 is important; without that condition, the rightmost sequence of zeros would have no non-zero digits to group with.

Note that the operation of eliminating the mandatory zeros does not result in ambiguity, that is, it does not cause two different messages to result in the same $m'_{grouped}$ value. The idea is that those zeros are always present in m'_{z-NAF} to be removed and the removal operation can always be reverted. For a more detailed explanation, refer to the security analysis in Section 6.1.

With $m'_{grouped}$ at hand, generate $m'' = (D''_{L_1-1}, D''_{L_1-2}, \dots, D''_1, D''_0)$ using the map $D'_i \mapsto D''_i$ defined as follows:

$$D''_i = \begin{cases} t_i 2^{z-1} + (N_i - 1)/2 & \text{if } N_i > 0 \text{ and } i \leq r \\ t_i 2^{z-1} + (2^z + N_i - 1)/2 & \text{if } N_i < 0 \text{ and } i \leq r \\ 0 & \text{if } i > r \end{cases}$$

This map was chosen because it is relatively simple to implement using additions, bit shifting operations and one multiplication (by t_i), and because it maps blocks D'_i with longer zero tails to larger values of D''_i , which would be translated into higher positions in the hash chains. In this way, the hash chain lengths in the signing operation are maximized and the hash chain lengths in the verification operation are minimized. Despite the seemingly complicated equations, what this map does is basically mapping the signed digits in two's complement form; for example, for $z = 3$ the non-zero digits $\{-1, 1, -3, 3\}$ mapping would be as follows:

$$\begin{cases} 1 \rightarrow 0 \\ 3 \rightarrow 1 \\ \bar{3} \rightarrow 2 \\ \bar{1} \rightarrow 3 \end{cases}$$

whereas each additional digit would add $2^{z-1} = 4$ to D_i'' , so with one leading zero the mapping would be

$$\begin{cases} 01 \rightarrow 4 \\ 03 \rightarrow 5 \\ 0\bar{3} \rightarrow 6 \\ 0\bar{1} \rightarrow 7 \end{cases}$$

Besides, if the total number of blocks D_i' is less than L_1 , the remaining D_i'' digits are set to zero (third equation of the map).

We note that other maps (and possibly more efficient ones) are possible, as long as they remain unambiguous (meaning no two different m' results in the same m''), $0 \leq D_i' < len$ and longer trailing zeros are mapped to larger values of D_i'' .

Of course, the steps from converting m' to m'_{z-NAF} , to $m'_{grouped}$ and then to m'' can be done all at once and, as a result, some operations could be simplified. Nevertheless, they are described separately here for the sake of clarity.

Figure 6 summarizes the conversion process from m to m'' before the digits in m'' are mapped to the hash chains.

Figure 6 – Conversion chain of z-NAF OTS in the signing algorithm, showing how m is converted to m'' .

Source: Author

The remaining steps of the signing algorithm comprises calculating the checksum, mapping the digits D_i'' to the hash chains and then calculating the hash chains themselves. The checksum calculation is similar to W-OTS(+), where the user calculates the checksum from m'' as

$$c = \sum_{i=0}^{L_1-1} len - 1 - D_i''$$

and divides the checksum into L_2 blocks of w bits each, $(B_{L_2-1}, \dots, B_0)$.

Finally, the signature is $\sigma = (k, (\sigma_m)_{L_1-1}, \dots, (\sigma_m)_0, (\sigma_c)_{L_2-1}, \dots, (\sigma_c)_0)$ where

$$\begin{cases} (\sigma_m)_i = f^{len-1-D''_i}((x_m)_i) \\ (\sigma_c)_i = f^{2^w-1-B_i}((x_c)_i) \end{cases}$$

Note that the length of the nonce k can be much smaller than b (e.g., 32 bits), as h is probably called much fewer than 2^{32} times, and no one would choose the z-NAF OTS parameters to require that many calls to h . However, if one desires to eliminate the collision resistance of h , then the length of k cannot be that small.

Figure 7 shows the hash chains that are calculated in the signature generation algorithm.

Figure 7 – Hash chains calculated in the signature generation algorithm.

Source: Author

5.4 Signature Verification

Given a message m , its signature $\sigma = (k, (\sigma_m)_{L_1-1}, \dots, (\sigma_m)_0, (\sigma_c)_{L_2-1}, \dots, (\sigma_c)_0)$ and the public key y , the verifier calculates m'' and the checksum c as in the signing algorithm. If, in the midst of the calculations, m'_{z-NAF} does not satisfy all three conditions given in the signing algorithm, the verifier rejects the signature as invalid. Otherwise, the verifier calculates y' (i.e. "completes" the hash chains) as follows:

$$y' = g((y_m)'_{L_1-1} || \dots || (y_m)'_0 || (y_c)'_{L_2-1} || \dots || (y_c)'_0)$$

where

$$\begin{cases} (y_m)_i' = f^{D''_i}((\sigma_m)_i) \\ (y_c)_i' = f^{B_i}((\sigma_c)_i) \end{cases}$$

If $y' = y$ then the signature is valid, otherwise it is invalid.

Figure 8 shows the hash chains calculated in the signature verification algorithm after the given m and k are used to obtain m'' and the checksum.

Figure 8 – Hash chains calculated in the signature verification algorithm.

Source: Author

5.5 Correctness of the algorithm

The correctness of the algorithm is shown in a similar way as that of the W-OTS or the clipped W-OTS(+). The idea is, since the hash chains built in the key generation process are the same ones being built in the signing and verification process combined, the public key reconstructed from the signature would necessarily match the genuine public key when the signature is authentic.

Suppose the message has not been corrupted or tampered with and the signature is authentic. Then, given the message m and an index k chosen by the signer, both the signer and the verifier would arrive to the same $m'' = (D''_{L_1-1}, D''_{L_1-2}, \dots, D''_1, D''_0)$ (since both of them perform exactly the same calculations from the message and the index to get it). Analogously, from the same m'' both would arrive to the same checksum blocks $(B_{L_2-1}, \dots, B_0)$.

Recall that the public key in the signature is given by

$$y = g((y_m)_{L_1-1} || \dots || (y_m)_0 || (y_c)_{L_2-1} || \dots || (y_c)_0)$$

where

$$\begin{cases} (y_m)_i = f^{len-1}((x_m)_i) \\ (y_c)_i = f^{2^w-1}((x_c)_i) \end{cases}$$

Also recall that the signer performs the calculation

$$\begin{cases} (\sigma_m)_i = f^{len-1-D''_i}((x_m)_i) \\ (\sigma_c)_i = f^{2^w-1-B_i}((x_c)_i) \end{cases}$$

and the verifier calculates $y' = g((y_m)'_{L_1-1} || \dots || (y_m)'_0 || (y_c)'_{L_2-1} || \dots || (y_c)'_0)$ where

$$\begin{cases} (y_m)'_i = f^{D''_i}((\sigma_m)_i) \\ (y_c)'_i = f^{B_i}((\sigma_c)_i) \end{cases}$$

Since both the signer and the verifier have the same values of D''_i and B_i , one can substitute the signer's equations into the verifier's and get

$$\begin{cases} (y_m)'_i = f^{D''_i}((\sigma_m)_i) = f^{D''_i}(f^{len-1-D''_i}((x_m)_i)) = f^{D''_i+len-1-D''_i}((x_m)_i) = f^{len-1}((x_m)_i) = (y_m)_i \\ (y_c)'_i = f^{B_i}((\sigma_c)_i) = f^{B_i}(f^{2^w-1-B_i}((x_c)_i)) = f^{B_i+2^w-1-B_i}((x_c)_i) = f^{2^w-1}((x_c)_i) = (y_c)_i \end{cases}$$

Thus, we have proven that $(y_m)'_i = (y_m)_i$ and $(y_c)'_i = (y_c)_i$ and, as a result, $y' = y$.

5.6 Example

Here we give an example for $z = 3$. For additional examples with $z = 2$ and $z = 4$, please refer to Appendix A.

5.6.1 Setup

In this example, let $n = 8$ bits, $z = 3$, $w = 4$, $L_1 = 4$ and $t_{max} = 3$. With these parameters, $L_2 = 2$ and $len = 16$.

5.6.2 Key Generation

The signer generates the key pair (x, y) with the method described in Section 5.2 and the above defined parameters. First, the signer generates x randomly, then he uses x and the PRF to generate $((x_m)_3, (x_m)_2, (x_m)_1, (x_m)_0, (x_c)_1, (x_c)_0)$ and calculates

$$y = g((y_m)_3 || (y_m)_2 || (y_m)_1 || (y_m)_0 || (y_c)_1 || (y_c)_0)$$

where

$$\begin{cases} (y_m)_i = f^{len-1}((x_m)_i) = f^{15}((x_m)_i) \\ (y_c)_i = f^{2^w-1}((x_c)_i) = f^{15}((x_c)_i) \end{cases}$$

5.6.3 Signature Generation

To sign the message m , the first thing the signer needs to do is choose an appropriate index k and calculate m' and m'' . Suppose that the signer tried the following values of k_i , with their respective m'_i and $(m'_i)_{3-NAF}$:

$$k_1 \rightarrow m'_1 = h(m||k_1) = 11001010 \rightarrow (m'_1)_{3-NAF} = (1, 0, 0, 0, \bar{3}, 0, 0, \bar{3}, 0)$$

$$k_2 \rightarrow m'_2 = h(m||k_2) = 11111111 \rightarrow (m'_2)_{3-NAF} = (1, 0, 0, 0, 0, 0, 0, 0, \bar{1})$$

$$k_3 \rightarrow m'_3 = h(m||k_3) = 01101111 \rightarrow (m'_3)_{3-NAF} = (0, 1, 0, 0, \bar{1}, 0, 0, 0, \bar{1})$$

The index k_1 does not work, because the resulting m'_{3-NAF} has the rightmost digit zero. The index k_2 does not work either, because, even though the rightmost digit is not zero, the zero run length at the left of the $\bar{1}$ digit exceeds $z - 1 + t_{max} = 5$. On the other hand, with k_3 all three conditions for signature generation are met, so the signer can move forward with the index k_3 .

