Optimizing Hash-Based Signatures in Java

Bachelor's Thesis of

Tim Rausch

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr. Jörn Müller-Quade
Second reviewer:	Prof. Dr. Thorsten Strufe
Advisor:	M.Sc. Felix Dörre
External advisor:	Dr. Anselme Tueno (SAP SE)

01. December 2022 – 28. March 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

Karlsruhe, 28. March 2023

. . . . . .

(Tim Rausch)

Abstract

Hash-based signature schemes are an extensively studied and well-understood choice for quantum-safe digital signatures. However, certain operations, most notably the key generation, can be comparably expensive. It is, therefore, essential to use well-optimized implementations.

This thesis aims to explore, implement, and evaluate optimization strategies for hash-based signature implementations in Java. These include the use of special hardware features like vector instructions and hardware acceleration for hash functions as well as the parallelization of the key generation. Overall, we are able to reduce the time required for an XMSS key generation with SHA-2 by up to 96.4% (on four CPU cores) compared to the unmodified BouncyCastle implementation. For SPHINCS$^+$ with the Haraka hash function family, we achieve a reduction of up to 95.7% on only one CPU core.

Furthermore, we investigate the use of two scheme variants WOTS-BR and WOTS+C proposed in the literature for verification-optimized signatures. We improve the existing theoretical analysis of both, provide a comparison and experimentally validate our improved theoretical analysis.

Zusammenfassung

Hashbasierte Signaturverfahren sind eine ausführlich untersuchte und gut verstandene Option für quantensichere digitale Signaturen. Jedoch können einige Operationen, vor allem die Schlüsselerzeugung, vergleichsweise teuer sein. Deswegen ist es notwendig, gut optimierte Implementierungen zu verwenden.

Diese Thesis zielt darauf ab, Optimierungsstrategien für hashbasierte Signaturverfahren in Java zu erkunden, zu implementieren und zu evaluieren. Dies umfasst die Nutzung von speziellen Hardwarefunktionen, wie Vektorinstruktionen und Hardwarebeschleunigung für Hashfunktionen, sowie die Parallelisierung der Schlüsselerzeugung. Insgesamt konnten wir die für eine XMSS Schlüsselerzeugung mit SHA-2 benötigte Zeit um bis zu 96.4% reduzieren (auf vier Prozessorkernen), verglichen mit der unveränderten BouncyCastle-Implementierung. Für SPHINCS$^+$ mit der Haraka-Hashfuntionenfamilie erreichen wir eine Reduktion um bis zu 95.7% auf nur einem Prozessorkern.

Zusätzlich untersuchen wir den Einsatz der zwei Verfahrensvarianten WOTS-BR und WOTS+C aus der Literatur für verifikationsoptimierte Signaturen. Wir verbessern die bestehende theoretische Analyse beider, vergleichen sie und bestätigen unsere verbesserte theoretische Analyse experimentell.

Contents

Abbreviations

ACCP	Amazon Corretto Crypto Provider
AES	Advanced Encryption Standard
AES-NI	AES New Instructions
AVX	Intel Advanced Vector Extensions
AVX-512	Intel Advanced Vector Extensions 512
AVX2	Intel Advanced Vector Extensions 2
AWS	Amazon Web Services
CMSS	Chained Merkle Signature Schemes
CRHF	Collision-Resistant Hash Function
EC2	Amazon Elastic Compute Cloud
EUF-CMA	Existential Unforgability under Chosen-Message Attack
EUF-SMA	Existential Unforgability under Single-Message Attack
FIPS	Federal Information Processing Standard
FORS	Forest of Random Subsets
FTS	Few-Time Signature Scheme
HBS	Hash-Based Signature Scheme
HORS	Hash to Obtain Random Subset
HORST	HORS with Trees
HSS	Hierarchical Signature Scheme
JCA	Java Cryptographic Architecture
JCE	Java Cryptographic Extension
JIT	Just-in-Time Compiler
JMH	Java Microbenchmark Harness
JNI	Java Native Interface
JVM	Java Virtual Machine
LD-OTS	Lamport-Diffie One-Time Signature Scheme
LMS	Leighton-Micali Signature Scheme
MSS	Merkle Signature Scheme
NIST	National Institute of Standards and Technology
OTS	One-Time Signature Scheme
OWF	One-Way Function
PPT	Probabilistic Polynomial-Time
PQC	Post-Quantum Cryptography
RFC	Request for Comments
SHA-NI	SHA New Instructions

SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multithreading
SP	Special Publication
SSE	Streaming SIMD Extensions
SUF-CMA	Strong Unforgability under Chosen-Message Attack
SUF-SMA	Strong Unforgability under Single-Message Attack
WOTS	Winternitz One-Time Signature Scheme
XKCP	eXtended Keccak Code Package
XMSS	eXtended Merkle Signature Scheme
XMSS^{MT}	eXtended Merkle Signature Scheme Multi-Tree
XOF	Extendable-Output Function

List of Figures

List of Tables

1. Introduction

This thesis proposes, implements, and evaluates optimizations for Hash-Based Signature Schemes (HBSs) in Java. Section 1.1 presents the motivation behind this thesis, Section 1.2 summarizes our main contributions, and Section 1.3 outlines the structure of this thesis.