Note that, in the 3-NAF form, any two non-zero digits must be separated by at least two zeros, so the maximum number of non-zero digits is limited by $\lceil \frac{n+1}{z} \rceil = \lceil \frac{8+1}{3} \rceil = 3$. Because of this, for the particular parameters chosen for this example, it is impossible to have more non-zero digits than $L_1 = 4$.

After choosing a suitable index, the signer removes $z - 1 = 2$ zeros at the left of each non-zero digit and obtains

$$m'_{grouped} = (1, \bar{1}, 0\bar{1})$$

The signer then calculates

$$m'' = (0, 0, 3, 7)$$

using the given map (note: since $m'_{grouped}$ only has three blocks, the leftmost value of m'' is zero) as well as the checksum and the checksum blocks

$$c = \sum_{i=0}^{L_1-1} len - 1 - D''_i = (50)_{10} = (00110010)_2 \rightarrow (B_1, B_0) = (3, 2)$$

Finally, the signer calculates the signature

$$\sigma = (k_3, (\sigma_m)_3, (\sigma_m)_2, (\sigma_m)_1, (\sigma_m)_0, (\sigma_c)_1, (\sigma_c)_0)$$

where

$$\begin{cases} (\sigma_m)_3 = f^{15-0}((x_m)_3) = f^{15}((x_m)_3) \\ (\sigma_m)_2 = f^{15-0}((x_m)_2) = f^{15}((x_m)_2) \\ (\sigma_m)_1 = f^{15-3}((x_m)_1) = f^{12}((x_m)_1) \\ (\sigma_m)_0 = f^{15-7}((x_m)_0) = f^8((x_m)_0) \\ (\sigma_c)_1 = f^{2^4-1-3}((x_c)_1) = f^{12}((x_c)_1) \\ (\sigma_c)_0 = f^{2^4-1-2}((x_c)_0) = f^{13}((x_c)_0) \end{cases}$$

5.6.4 Signature verification

Given the message m and the signature $\sigma = (k_3, (\sigma_m)_3, (\sigma_m)_2, (\sigma_m)_1, (\sigma_m)_0, (\sigma_c)_1, (\sigma_c)_0)$, the verifier proceeds with the same calculations that the signer did for m'' and get $m'' = (0, 0, 3, 7)$. He also calculates the checksum blocks the same way the signer did and get $(B_1, B_0) = (3, 2)$. Then, he calculates

$$y' = g((y_m)_3' || (y_m)_2' || (y_m)_1' || (y_m)_0' || (y_c)_1' || (y_c)_0')$$

where

$$\begin{cases} (y_m)_3' = f^0((\sigma_m)_3) \\ (y_m)_2' = f^0((\sigma_m)_2) \\ (y_m)_1' = f^3((\sigma_m)_1) \\ (y_m)_0' = f^7((\sigma_m)_0) \\ (y_c)_1' = f^3((\sigma_c)_1) \\ (y_c)_0' = f^2((\sigma_c)_0) \end{cases}$$

If $y' = y$ then the signer accepts the signature as valid; otherwise he rejects the signature as invalid.

6 ANALYSIS OF THE Z-NAF OTS SCHEME

In this chapter we provide an analysis of the z-NAF OTS scheme for its security and performance. Then, we present the performance results of our proof-of-concept for z-NAF OTS and compare its performance to that of W-OTS.

6.1 Security Analysis

Since both W-OTS and z-NAF OTS uses the same hash chain structure with checksum, the proof of security of W-OTS also applies to z-NAF OTS, thus z-NAF OTS has the same security of W-OTS in that an attacker cannot forge new signatures by attacking the hash chains or the checksum. Furthermore, one could also trivially replace f in the z-NAF OTS by a chaining function in the same way W-OTS+ does for W-OTS (which results in "z-NAF OTS+"), as well as replace g and the concatenation by an L-tree, thus improve security by only requiring pre-image resistant one-way functions instead of collision resistance ones.

The difference between z-NAF OTS and W-OTS (and also w-OTS+, if the chaining function and the L-tree are employed in z-NAF OTS) is, given the message to be signed, the way the signer maps the message to the number of calls to f in each hash chain. While W-OTS divides a message m into w -bit blocks and maps them directly to the hash chains, z-NAF OTS employs the z-NAF form to first convert a message m into m'' , then map m'' to the hash chains in such a way that the number of calls to f in signature verification is significantly smaller than the number of calls in signature generation. In this way, what remains to be proven is that the mapping scheme of z-NAF OTS from m to m'' is secure.

For the following discussion, please refer to Figure 6 for the four steps in converting m to m'' . We show that an attacker fails to break each one of the four steps, so the conversion is secure and, thus, z-NAF OTS is secure.

Consider the first step, $m' = h(m||k)$ (a randomized hash is applied to the message

m concatenated to a nonce k). As with all hashing operations, this operation is not necessarily injective, however, by using a second pre-image resistant hash function, the probability of finding another pair (m_2, k_2) with $m_2 \neq m$ such that $h(m||k) = h(m_2||k_2)$ is negligible.

In the second step, m' is converted to the z-NAF form. From the properties of the z-NAF encoding (see Subsection 2.2.2), the z-NAF representation exists and is unique for all integers, so an attacker has no hope finding $m'_2 \neq m'$ such that $m'_{z-NAF} = (m'_2)_{z-NAF}$.

In the third step, the mandatory zeros on the left of each non-zero digit are removed and the remaining zeros are grouped as zero runs with one non-zero digit on the right of each zero run. This is a bijective operation for two reasons:

- The properties of the z-NAF encoding ensures that any two non-zero digits are necessarily separated by at least $z-1$ zeros. This fact, together with the requirement that the rightmost digit of m'_{z-NAF} must not be zero, ensures that there are always $z-1$ zero digits on the left of each non-zero digit that can be removed (this is valid even for the leftmost non-zero digit, when one considers that padding an integer with zeros on the left, whether or not in the z-NAF form, does not change its value);
- As a result of the previous statement, the previous m'_{z-NAF} can always be reconstructed by adding back $z-1$ zeros to the left of each non-zero digit and then truncating the sequence if necessary by throwing out the leftmost zeros until the whole sequence has exactly $n+1$ digits.

Therefore, an attacker cannot find a different m'_{z-NAF} having the same $m'_{grouped}$.

Finally, in the fourth step each block $0^{t_i}N_i$ of $m'_{grouped}$ is mapped to a non-negative integer digit D''_i in the interval $[0, (1+t_{max})2^{z-1})$ according to the map

$$D''_i = \begin{cases} t_i 2^{z-1} + (N_i - 1)/2 & \text{if } N_i > 0 \text{ and } i \leq r \\ t_i 2^{z-1} + (2^z + N_i - 1)/2 & \text{if } N_i < 0 \text{ and } i \leq r \\ 0 & \text{if } i > r \end{cases}$$

The first two equations of the mapping is intentionally chosen to be bijective in order to prevent any ambiguities that would allow an attacker to forge valid signatures. To see that, note that the rightmost $z-1$ bits of D''_i are only affected by N_i while the remaining bits are only affected by t_i . In this way, every different block $0^{t_i}N_i$ will be mapped to a

different D_i'' . For example, for $z = 2$ the mapping would be

$$\left\{ \begin{array}{l} 1 \rightarrow 0 \\ \bar{1} \rightarrow 1 \\ 01 \rightarrow 2 \\ 0\bar{1} \rightarrow 3 \\ \vdots \end{array} \right.$$

and for $z = 3$ the mapping would be

$$\left\{ \begin{array}{l} 1 \rightarrow 0 \\ 3 \rightarrow 1 \\ \bar{3} \rightarrow 2 \\ \bar{1} \rightarrow 3 \\ 01 \rightarrow 4 \\ 03 \rightarrow 5 \\ 0\bar{3} \rightarrow 6 \\ 0\bar{1} \rightarrow 7 \\ \vdots \end{array} \right.$$

One could observe that, when $N_i = 1$ and $t_i = 0$, we have $D_i'' = 0$, which is the same result when applying the third equation, and wonder if that coincidence would constitute a security breach. In reality, this is not a problem, because a different message resulting in the same m'' where the last digits come from the first equation instead of the third would require more digits in m'_{z-NAF} (after all, a zero in m'' coming from the first equation means an extra 1 in $m'_{grouped}$), which means that m' , the output of $h(m||k)$, would need to be longer than n bits. This clearly cannot occur, since, by definition, the output of h cannot be longer than n bits. Therefore, no attacker could explore that coincidence and come up with a different m' which would result in the same m'' .

Thus, since all steps in the conversion from m to m'' are secure, as are the hash chain structure and the checksum, we have shown that z-NAF OTS is secure as long as the used hash functions are secure.

6.2 Performance Analysis

In this section we analyze the performance of the z-NAF OTS scheme and compare it to the W-OTS(+) scheme. Both W-OTS and W-OTS+ have the same performance when the processing cost is measured as the number of one-way function (or chaining function) evaluations. The signature size is also equivalent in terms of the number of b -bit values in each signature; the difference is that W-OTS+ can use smaller values of b while maintaining the same security level as W-OTS. Therefore, to avoid confusion and for the sake of simplicity, the following analysis is done using W-OTS, although the results are equally applicable to W-OTS+.

It is well known that, for W-OTS, the signature size and processing time forms an exponential trade-off regulated by the Winternitz parameter w . More specifically, the signature size is given by $|\sigma|_{W-OTS} = n/w$, whereas the total number of one-way function operations in key generation process is given by $\#(f)_{W-OTS} = \lceil \frac{n}{w} \rceil (2^w - 1)$. These equations are valid when ignoring the checksum, which normally represents less than 5% of the total signature size and processing overhead. However, because w can only be positive integers, both the signature size and the processing time changes in large steps when w changes, leaving large gaps in between. Hence, to compare W-OTS with the hereby proposed z-NAF OTS scheme, we use an interpolated model for W-OTS, thus filling in the signature size and processing time gaps.

To accomplish this, we note that the equations for $|\sigma|_{W-OTS}$ and $\#(f)_{W-OTS}$ can be combined and $\#(f)_{W-OTS}$ can be written as a function of $|\sigma|_{W-OTS}$ in the following way:

$$\#(f)_{W-OTS} = \lceil |\sigma|_{W-OTS} \rceil (2^{n/|\sigma|_{W-OTS}} - 1) \quad (6.1)$$

The curve formed by equation 6.1 contains the points for $w = 1, 2, 3, \dots$ as well as the interpolated values in between. The curve is, thus, useful as basis for comparison against other hash-based OTS schemes whose signature sizes or total number of hashes do not necessarily match those points for integer w values, as is the case of the z-NAF OTS scheme. Figure 9 shows the points for $w = 1, 2, \dots, 7, 8$ overlaid on the continuous curve obtained when plotting equation 6.1 over $\mathbb{R}$.

Another important aspect of performance analysis is the choice of z . The parameter z is the main parameter in z-NAF OTS since it controls the signature size and performance trade-offs (similarly to the Winternitz parameter in W-OTS). It affects all three algorithms of key generation, signature generation and signature verification, as well as determines

Figure 9 – The W-OTS total number of one-way function evaluations as a function of signature size. The small squares are the curve points for $w = 1, 2, \dots, 8$ while the line is the curve generated when plotting the equation 6.1 over $\mathbb{R}$.

Source: Author

the choice of L_1 , t_{max} and w . A large value of z results in a small signature size, but the key generation, signature generation and signature verification algorithms should take longer to execute. Conversely, choosing a small z speeds up the signature algorithms, but the signature grows larger.

With these concepts in mind, we proceed to analyze the signature size and processing time of the z -NAF OTS and compare them to those of W-OTS.

6.2.1 Signature Size

The signature size of z -NAF OTS, in terms of the number of b -bit values in the signature, is given by

$$|\sigma|_{z-NAFOTS} = L_1 + L_2$$

where L_1 is the number of hash chains used to sign m'' and L_2 the number of hash chains used to sign the checksum (the length of the nonce k was not included in the analysis because it is generally much smaller than b). In the same way as W-OTS, since the checksum part of the signature is generally less than 5% of the total checksum size, we

can simplify the analysis and adopt the approximation

$$|\sigma|_{z-NAFOTS} \approx L_1$$

Before discussing how one should pick a number for the parameter L_1 , recall from Section 2.2.2 that any n -bit integer only needs $n + 1$ z -NAF digits to represent it, and that the non-zero digit density of the z -NAF form is given by $1/(z + 1)$. This means that there will be on average $(n + 1)/(z + 1)$ non-zero z -NAF digits in a n -bit integer, thus the majority of the n -bit integers will have this number of non-zero digits. Figure 10 shows the distribution of integers according to their number of non-zero digits for $z = 2$ and $z = 3$ when $n = 256$.

Figure 10 – The distribution of 256-bit integers as a function of their number of non-zero z -NAF digits for $z = 2$ and $z = 3$. One million random samples were used for each z value in the simulations.

Source: Author

Considering the distribution from Figure 10, the parameter L_1 is chosen in the algorithm set up phase according to the chosen value of z . This, together with the parameter t_{max} , allows for many different trade-offs between signature size and processing overhead.

In order to determine an adequate value of L_1 for signature size, we recall that the maximum number of non-zero digits $r \leq \lceil \frac{n+1}{z} \rceil$, since there must be at least $z - 1$ zeros between any non-zero digit. Therefore, choosing values of $L_1 > \lceil \frac{n+1}{z} \rceil$ will only increase

the signature size without any added benefit. On the other hand, choosing L_1 much smaller than $(n+1)/(z+1)$ would make one spend much more time finding a suitable k than performing the actual evaluations of f , which could make the signature generation process impractical. Ideally, thus, one should minimize the value of L_1 to reduce signature size while keeping the signature generation time manageable.

6.2.2 Processing Time

we analyze the processing time for the three algorithms comprised in z-NAF OTS: key generation, signature generation and signature verification. Each of these processing times can be expressed in terms of the number of evaluations of f , g and h , the number of PRF calls and the number of z-NAF conversions.

Normally, the number of evaluations of f is the dominant factor in the key generation and signature verification times, while the signature generation time is dominated by the number of evaluations of f , the number of evaluations of h (when calculating $m' = h(m||k)$) as well as the number of z-NAF conversions. Note that all three algorithms are sensitive to the value of z : a larger z makes all three algorithms slower, whereas a small z has the opposite effect.

Each algorithm is analyzed separately in what follows.

6.2.2.1 Key generation

In this algorithm, given x , first there are $L_1 + L_2$ calls to the PRF function to generate all the $(x_m)_i$ and $(x_c)_i$ secret values. Then, the function f is applied $len - 1 = (1 + t_{max})2^{z-1} - 1$ times for each $(x_m)_i$ to generate the corresponding $(y_m)_i$ and $2^w - 1$ times for each $(x_c)_i$ to generate the corresponding $(y_c)_i$, totalling $L_1((1 + t_{max})2^{z-1} - 1) + L_2(2^w - 1)$ evaluations of f . Finally, g is called once on $(y_m)_{L_1-1} || \dots || (y_m)_0 || (y_c)_{L_2-1} || \dots || (y_c)_0$ to generate y .

From this description, it is easy to see that the key generation time increases exponentially with z and linearly with L_1 and t_{max} .

6.2.2.2 Signature generation

In the signature generation algorithm, first one needs to generate a number of nonces k_i in order to calculate $m' = h(m||k_i)$ and m'_{z-NAF} . In this way, one needs to evaluate

h and do the z-NAF conversion $1/p$ times on average, where p is the probability of finding m'_{z-NAF} satisfying the three conditions described in Section 5.3. Then, after finding a value of k_i where m'_{z-NAF} satisfies the conditions and calculating m'' , one needs to calculate the hash chains by evaluating f repeatedly and publish the signature as described in Section 5.3.

As with the key generation algorithm, signature generation time increases exponentially with z . The parameters L_1 and t_{max} , however, have a different effect in signature generation. Namely, large values of L_1 and t_{max} increase the probability p of finding a suitable k_i and reduce the number of evaluations of h , as well as the number of z-NAF conversions required. At the same time, however, it increases the number and length of hash chains that needs to be calculated. Hence, it increases the number of evaluations of f . A small value of L_1 has the opposite effect. Because of that, one needs to choose values of L_1 and t_{max} that minimizes the total amount of evaluations of h , f and z-NAF conversions.

6.2.2.3 Signature verification

In the signature verification algorithm, one needs to perform one evaluation of h and one z-NAF conversion to calculate m'' from the message and the index k provided in the signature. Then, one "completes" the hash chains by evaluating f repeatedly and recovers the top chain values y'_i . Finally, one performs a single evaluation of g to recover y' , which is compared against the public key y to validate the signature.

Similarly to the key generation and signature generation algorithms, signature verification time increases exponentially with z , but the behaviour with different values of L_1 and t_{max} is more complex.

Originally, before filtering by L_1 and t_{max} , the average m'_{z-NAF} has non-zero digit density of $1/(z+1)$ and a zero digit density of $z/(z+1)$. So, after removing the mandatory $z-1$ zeros at the left of each non-zero digit, one would have a zero density of about $z/(z+1) - (z-1)/(z+1) = 1/(z+1)$, which is the same density as the number of non-zero digits. This means that the number of zero and non-zero digits in $m'_{grouped}$ is the same on average.

However, because L_1 limits the number of non-zero digits in m_{z-NAF} , a value of L_1 too small would select only those m_{z-NAF} with more zeros than non-zero digits. Due to the map defined in Section 5.3, though, each non-zero digit increases the total number of evaluations of f in the verification algorithm by some value in $[0, 2^{z-1})$, while each zero

digit increases it by exactly 2^{z-1} .

Therefore, making L_1 too small reduces the number of hash chains that need to be calculated, but also increases the individual digits in m'' . This potentially increases the overall number of evaluations of f , making verification slower. Choosing L_1 too large removes this digit imbalance, but increases the number of hash chains and slows down the verification process.

The effects of parameter t_{max} is even more interesting: although choosing a small t_{max} decreases the number of evaluations of f during signature verification, making t_{max} very large does not significantly increase it. That happens because, even when very long runs of zeros are allowed in m'_{z-NAF} , the probability of occurrence of that zero run decreases exponentially by P_Z with each additional zero digit, where $P_Z < 1$ is the probability of a z-NAF digit not be zero (empirically, it is about $1/2$), while the total number of hashes only increases linearly by 2^{z-1} . This means that the number of calls to f in signature verification is bounded with t_{max} . For that reason, in order to optimize signature size and signature verification time, one could use smaller values of L_1 and compensate the increased signature generation time with larger values of t_{max} , which doesn't increase the verification time by much.

Besides reducing the total chain length len , there is another reason why a small t_{max} can make verification faster: while L_1 limits the number of non-zero digits in m'_{z-NAF} , t_{max} tends to limit the number of zero digits. In this way, a small t_{max} decreases the amount of zero digits relative to the amount of non-zero digits in m'_{z-NAF} . According to the effects of the balance of zero and non-zero digits discussed before, this reduces the digits in m'' and requires less evaluations of f for signature verification.

Finally, we note that, differently from the W-OTS scheme, because long zero runs at the left of each non-zero digit is exponentially less probable, the digits in m'' are skewed towards zero. The result is that the number of evaluations of f in signature verification is significantly lower than half of the total evaluations of f in the key generation algorithm. This provides the increased verification speed at the cost of decreased signature generation speed, compared to the W-OTS scheme.

6.2.3 Signature size - processing time trade-off

In the z-NAF OTS scheme, the trade-off between signature size and processing time can be made by choosing the appropriate values for z , L_1 and t_{max} . The underlying

Winternitz parameter w can then be chosen such that the checksum hash chains have roughly the same length as the message hash chains.