1.1. Motivation

HBSs are promising candidates for quantum-safe signature schemes. As opposed to most other established and post-quantum cryptographic schemes, they do not rely on any number-theoretical assumptions and only require a secure hash function. Secure hash-based signature schemes are proven to exist as long as any secure digital signature scheme exists [9]. Furthermore, HBSs have been extensively studied and are considered to be well-understood. They are, therefore, deemed a conservative choice for quantum-safe signatures.

Two stateful HBS schemes are already standardized as RFCs: the eXtended Merkle Signature Scheme (XMSS) [9, 36] and the Leighton-Micali Signature Scheme (LMS) [49]. SPHINCS$^+$ [5] is a stateless HBS scheme that was chosen by NIST in the Post-Quantum Cryptography Standardization project and will therefore be standardized [51].

Implementation Optimizations In contrast to most other signature schemes, key pairs for both stateful and stateless HBSs can only be used for a limited number of signatures. For XMSS and LMS, the time required to generate a key pair is linear in the number of signatures that can be created with it. Generating XMSS keys valid for 2^{20} signatures may take several hours. For practical applications, it is therefore essential to optimize and speed-up HBS operations.

While most HBS implementations are written in C or C++ for better performance, the Java platform is one of the most popular programming environments for business applications. For the security and performance of Java applications, it is critical to have fast and quantum-safe cryptographic primitives available. Therefore, this thesis aims to improve the performance of HBS implementations in Java.

Verification-Optimized Signatures An important application scenario for digital signatures is code signing. Software and firmware packages are signed to prove their authenticity to the users. For code signing, the number of verification operations is significantly greater than the number of key generation or signature operations. Consider a device that verifies the signature of its firmware at every startup. In this scenario, the verification of the signature must be as fast as possible. At the same time, a greater cost of signing may be acceptable due to the low number of created signatures.

Thus, this thesis investigates and implements scheme variants for verification-optimized signatures.

1.2. Contribution

Implementation Optimizations We present, implement, and evaluate various optimizations to accelerate the implementation of XMSS, LMS, and SPHINCS$^+$ in Java.

Therefore, we explore strategies to optimize the evaluation of underlying hash functions. These include the scheme-agnostic integration of native implementations that utilize specific hardware acceleration as well as scheme-specific software optimizations. We, furthermore, investigate the use of parallelization on modern multi-core systems for certain HBS operations. We provide extensive benchmark results and guidance for HBS implementations in Java based on our insights.

For XMSS with SHA-2, we are able to reduce the key generation time by up to 85.0% using one CPU core and by up to 96.4% on four CPU cores. We achieve even better results for SPHINCS$^+$ with the Haraka hash function. On one core, we are able to reduce the key generation time by 95.4%.

Verification-Optimized Signatures This thesis provides a theoretical analysis, implementation, experimental validation, and comparison of the use of the schemes WOTS-BR and WOTS+C for verification-optimized signatures.

In the theoretical analysis, we present a new and more accurate model for the behavior of the checksum in WOTS-BR than the models presented in the literature. For WOTS+C, we correct the model provided in the literature and give a theoretical comparison of both schemes based on the presented models. Furthermore, we experimentally validate them by implementing and benchmarking the schemes.

1.3. Organization

The remainder of this thesis is organized as follows: Chapter 2 presents the theoretical background required for this thesis and introduces the underlying constructions used in the HBSs. Chapter 3 presents how these constructions are used to build HBSs for practical

use. An overview of related work that optimizes HBSs is given in Chapter 4. Chapter 5 introduces the architecture of the Java platform, the various software projects used in this thesis, and the benchmarking methodology used to evaluate the optimizations.

Chapter 6 describes the implementation optimizations in detail and provides an analysis of WOTS-BR and WOTS+C for verification-optimized signatures. Chapter 7 presents and evaluates the results of benchmarks based on the previous chapter. This thesis concludes in Chapter 8 by providing a summary and an overview of potential future work.

2. Background and Theory

The current chapter discusses the theoretical background relevant for the understanding of the thesis. It starts in Section 2.1 by describing the notation used throughout this document. In Sections 2.2 and 2.3, we give a brief overview over digital signatures and hash functions. These two concepts will be used later in this chapter to introduce hash-based one-time signature schemes in Section 2.4 and hash-based few-time signature schemes in Section 2.7. Sections 2.5 and 2.6 present constructions building upon these signature schemes to achieve keys that can be used for many signatures.