The most important parameter among those is z , since it has the strongest influence over the signature size and processing time of the scheme. Actually, z is somewhat similar to the Winternitz parameter in W-OTS, regulating an exponential trade-off between signature size and processing time.

To understand that, recall that the signature size is basically determined by L_1 , which is generally chosen as being proportional to z : according to the analysis in subsection 6.2.2, generally L_1 is chosen to be somewhere around the average number of non-zero digits in m'_{z-NAF} , which is given by $(n+1)/(z+1)$. The total chain length len , however, is given by $len = (1 + t_{max})2^{z-1} - 1$ and thus grows exponentially with z . In this way, the total number of evaluations of f grows exponentially with z , but the signature size only decreases linearly when z increases. This behaviour is similar to the effect of the w parameter in W-OTS, where the signature size is given by $\lceil n/w \rceil$ and the total chain length is given by $2^w - 1$.

While the parameter z determines the main trade-off between signature size and processing time, the parameters L_1 and t_{max} allows one to fine-tune that trade-off. Nevertheless, these parameters should be chosen carefully, as inadequate choices can lead inefficiencies.

Normally, one would want to pick values for them that are as small as possible while keeping signature generation and signature verification times manageable, because this would help keep signature size small and key generation time short, while still making signature verification significantly faster than signature generation. In some circumstances, it is even possible to trade a smaller value of L_1 for a slightly larger value of t_{max} , which, as explained in Subsection 6.2.2.3, allows for smaller signature sizes without increasing much the signature verification time.

With the formulas for signature size and key generation time for both z-NAF OTS and W-OTS, we can compare the trade-off in both algorithms. Keeping the signature size the same and assuming that one chooses $L_1 = \lceil (n+1)/(z+1) \rceil$, one could easily realize that the parameter z corresponds to the parameter $w - 1$ for the usual values of n (such as $n = 256$). For example, $z = 3$ would produce the same signature size in z-NAF OTS than $w = 4$ would in W-OTS. Substituting

$$w = z + 1$$

in the formulas for W-OTS, one would have the total number of evaluations of f in the key generation as

$$\begin{cases} \#(f)_{\text{z-NAF OTS}} = \left\lceil \frac{n+1}{z+1} \right\rceil ((1 + t_{\max})2^{z-1} - 1) \\ \#(f)_{\text{W-OTS}} = \left\lceil \frac{n}{z+1} \right\rceil (2^{z+1} - 1) \end{cases}$$

As z tends to infinity, the ratio $\#(f)_{\text{z-NAF OTS}}/\#(f)_{\text{W-OTS}}$ would be basically given by

$$\lim_{z \rightarrow \infty} \frac{\#(f)_{\text{z-NAF OTS}}}{\#(f)_{\text{W-OTS}}} = \lim_{z \rightarrow \infty} \frac{\left\lceil \frac{n+1}{z+1} \right\rceil ((1 + t_{\max})2^{z-1} - 1)}{\left\lceil \frac{n}{z+1} \right\rceil (2^{z+1} - 1)} = \frac{1 + t_{\max}}{4}$$

This shows that, if $t_{\max}$ can be chosen to be smaller than 3 for large z , then the z-NAF OTS should have better asymptotic behaviour than W-OTS. Indeed, that is the case: since $t_{\max}$ represents the maximum length of zero runs adjacent to non-zero digits in m'_{grouped} , in the extreme when z is as large as possible (e.g. for $n = 256$, it happens when $z = 256$), there is only one hash chain, so one could trivially choose $t_{\max} = 0$. This demonstrates that the z-NAF OTS has better asymptotic behaviour than W-OTS. When z is large, the total number of f evaluations in key generation (which, for both schemes, is the total number of evaluations of f in signature generation plus the total number of evaluations of f in signature verification) is up to four times larger in W-OTS than in z-NAF OTS.

These considerations are illustrated in Figures 11, 12 and 13. Namely, Figure 11 shows the comparison between z-NAF OTS and W-OTS as far as the calls to f in key generation is concerned. Figures 12 and 13 show, respectively, zoomed-in visions of the lower and upper regions of the graph in Figure 11.

6.2.4 Storage Requirements

The storage requirements of the z-NAF OTS is similar to that of W-OTS, namely, the public and private keys for the signer and the public key for the verifier. In the z-NAF OTS scheme, the private key as well as the public key are b bits long, so the signer needs to store $2b$ bits while the verifier needs to store only b bits.

Figure 11 – Asymptotic behaviour of key generation time of z-NAF OTS compared to W-OTS (these are theoretical values, not simulation results). The vertical axis is displayed in logarithmic scale. When zoomed out, it looks like they both have similar performance as far as the total number of calls to f in key generation is concerned. Still, it can be realized that W-OTS should have faster key generation than z-NAF OTS when z is small, whereas z-NAF OTS should be faster when z is large.

Source: Author

6.3 Proof of Concept

To test the z-NAF OTS, a proof of concept was written in C language using the OpenSSL library for random number generator and hash functions and standard libraries for everything else. The security parameter b and the message length n were both chosen to be 256 bits, while the functions f , g and h were all implemented using SHA256 hash functions.

For this proof of concept, no special hardware optimizations were used, and the code was compiled with GCC for the x86-64 platform without any special compiler flags. All simulations were run on a lightweight version of Ubuntu 16.04 (called Lubuntu 16.04) with kernel 4.15.0-39-generic, using a desktop computer with the Intel Core i5 750 CPU and 6GB RAM.

We note that, in an actual implementation, besides calculating all hash chains in parallel, one could also evaluate all calls to h and all z-NAF conversions in parallel when searching for a suitable k_i in signature generation process. That would allow smaller

Figure 12 – Zoomed-in view of the lower part of the graph in Figure 11. For small values of z , the average number of non-zero digits in m'_{z-NAF} is higher. Hence, t_{max} must be kept relatively large (4 to 5) so that signature generation is not too slow. In this region, the z-NAF OTS has about 25% to 50% more calls to f than W-OTS.

Source: Author

Figure 13 – Zoomed-in view of the upper part of the graph in Figure 11. When z is large, small values of t_{max} can be used (as small as zero in the extreme case), so z-NAF OTS becomes faster than W-OTS. In this region of the graph, the total number of evaluations of f in key generation is four times larger in W-OTS than in z-NAF OTS.

Source: Author

values of L_1 and t_{max} to be used and thus smaller signature sizes and possibly faster signature verification.

Besides the z-NAF OTS proof of concept, a proof of concept of W-OTS was implemented using the same language, same libraries, same parameters (where applicable), same optimization level and the simulations were done on the same environment and same machine, in order to provide useful numeric comparisons.

The simulations were done for many different values of z and involved two phases: the goal of the first phase was to understand quantitatively how the parameters L_1 and t_{max} impact the key generation, signature generation and signature verification times. In the second phase, what was believed to be the best choices of parameters from phase one were chosen and used for the actual simulations. The results, such as the number of hash function evaluations as well as the CPU cycles taken by each algorithm, were recorded for evaluation and were used to choose optimized parameters compared to those obtained in phase one. Finally, the results from the z-NAF OTS proof of concept using the optimized parameters were compared to those from the W-OTS implementation, in order to estimate the practical advantages and disadvantages of the z-NAF OTS.

For the experiments, we used the W-OTS interpolated model to compare the number of calls to f between W-OTS and z-NAF OTS when the latter had a signature size not present in W-OTS, because all W-OTS signature sizes are given by $\lceil n/w \rceil$.

6.3.1 Searching for appropriate values for L_1 and t_{max}

To get average values of the processing times for key generation, signature generation and signature verification, we had to run each algorithm with each parameter set many times and then calculate the average of those values. As it was not practical so run them for every combination of z , L_1 and t_{max} , we had to restrict ourselves to a subset of the possible combinations and run the algorithm using them.

The choices for the parameter z were easy to pick. Since we knew the relation $z \approx w - 1$ for signature size equivalence between z-NAF OTS and W-OTS, we did not need to run the three algorithms for very large values of z , as those might never get used in practice (the largest value of w being used in practice in modern W-OTS(+) implementations is 4, which yields chain lengths of 15). Because of that, we restricted our simulations to values of z such that $2 \leq z \leq 8$.

The different parameters for L_1 and t_{max} , however, as we discussed above, have a

more complex behaviour in the algorithms. For example, the distribution of non-zero digits certainly restrict the values that are practical for L_1 , while the distribution of zero runs dictates how one should choose t_{max} . In order to define which values of L_1 and t_{max} should be used, we first did a preliminary statistical analysis of the messages in the z-NAF format to better understand the distributions of digits in them.

To do that, a Python script was written to generate (pseudo-)random 256-bit integers, convert them to the z-NAF format, and then record the distribution of non-zero digits in them as well as that of zero runs of different lengths. One million integers were generated (pseudo-)randomly for each value of z . Figures 14 and 15 shows the distribution of non-zero digit counts and zero-runs respectively, while Table 2 contains the average values of each of the two distributions.

Figure 14 – Distribution of non-zero digit count for different values of z . One million samples were used for each value of z . It looks like the non-zero digit counts follow normal distributions.

Source: Author

The results of the simulations show that the average number of non-zero digits conforms to our calculations, and that the average length of zero runs is actually pretty close to 1 and does not change with z . Table 3 is a cumulative sum of the probabilities of finding a zero-run with a certain length. This is the same as the probability of finding a zero-run up to a certain length, and is useful to determine the values that should be used for t_{max} .

Figure 15 – Distribution of zero-runs for different values of z . One million samples were used for each value of z . Note that, generally, one sample contains more than one zero run. The distributions here are likely exponential distributions. The actual zero-run count reduces with z , because there are generally fewer zero runs in shorter digit sequences.

Source: Author

Table 2 – Average number of non-zero digits in a message, as well as the average length of a zero run in the messages. The results in this table confirms the calculations that we had for the number of non-zero digits for the z -NAF form, that is, $(n+1)/(z+1)$. Surprisingly, the average zero run length does not change with z , which means that their distributions are also the same regardless of the value of z .

z	Average number of non-zero digits	Average zero run length
2	86.44	0.99
3	65.19	0.99
4	52.44	0.99
5	43.95	0.99
6	37.88	0.99
7	33.33	0.99
8	29.79	0.99

Source: Author

Table 3 – Probability of finding a zero run up to a certain length for $z = 2$. As noted before, this probability does not change with z . From the table, and because the distribution is exponential, the table can be summarized by the equation $P(\text{zero-run length} < i) = 1 - 1/2^{i+1}$.

Maximum zero-run length for one non-zero digit	Probability
0	50.19%
1	75.19%
2	87.65%
3	93.85%
4	96.94%
5	98.47%
6	99.24%
7	99.62%
8	99.81%

Source: Author

Another Python script was written in order to investigate the distribution of zero and non-zero digits in the z-NAF form. That script also generated one million (pseudo-)random integers, calculated their z-NAF form and counted the number of occurrences of each digit. Table 4 shows the average percentage of zero and non-zero digits in each message after the mandatory zeros at the left of each non-zero digit were removed.

Table 4 – Average number of zero and non-zero digits in a message, shown as a percentage of the total number of digits. Here we have another interesting fact: after the removal of the mandatory zeros, the remaining zero digits occur as frequently as all non-zero digits added together. This also confirms our earlier calculations that the density of both zero and non-zero digits after the removal of mandatory zeros is given by $1/(z + 1)$ and, thus, they occur with the same probability.

z	Average zero digit count	Average non-zero digit count
2	50.00%	50.00%
3	50.00%	50.00%
4	50.01%	49.99%
5	49.99%	50.00%
6	50.01%	49.99%
7	49.99%	50.01%
8	50.01%	49.99%

Source: Author

The simulations has shown that, after the removal of the mandatory zeros at the left of each non-zero digit, the remaining zero digits occur with the same frequency as all the non-zero digits added together, which confirms our previous calculations that the remaining zero digits would have the same density as the non-zero digits. However, as discussed previously in Section 4.1.2, this result is only valid before we filter out the

messages by L_1 and t_{max} , or if we do not choose too small values for L_1 and t_{max} .

A similar simulation was performed to find out the relative frequencies of non-zero digits among themselves and, as one would suspect, it turned out they all occur with similar frequencies. The tables that show those frequencies, however, are too large to be included when z is large (for $z = 8$ one would have $2^{8-1} = 128$ different non-zero digits).

With this information, it would be natural to try values for L_1 that are around the average number of non-zero digits for a given z , and also try values of t_{max} such that the probability of all digits having a zero-run of no more than t_{max} , that is,

$$P(\text{all zero-run length} \leq t_{max}) = L_1^{P(\text{single zero-run length} < t_{max})}$$

is not too small. Table 5 shows the values chosen for L_1 and t_{max} for each value of z , together with the estimated probability of occurrence of the message due to the t_{max} restriction. The Winternitz parameter w for the checksum was chosen according to the signature size equivalence equation $w = z + 1$.

Table 5 – Initial parameters chosen to run the three algorithms: key generation, signature generation and signature verification.

z	L_1	t_{max}	w
2	86	5	3
3	64	5	4
4	52	5	5
5	43	4	6
6	37	4	7
7	32	4	8
8	29	4	9

Source: Author

We note that the overall probability of finding a suitable message is a combination of the following probabilities:

- The probability that the first digit is not zero (which is $1/2$, because odd integers would have the first z-NAF digit non-zero while even integers would have the first digit zero, and half of all n -bit integers is odd while the other half is even);
- The probability that the total number of non-zero digits in m'_{z-NAF} does not exceed the value of L_1 ;
- The probability that no zero-run in $m'_{grouped}$ is longer than t_{max} .

Since the aforementioned probabilities are not independent variables, calculating the overall probability is not trivial. However, the individual probabilities give us a good idea of what parameters should be chosen in order to make signature generation a manageable task.

With these parameters in hands, the next phase would be to run the three algorithms with them and, if necessary, refine the choice of the parameters to minimize the signature size or the processing time required.

6.3.2 Simulations

Before running the actual simulations, a benchmark was done for the most time-consuming operations in the three algorithms. Table 6 summarizes the average number of machine cycles needed for the generation of (pseudo-)random 256-bit strings, h and f . In addition, Table 7 summarizes the average machine cycles needed for h , partial z-NAF conversion, full z-NAF conversion and g for different values of z , according to the values of L_1 and t_{max} given in Table 5. Partial z-NAF conversions are performed when searching for a suitable k_i , as the conversion stops as soon as the result fails to meet one of the requirements (first digit non-zero, L_1 restriction and t_{max} restriction), while full z-NAF conversions are done during signature verification, when the index k is already given. Our simulations show that partial conversions take significantly less time than full conversions.

Note that the number of cycles taken by the (pseudo-)random string generation does not change with z , L_1 or t_{max} .

Table 6 – Average CPU cycles for (pseudo-)random generation of 256-bit strings, single h evaluation and single f evaluation.

Operation	CPU cycles
Generation of (pseudo-)random 256-bit string	3424
Single evaluation of h	854
Single evaluation of f	851

Source: Author

After the benchmarks, the three algorithms key generation, signature generation and signature verification were executed ten thousand times according to the initial parameters set in Table 5 and the results were averaged. The average values are summarized in Table 8 and presented in Figure 16 and Figure 17.

As expected, the calls to f and the total machine cycles of all three algorithms grow

Table 7 – Average CPU cycles for partial z-NAF conversion, full z-NAF conversion and single g evaluation for different choices of z according to the parameters given in Table 5. Note that all CPU cycles decrease when z increases, because a larger z results in larger and fewer z-NAF conversion blocks as well as fewer values to concatenate when calculating g .

z	Partial z-NAF conversion	Full z-NAF conversion	g
2	5808	20382	30823
3	5058	15910	22865
4	4579	13003	18878
5	3148	11092	16187
6	3144	10093	13560
7	2987	9246	12259
8	2530	7864	10921

Source: Author

Table 8 – Results using the preliminary simulations from Table 5. *The calls to h refer to the average number of calls performed in the signature generation algorithm until a suitable value of k_i is found; its standard deviation is in the same order as its value and was omitted. **The calls to f in the table does not include those performed during the calculation of the checksum chains.

z	Calls to h^*	Calls to f^{**}			Total CPU cycles ($\times 10^3$)		
		KG	Sign	Ver	KG	Sign	Ver
2	28	946	726 ± 8	220 ± 8	943 ± 22	869 ± 148	254 ± 12
3	21	1472	1103 ± 16	369 ± 16	1388 ± 54	1123 ± 108	384 ± 21
4	10	2444	1839 ± 34	605 ± 34	2245 ± 111	1701 ± 90	601 ± 40
5	33	3397	2364 ± 55	1033 ± 55	3115 ± 116	2217 ± 140	993 ± 66
6	21	5883	4110 ± 110	1773 ± 110	5288 ± 100	3704 ± 139	1670 ± 104
7	29	10208	7451 ± 303	2757 ± 303	9245 ± 190	6626 ± 319	2674 ± 283
8	10	18531	15494 ± 365	3037 ± 365	16765 ± 460	13723 ± 455	3070 ± 357

Source: Author

Figure 16 – Plot of the preliminary results for the average number of calls to f of each algorithm, for various values of z as given in Table 8.

Source: Author

Figure 17 – Plot of the preliminary results for the total CPU cycles spent for each algorithm (average values), for various values of z as given in Table 8.

Source: Author

exponentially with z , a behaviour similar to what the parameter w does for W-OTS(+). However, the time taken for the signature verification algorithm is significantly less than that taken for the signature generation and grows more slowly, so the asymmetric effect of the scheme is clearly shown.

To refine the parameters given in Table 5, a search was done by increasing and decreasing L_1 and t_{max} around the preliminary values for various choices of z , while comparing both the number of calls to f and the total machine cycles resulting from the new parameters to those of W-OTS using the interpolated model developed previously. In this process we found that, generally, by reducing L_1 slightly and increasing t_{max} to compensate, we can get better performance compared to the initial values. The new parameters that we settled on are shown in Table 9.

Table 9 – Optimized parameters to run the three algorithms: key generation, signature generation and signature verification. These parameters were optimized for shorter signature verification times when compared to the equivalent W-OTS of the same signature size.

z	L_1	t_{max}	w
2	75	8	3
3	56	9	4
4	46	6	5
5	39	5	6
6	32	10	7
7	28	10	8
8	25	9	9

Source: Author

Note that the machine cycle numbers depend on the particular choices and implementations of h , f , g and the z-NAF conversion algorithm, therefore, depending on how one implements those functions, the optimal parameters may change. In this particular proof of concept, for demonstration purposes, a very simple and unoptimized version of the z-NAF conversion algorithm was implemented, so the CPU cycle results are suboptimal. In actual implementations, an optimized algorithm should give significantly better results for the machine cycles, specially in the case of the signature generation algorithm.

Using the optimized set of parameters given in Table 9, we run the same three algorithm again using the same procedures, and recorded the average results in Table 10. Figures 18 and 19 shows the new plots based on the new table.

There are a few interesting things worth noting. The first is that the optimized parameters had smaller values of L_1 , which caused the calls to h (and thus also the number of z-NAF conversions) to increase dramatically during signature generation compared to

Table 10 – Results using the optimized simulations from Table 9. Even though the absolute values for the number of calls to f and the CPU cycles increased compared to Table 8, the optimized results are actually more competitive against W-OTS because they use smaller values of L_1 and, thus, result in smaller signature sizes. *The calls to h refer to the average number of calls performed in the signature generation algorithm until a suitable value of k_i is found; its standard deviation is in the same order as its value and was omitted. **The calls to f in the table does not include those performed during the calculation of the checksum chains.

z	Calls to h^*	Calls to f^{**}			Total CPU cycles ($\times 10^3$)		
		KG	Sign	Ver	KG	Sign	Ver
2	489	1275	1020 ± 6	255 ± 6	1209 ± 49	4937 ± 4002	281 ± 14
3	1113	2184	1738 ± 12	446 ± 12	1972 ± 75	8853 ± 7261	439 ± 24
4	2562	2530	1776 ± 22	754 ± 22	2306 ± 75	14126 ± 12619	719 ± 32
5	5082	3705	2409 ± 38	1296 ± 38	3340 ± 54	19649 ± 17526	1192 ± 47
6	8865	11232	8916 ± 231	2316 ± 231	9844 ± 184	44888 ± 37343	2116 ± 205
7	17164	19684	16890 ± 340	2794 ± 340	17370 ± 452	79395 ± 66145	2686 ± 320
8	27332	31975	29077 ± 354	2898 ± 354	28188 ± 628	110103 ± 86662	2764 ± 344

Source: Author

Figure 18 – Plot of the optimized results for the average number of calls to f of each algorithm, for various values of z as given in Table 10.

Source: Author

Figure 19 – Plot of the optimized results for the total CPU cycles spent for each algorithm (average values), for various values of z as given in Table 10.

Source: Author

the initial parameters. This is evident from the total CPU cycles for signature generation, which increased dramatically and had almost the same dispersion pattern as the number of calls to h . We reiterate that most of the increase in CPU cycles are due to the inefficient z -NAF conversion algorithm and, with an optimized implementation of the algorithm, the increase would be much smaller.

The second thing is that, with the larger values of t_{max} , we also saw significant increase to the number of calls to f during key generation and signature generation, which contributed as well to increase the total CPU cycles of these algorithms.

Finally, the increase to the signature verification time was only modest; this fact, combined with the smaller values of L_1 used and thus the smaller signature sizes achieved, make the z -NAF OTS signature verification more competitive against W-OTS with the optimized parameters than with the initial parameters, as presented in the next subsection.

6.3.3 Performance comparison against W-OTS

In this subsection we compare z -NAF OTS to W-OTS in terms of the average number of calls to f and the total number of CPU cycles. We do the comparison using both the

initial parameters as well as the optimized parameters in z-NAF OTS. The goal is to show the improvements to the signature verification as well as the added overhead to the key generation and signature generation when switching from the initial parameters to the improved ones.

We strive to make the W-OTS implementation as comparable as possible to the z-NAF OTS implementation by using the same language, same libraries, same coding style and software optimizations, same compiler and compiler flags, same machine, and same operating system. Each of the three algorithms in W-OTS was run ten thousand times and the results were averaged and shown in table 11.

Table 11 – Signature size and average number of CPU cycles needed for each algorithm of W-OTS for many values of w when $n = 256$ bits. *Signature size is given as a multiple of 256-bit integers and does not include the checksum part. **The calls to f in the table does not include those performed during the calculation of the checksum chains.

w	Sig. Size*	Calls to f^{**}			Total CPU Cycles ($\times 10^3$)		
		KG	Sign	Ver	KG	Sign	Ver
1	256	256	128 ± 8	128 ± 8	550 ± 18	348 ± 13	211 ± 11
2	128	384	192 ± 13	192 ± 13	505 ± 18	289 ± 17	225 ± 16
3	86	602	303 ± 22	299 ± 22	650 ± 20	350 ± 23	307 ± 21
4	64	960	480 ± 37	480 ± 37	941 ± 24	481 ± 37	464 ± 38
5	52	1612	805 ± 66	807 ± 66	1526 ± 60	760 ± 63	767 ± 63
6	43	2709	1351 ± 119	1358 ± 119	2532 ± 139	1266 ± 125	1275 ± 125
7	37	4699	2345 ± 225	2354 ± 225	4262 ± 49	2120 ± 199	2155 ± 199
8	32	8160	4085 ± 421	4075 ± 421	7440 ± 165	3617 ± 368	3798 ± 368
9	28	14819	7422 ± 789	7397 ± 789	13523 ± 263	6679 ± 677	6861 ± 677

Source: Author

The comparison percentages between z-NAF OTS and W-OTS in the following subsections were calculated using the following formulas:

$$\text{Diff}_{\#(f)} = \frac{\#(f)_{z\text{-}NAFOTS} - \#(f)_{W\text{-}OTS}}{\#(f)_{W\text{-}OTS}} \times 100$$

$$\text{Diff}_{\text{CPU Cycles}} = \frac{\text{CPU Cycles}_{z\text{-}NAFOTS} - \text{CPU Cycles}_{W\text{-}OTS}}{\text{CPU Cycles}_{W\text{-}OTS}} \times 100$$

In other words, the results from W-OTS algorithms were used as the reference to compare the z-NAF OTS algorithms against. In this way, a positive percentage value means that the z-NAF OTS has more calls to f (or takes up more CPU cycles) than W-OTS, whereas a negative value signifies the opposite.

6.3.3.1 z-NAF OTS vs. W-OTS: initial parameters

For the initial parameters, no interpolation over W-OTS results was needed because the signature sizes chosen in the z-NAF OTS scheme matches those in the W-OTS scheme. The Table 12 summarizes the comparison between z-NAF OTS and W-OTS using the initial parameters for z-NAF OTS.

Table 12 – Comparison of the processing time (in terms of the number of calls to f and the number of CPU cycles used) between z-NAF OTS and W-OTS using the initial parameters for z-NAF OTS. *Signature size is given as a multiple of 256-bit integers and does not include the checksum part. **The calls to f in the table does not include those performed during the calculation of the checksum chains.

Sig. Size*	Calls to f^{**}			CPU Cycles		
	KeyGen	Sign	Ver	KeyGen	Sign	Ver
86	57.1%	140.0%	-26.5%	45.1%	148.1%	-17.3%
64	53.3%	129.6%	-23.1%	47.5%	133.2%	-17.2%
52	51.6%	128.6%	-25.1%	47.2%	123.8%	-21.7%
43	25.4%	75.0%	-23.9%	23.0%	75.1%	-22.1%
37	25.2%	75.2%	-24.7%	24.1%	74.7%	-22.5%
32	25.1%	82.4%	-32.3%	24.3%	83.2%	-29.6%
29	25.1%	108.8%	-58.9%	24.0%	105.5%	-55.3%

Source: Author

As can be seen in Table 12, both the key generation and the signature generation operations for z-NAF OTS require more processing time than for W-OTS, whereas the signature verification operation requires less processing time compared to W-OTS. Moreover, compared to W-OTS, the key generation algorithm requires from 25% to 50% more processing depending on the signature size chosen, while the signing operation requires around twice as much processing time. However, the verification time is reduced somewhere from 25% to 50% depending on the signature size chosen.

It is also interesting to observe the trends of the processing comparison as signature size decreases. For the chosen parameters, the processing in the signing operation remains more or less the same, while the signature verification of z-NAF OTS seems to decrease faster than that of W-OTS as signature size decreases, so smaller signature sizes should show even greater reductions for z-NAF OTS.

Also note that the improvements of z-NAF OTS over W-OTS are greater for the calls to f than for the CPU cycles. This happens because our z-NAF OTS implementation was not fully optimized, and because other calculations within the algorithms (such as

PRF calls and calls to g and h) are not accounted by the calls to f . Therefore, better implementations of our scheme, specially for the z-NAF conversion, should reduce the CPU overhead of z-NAF OTS and improve significantly the CPU cycle advantages during signature verification.

6.3.3.2 z-NAF OTS vs. W-OTS: optimized parameters

In the case of the optimized parameters, we needed to interpolate the W-OTS results since the W-OTS signature sizes do not match those of the z-NAF OTS signature sizes. Table 13 shows the comparison between z-NAF OTS and W-OTS processing time when using the optimized parameters for z-NAF OTS.

Table 13 – Comparison between the processing time (in terms of the number of calls to f and the number of CPU cycles used) of z-NAF OTS and W-OTS using the optimized parameters for z-NAF OTS. *Signature size is given as a multiple of 256-bit integers and does not include the checksum part. **The calls to f in the table does not include those performed during the calculation of the checksum chains.

Sig. Size*	Calls to f^{**}			CPU Cycles		
	KeyGen	Sign	Ver	KeyGen	Sign	Ver
75	76.1%	181.8%	-29.6%	73.4%	1240.6%	-15.9%
56	71.2%	172.5%	-30.1%	49.7%	1239.3%	-33.4%
46	18.7%	66.6%	-29.3%	16.2%	1332.5%	-28.2%
39	1.5%	32.0%	-29.0%	-9.6%	959.9%	-35.9%
32	37.7%	118.5%	-43.2%	32.3%	1141.0%	-44.3%
28	24.6%	113.8%	-64.6%	1.7%	840.4%	-69.1%
25	5.9%	92.5%	-80.8%	-18.2%	546.2%	-84.3%

Source: Author

Note that, compared to the initial parameters, the overhead in key generation and signature generation using the optimized parameters increased significantly, while the signature verification decreased substantially. This was the intended behaviour and shows that, by choosing carefully the parameters L_1 and t_{max} , one can trade more key generation and signature generation time for less signature verification time. We believe that our optimized parameters are still not optimal, and that better implementations of the z-NAF OTS would certainly require a different set of parameters to be optimal.

As with the initial parameters, we observe some trends, namely, that, as signature size decreases, the key generation overhead of z-NAF OTS reduces significantly compared to W-OTS, while the the signature generation overhead does not reduce much. The signature

verification time, however, reduces faster as signature size decreases, so for signature sizes smaller than 29×256 bits we expect even less signature verification time compared to W-OTS.

We also observe that, in the case of the optimized parameters, the increase in signature generation CPU cycles in z-NAF OTS compared to W-OTS was much larger than the increase using the initial parameters. This happens because the optimized parameters cause a large increase in the number of z-NAF conversions. Because the conversion algorithm used in this proof-of-concept implementation is not very efficient, the total CPU cycles of the signature generation algorithm, which performs many z-NAF conversions, was greatly increased. Therefore, we expect a large improvement on the CPU cycles for the signature verification algorithm using the optimized parameters if the conversion algorithm implementation is optimized to the same degree as the hash functions found in the OpenSSL library.

7 CONCLUSION

With the growing interest and investments in quantum technology by governments and private corporations, large scale quantum computers are growing closer to reality every year. This motivates the search for post-quantum cryptographic algorithms that can replace traditional schemes based on the (elliptic) discrete logarithm or number factorization, which would be vulnerable to attacks in a quantum scenario. Among the many existing proposals, hash-based digital signature schemes are quite promising, in particular due to their small public/private key pairs and to the fact that their security builds upon well-known properties of hash functions.

In this work, we propose two techniques aimed at improving hash-based signatures. The first, called “clipping technique”, is a novel approach that uses two layers of PRFs when generating public keys from private keys. The resulting scheme enables the compression of signatures generated from state-of-the-art hash-based OTS schemes, such as W-OTS and W-OTS+, as well as any MTS built upon them (e.g., XMSS and SPHINCS). The technique is probabilistic, meaning that the actual compression factor depends on the data to be signed; namely, one can obtain better compression rates for data where pairs of w -bit blocks processed by the OTS are filled with zeros, where w is the scheme’s Winternitz parameter. On average, the proposal can save roughly 12.5% in the signature size when $w = 1$, whereas the savings are lower for larger values of w . Nevertheless, since the resulting processing overheads drop for larger w , in practice it provides flexibility for signers, who can choose whether or not to compress the signature depending on the most constrained resource (bandwidth or processing) at verifiers. Therefore, the proposed clipping technique further advances the practical adoption of hash-based signatures in the post-quantum era. To the best of our knowledge, this is the first proposal in the literature that takes advantage of the private/public key generation process to reduce hash-based signature sizes.

The second technique proposes a novel OTS that uses the z-NAF encoding for the message to be signed. This has the advantage of skewing the frequency of zero and non-

zero digits and, thus, it can be used to speed up verification by slowing down signature generation. This is done while keeping the advantages of W-OTS and its variants, that is, the ability to recover the public key from the signature keeping small key sizes and comparable signature sizes. Hence, the proposed z-NAF OTS scheme can be used as a drop-in replacement for W-OTS and W-OTS+ where signature verifications are done much more frequently than signature generations, such as in software distribution applications.

7.1 Publications

An article titled "A clipping technique" was accepted and published by the symposium *Simpósio Brasileiro de Segurança* (SBSeg) 2018 (ZHENG; FERRAZ; SIMPLICIO JR, 2018). Another article is being written for describing the z-NAF OTS scheme.

7.2 Future Work

Both the clipping technique and the z-NAF OTS can be improved.

In the case of the clipping technique, one could investigate ways to make the number of zeros more frequent and/or more aligned (by using randomization techniques or more sophisticated message semantics, for example), which would make the average signature size even smaller.

For the z-NAF OTS, because it is a new proposal, many aspects could be improved. First, one could investigate whether the mapping $D'_i \rightarrow D''_i$ used in the proposal is optimal and, if not, what are the other better mappings available. Second, the z-NAF encoding can probably be explored in other ways to improve signature verification time. For example, requiring the first digit to be non-zero and encoding non-zero digits followed by zero runs is only one way of implementing the proposed scheme, and other options might provide even better results. Finally, the numbers presented in this article can certainly be improved by better implementations. A better implementation for the z-NAF conversion algorithm, for example, is likely to improve the signature generation times for small values of L_1 and/or t_{max} .

REFERENCES

- ANTHONY, S. *IBM will sell 50-qubit universal quantum computer “in the next few years”*. 2017. Accessed: 2017-09-28. Disponível em: <https://arstechnica.co.uk/gadgets/2017/03/ibm-q-50-qubit-quantum-computer>.
- AVANZI, R. M. A note on the signed sliding window integer recoding and a left-to-right analogue. In: SPRINGER. *International Workshop on Selected Areas in Cryptography*. Berlin, Heidelberg, 2004. p. 130–143.
- BERNSTEIN, D. J.; BUCHMANN, J.; DAHMEN, E. *Post-Quantum Cryptography*. Heidelberg, Deutschland: Springer, 2008.
- BERNSTEIN, D. J. et al. SPHINCS: practical stateless hash-based signatures. *Advances in Cryptology – EUROCRYPT 2015*, Springer, v. 9056 of Lecture Notes in Computer Science, p. 368–397, 2015.
- BERNSTEIN, D. J.; LANGE, T. *Post-quantum cryptography – dealing with the fallout of physics success*. 2017. Cryptology ePrint Archive, Report 2017/314. <https://eprint.iacr.org/2017/314>.
- BUCHMANN, J.; DAHMEN, E.; HÜLSING, A. XMSS – A practical forward secure signature scheme based on minimal security assumptions. *Post-Quantum Cryptography*, Springer, v. 7071 of Lecture Notes in Computer Science, p. 117–129, 2011.
- BUCHMANN, J. et al. CMSS – an improved Merkle signature scheme. In: BARUA, R.; LANGE, T. (Ed.). *Progress in Cryptology - INDOCRYPT 2006*. Berlin, Heidelberg: Springer, 2006. p. 349–363.
- COHEN, H. *Analysis of the flexible window powering algorithm*. 2001. <https://www.math.u-bordeaux.fr/~hecohen/>.
- DING, J.; SCHMIDT, D. Rainbow, a new multivariable polynomial signature scheme. *International Conference on Applied Cryptography and Network Security – ACNS 2005*, Springer, v. 3531 of Lecture Notes in Computer Science, p. 164–175, 2005.
- DODS, C.; SMART, N. P.; STAM, M. Hash based digital signature schemes. *Cryptography and Coding*, Springer, v. 3796 of Lecture Notes in Computer Science, p. 96–115, 2005.
- GOLDREICH, O.; GOLDWASSER, S.; HALEVI, S. Public-key cryptosystems from lattice reduction problems. *Advances in Cryptology – CRYPTO ’97*, Springer, v. 1294 of Lecture Notes in Computer Science, p. 112–131, 1997.
- GROVER, L. K. A fast quantum mechanical algorithm for database search. In: ACM. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. New York, NY, 1996. p. 212–219.

- HEUBERGER, C.; KRENN, D. Optimality of the width- w non-adjacent form: General characterisation and the case of imaginary quadratic bases. *J. Théor. Nombres Bordeaux*, v. 25, n. 2, p. 353–386, 2013.
- HÜLSING, A. W-OTS+ – shorter signatures for hash-based signature schemes. *Progress in Cryptology – AFRICACRYPT 2013*, Springer, Berlin, Heidelberg, v. 7918 of Lecture Notes in Computer Science, p. 173–188, 2013.
- HÜLSING, A.; BUSOLD, C.; BUCHMANN, J. Forward secure signatures on smart cards. *Selected Areas in Cryptography*, Springer, v. 7707 of Lecture Notes in Computer Science, p. 66–80, 2013.
- HÜLSING, A.; RAUSCH, L.; BUCHMANN, J. Optimal parameters for XMSS-MT. In: SPRINGER. *International Conference on Availability, Reliability, and Security*. Berlin, Heidelberg, 2013. p. 194–208.
- HÜLSING, A.; RIJNEVELD, J.; SCHWABE, P. ARMed SPHINCS. In: *Public-Key Cryptography–PKC 2016*. Berlin, Heidelberg: Springer, 2016. p. 446–470.
- JAO, D.; FEO, L. D. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: SPRINGER. *International Workshop on Post-Quantum Cryptography*. Berlin, Heidelberg, 2011. p. 19–34.
- JOHNSON, D.; MENEZES, A.; VANSTONE, S. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, v. 1, n. 1, p. 36–63, Aug 2001.
- KOBLITZ, N. Elliptic curve cryptosystems. *Mathematics of computation*, v. 48, n. 177, p. 203–209, 1987.
- LAMPORT, L. *Constructing Digital Signatures from a One Way Function*. Menlo Park, CA, 1979.
- MCELIECE, R. J. A public-key cryptosystem based on algebraic coding theory. *DSN Progress Report*, v. 42, p. 114–116, 1978.
- MENEZES, A. J.; VANSTONE, S. A.; OORSCHOT, P. C. V. *Handbook of Applied Cryptography*. 1st. ed. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN 0849385237.
- MERKLE, R. C. A certified digital signature. *Advances in Cryptology — CRYPTO' 89 Proceedings*, p. 218–238, 1990.
- MILLER, V. S. Use of elliptic curves in cryptography. In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Berlin, Heidelberg: Springer, 1986. p. 417–426.
- MIYAJI, A.; ONO, T.; COHEN, H. Efficient elliptic curve exponentiation. In: SPRINGER. *International Conference on Information and Communications Security*. Berlin, Heidelberg, 1997. p. 282–290.
- MUIR, J.; STINSON, D. Minimality and other properties of the width- w nonadjacent form. *Mathematics of Computation*, v. 75, n. 253, p. 369–384, 2006.

NIST. *Federal Information Processing Standard (FIPS 197) – Advanced Encryption Standard (AES)*. Gaithersburg, MD, 2001. Disponível em: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

_____. *Federal Information Processing Standard (FIPS 202) – SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Gaithersburg, MD, 2015. DOI:10.6028/NIST.FIPS.202.

_____. *FIPS 180-4 – Secure Hash Standard (SHS)*. Gaithersburg, MD, 2015.

PERRIG, A. The BiBa one-time signature and broadcast authentication protocol. In: ACM. *Proceedings of the 8th ACM conference on Computer and Communications Security*. New York, NY, 2001. p. 28–37.

REYZIN, L.; REYZIN, N. Better than BiBa: Short one-time signatures with fast signing and verifying. In: SPRINGER. *Australasian Conference on Information Security and Privacy*. Berlin, Heidelberg, 2002. p. 144–153.

RIVEST, R. L.; SHAMIR, A.; ADELMAN, L. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Commun. ACM*, ACM, v. 21, p. 120–126, 1977.

ROH, D.; JUNG, S.; KWON, D. Winternitz signature scheme using nonadjacent forms. *Security and Communication Networks*, Hindawi, v. 2018, 2018.

SAARINEN, M.-J. O.; AUMASSON, J.-P. The BLAKE2 cryptographic hash and message authentication code (MAC). *RFC*, v. 7693, p. 1–30, 2015.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.*, v. 41(2), p. 303–332, 1999.

SOLINAS, J. A. An improved algorithm for arithmetic on a family of elliptic curves. In: SPRINGER. *Annual International Cryptology Conference*. Berlin, Heidelberg, 1997. p. 357–371.

STEINWANDT, R.; VILLÁNYI, V. I. A one-time signature using run-length encoding. *Information Processing Letters*, Elsevier, v. 108, n. 4, p. 179–185, 2008.

ZHENG, A.; FERRAZ, L.; SIMPLICIO JR, M. A. A clipping technique for shorter hash-based signatures. In: SBSEG. *Anais do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*. Natal, RN, 2018. p. 167–180.

APPENDIX A – ADDITIONAL EXAMPLES OF THE Z-NAF OTS SCHEME

In Section 5.6 we gave an example of the z-NAF OTS scheme using $z = 3$. In this appendix we give additional examples in order to facilitate understanding of the scheme: the first additional example uses $z = 2$, while the second uses $z = 4$. Both examples use $n = 8$ for the message length.

A.1 Example with $z = 2$

A.1.1 Setup

In this example, $w = 3$, $L_1 = 4$ and $t_{max} = 3$. With these parameters, $L_2 = 2$ and $len = 8$.

A.1.2 Key Generation

The signer generates the key pair (x, y) with the method described in Section 5.2 and the above defined parameters. First, the signer generates x randomly, then he uses x and the PRF to generate $((x_m)_3, (x_m)_2, (x_m)_1, (x_m)_0, (x_c)_1, (x_c)_0)$ and calculates

$$y = g((y_m)_3 || (y_m)_2 || (y_m)_1 || (y_m)_0 || (y_c)_1 || (y_c)_0)$$

where

$$\begin{cases} (y_m)_i = f^{len-1}((x_m)_i) = f^7((x_m)_i) \\ (y_c)_i = f^{2^w-1}((x_c)_i) = f^7((x_c)_i) \end{cases}$$

A.1.3 Signature Generation

To sign the message m , the first thing the signer needs to do is choose an appropriate index k and calculate m' and m'' . Suppose that the signer tried many values of k_i , and found the suitable k_{10} with its corresponding m'_i and $(m'_i)_{3-NAF}$:

$$k_{10} \rightarrow m'_{10} = h(m||k_{10}) = 11001001 \rightarrow (m'_{10})_{2-NAF} = (1, 0, \bar{1}, 0, 0, 1, 0, 0, 1)$$

k_{10} works because the number of non-zero digits is $4 \leq L_1 = 4$ and the maximum number of trailing zeros after removing the mandatory zeros is $1 \leq t_{max} = 3$.

After choosing a suitable index, the signer removes $z - 1 = 1$ zeros at the left of each non-zero digit and obtains

$$m'_{grouped} = (1, \bar{1}, 01, 01)$$

The signer then calculates

$$m'' = (0, 1, 2, 2)$$

using the given map as well as the checksum and the checksum blocks

$$c = \sum_{i=0}^{L_1-1} len - 1 - D''_i = (23)_{10} = (010111)_2 \rightarrow (B_1, B_0) = (2, 7)$$

Finally, the signer calculates the signature

$$\sigma = (k_{10}, (\sigma_m)_3, (\sigma_m)_2, (\sigma_m)_1, (\sigma_m)_0, (\sigma_c)_1, (\sigma_c)_0)$$

where

$$\left\{ \begin{array}{l} (\sigma_m)_3 = f^{7-0}((x_m)_3) = f^7((x_m)_3) \\ (\sigma_m)_2 = f^{7-1}((x_m)_2) = f^6((x_m)_2) \\ (\sigma_m)_1 = f^{7-2}((x_m)_1) = f^5((x_m)_1) \\ (\sigma_m)_0 = f^{7-2}((x_m)_0) = f^5((x_m)_0) \\ (\sigma_c)_1 = f^{2^3-1-2}((x_c)_1) = f^5((x_c)_1) \\ (\sigma_c)_0 = f^{2^3-1-7}((x_c)_0) = f^0((x_c)_0) \end{array} \right.$$

A.1.4 Signature verification

Given the message m and the signature $\sigma = (k_{10}, (\sigma_m)_3, (\sigma_m)_2, (\sigma_m)_1, (\sigma_m)_0, (\sigma_c)_1, (\sigma_c)_0)$, the verifier proceeds with the same calculations that the signer did for m'' and get $m'' = (0, 1, 2, 2)$. He also calculates the checksum blocks the same way the signer did and get $(B_1, B_0) = (2, 7)$. Then, he calculates

$$y' = g((y_m)_3' || (y_m)_2' || (y_m)_1' || (y_m)_0' || (y_c)_1' || (y_c)_0')$$

where

$$\begin{cases} (y_m)_3' = f^0((\sigma_m)_3) \\ (y_m)_2' = f^1((\sigma_m)_2) \\ (y_m)_1' = f^2((\sigma_m)_1) \\ (y_m)_0' = f^2((\sigma_m)_0) \\ (y_c)_1' = f^2((\sigma_c)_1) \\ (y_c)_0' = f^7((\sigma_c)_0) \end{cases}$$

If $y' = y$ then the signer accepts the signature as valid; otherwise he rejects the signature as invalid.

A.2 Example with $z = 4$

A.2.1 Setup

In this example, $w = 4$, $L_1 = 4$ and $t_{max} = 1$. With these parameters, $L_2 = 2$ and $len = 16$.

A.2.2 Key Generation

The signer generates the key pair (x, y) with the method described in Section 5.2 and the above defined parameters. First, the signer generates x randomly, then he uses x and the PRF to generate $((x_m)_3, (x_m)_2, (x_m)_1, (x_m)_0, (x_c)_1, (x_c)_0)$ and calculates

$$y = g((y_m)_3 || (y_m)_2 || (y_m)_1 || (y_m)_0 || (y_c)_1 || (y_c)_0)$$

where

$$\begin{cases} (y_m)_i = f^{len-1}((x_m)_i) = f^{15}((x_m)_i) \\ (y_c)_i = f^{2^w-1}((x_c)_i) = f^{15}((x_c)_i) \end{cases}$$

A.2.3 Signature Generation

To sign the message m , the first thing the signer needs to do is choose an appropriate index k and calculate m' and m'' . Suppose that the signer tried many values of k_i , and found the suitable k_{10} with its corresponding m'_i and $(m'_i)_{4-NAF}$:

$$k_{10} \rightarrow m'_{10} = h(m || k_{10}) = 11001001 \rightarrow (m'_{10})_{4-NAF} = (1, 0, 0, 0, \bar{3}, 0, 0, 0, \bar{7})$$

k_{10} works because the number of non-zero digits is $3 \leq L_1 = 4$ and the maximum number of trailing zeros after removing the mandatory zeros is $0 \leq t_{max} = 1$.

After choosing a suitable index, the signer removes $z - 1 = 3$ zeros at the left of each non-zero digit and obtains

$$m'_{grouped} = (1, \bar{3}, \bar{7})$$

The signer then calculates

$$m'' = (0, 0, 5, 4)$$

using the given map (note: the leftmost digit in m'' is zero because the number of non-zero digits in m'_{z-NAF} is $3 \leq L_1 = 4$) as well as the checksum and the checksum blocks

$$c = \sum_{i=0}^{L_1-1} len - 1 - D''_i = (36)_{10} = (00100100)_2 \rightarrow (B_1, B_0) = (2, 4)$$

Finally, the signer calculates the signature

$$\sigma = (k_{10}, (\sigma_m)_3, (\sigma_m)_2, (\sigma_m)_1, (\sigma_m)_0, (\sigma_c)_1, (\sigma_c)_0)$$

where

$$\begin{cases} (\sigma_m)_3 = f^{15-0}((x_m)_3) = f^{15}((x_m)_3) \\ (\sigma_m)_2 = f^{15-0}((x_m)_2) = f^{15}((x_m)_2) \\ (\sigma_m)_1 = f^{15-5}((x_m)_1) = f^{10}((x_m)_1) \\ (\sigma_m)_0 = f^{15-4}((x_m)_0) = f^{11}((x_m)_0) \\ (\sigma_c)_1 = f^{2^4-1-2}((x_c)_1) = f^{13}((x_c)_1) \\ (\sigma_c)_0 = f^{2^4-1-4}((x_c)_0) = f^{11}((x_c)_0) \end{cases}$$

A.2.4 Signature verification

Given the message m and the signature $\sigma = (k_{10}, (\sigma_m)_3, (\sigma_m)_2, (\sigma_m)_1, (\sigma_m)_0, (\sigma_c)_1, (\sigma_c)_0)$, the verifier proceeds with the same calculations that the signer did for m'' and get $m'' = (0, 0, 5, 4)$. He also calculates the checksum blocks the same way the signer did and get $(B_1, B_0) = (2, 4)$. Then, he calculates

$$y' = g((y_m)'_3 || (y_m)'_2 || (y_m)'_1 || (y_m)'_0 || (y_c)'_1 || (y_c)'_0)$$

where

$$\begin{cases} (y_m)'_3 = f^0((\sigma_m)_3) \\ (y_m)'_2 = f^0((\sigma_m)_2) \\ (y_m)'_1 = f^5((\sigma_m)_1) \\ (y_m)'_0 = f^4((\sigma_m)_0) \\ (y_c)'_1 = f^2((\sigma_c)_1) \\ (y_c)'_0 = f^4((\sigma_c)_0) \end{cases}$$

If $y' = y$ then the signer accepts the signature as valid; otherwise he rejects the signature as invalid.