2.1. Notation

For strings s and t, let $|s|$ denote the length of the string s and $s \parallel t$ their concatenation. By $\{0,1\}^n$, we denote the set of all bit strings of length n and by $\{0,1\}^*$, the set of all bit strings of arbitrary length. We denote the set of all j-tuples of bit strings of length n as $\{0,1\}^{(n,j)}$. The function $\mathsf{trunc}_i(x)$ truncates the bit string x to length i. More precisely, it returns the first i characters of x. For $x \in \mathbb{N}$, the function $\mathsf{toByte}(x, y)$ returns the binary representation of x as a y-byte string in big-endian order.

By $x \xleftarrow{\$} K$, we denote that x is randomly sampled from the set K using the uniform distribution. This thesis only uses the logarithm to base 2. Consequently, the function log refers to the logarithm to base 2, $\log_2$.

For a set S, let $\mathbb{P}(S)$ denote its power set, i.e. the set of all subsets of S. Let $\mathbb{P}_k(S)$ be the set of all non-empty subsets of S with no more than k elements. Precisely, $\mathbb{P}_k(S) = \{s \in \mathbb{P}(S) \mid 0 < |s| \leq k\}$. By $[z]$, we denote the set $\{0, \ldots, z-1\}$.

2.2. Digital Signatures

Digital signatures are a means to ensure the authenticity and integrity of a message in a public-key setting: A signer has a key pair consisting of a secret key X, which is only known to the signer, and a public key Y which is assumed to be publicly known. The signer can generate a signature for a certain message using X and transmit both the message and signature to one or multiple verifiers. A verifier can now verify the signature for this message using the public key Y. The following section is based on Katz and Lindell [40].

Formally, a digital signature scheme is a tuple of Probabilistic Polynomial-Time (PPT) algorithms $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ where:

1. Gen is the key-generation algorithm: $(X, Y) \leftarrow \mathsf{Gen}(1^\lambda)$. It takes a security parameter 1^λ as input and returns a key pair consisting of the secret key X and the public key Y,
2. Sign is the signature algorithm: $\sigma \leftarrow \mathsf{Sign}_X(M)$. It takes a private key X and a message M as input and returns a signature σ, and
3. Verify is the deterministic verification algorithm: $b = \mathsf{Verify}_Y(M', \sigma')$. The algorithm takes a public key Y, a message M', and a signature σ' as input and returns a bit b. If $b = 1$ the signature is considered valid, otherwise invalid.

For this thesis, we require perfect correctness: For every $\lambda \in \mathbb{N}$, $(X, Y) \leftarrow \mathsf{Gen}(1^\lambda)$, $M \in \{0,1\}^*$, and $\sigma \leftarrow \mathsf{Sign}_X(M)$, the condition $\mathsf{Verify}_Y(M, \sigma) = 1$ must hold.

For this to be useful, it must additionally be infeasible for an attacker to create a forgery, that is, to compute a new signature that is accepted by a verifier but was not created by the signer. The most common security property for digital signature schemes is Existential Unforgability under Chosen-Message Attack (EUF-CMA) which is defined using the experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(\lambda)$.

For a signature scheme $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$, an algorithm $\mathcal{A}$, and a security parameter λ, the experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(\lambda)$ is defined as:

1. A key pair is generated using the key-generation algorithm: $(X, Y) \leftarrow \mathsf{Gen}(1^\lambda)$.
2. The public key Y and access to a signature oracle $\mathsf{Sign}_Y(\cdot)$ are given to the algorithm $\mathcal{A}$, which then outputs a tuple (M, σ).
3. Let Q be the set of all oracle queries made by $\mathcal{A}$. The experiment outputs 1 if and only if $\mathsf{Ver}_Y(M, \sigma) = 1$ and $M \notin Q$.

A signature scheme $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ is considered existentially unforgeable under chosen-message attack (EUF-CMA) if, for all PPT algorithms $\mathcal{A}$, a negligible function negl exists such that:

$$\Pr\left[\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda).$$

A related, but stronger security property is Strong Unforgability under Chosen-Message Attack (SUF-CMA) which is defined via the experiment $\mathsf{Sig\text{-}sforge}_{\mathcal{A},\Pi}(\lambda)$. This experiment is defined analogous to $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(\lambda)$ with one difference: We define Q as the set of all message-signature tuples provided by the oracle. A tuple (M, σ) output by $\mathcal{A}$ is only accepted if $(M, \sigma) \notin Q$.

An algorithm $\mathcal{A}$ wins the experiment $\mathsf{Sig\text{-}sforge}_{\mathcal{A},\Pi}(\lambda)$ if it outputs a valid signature for any message, as long as the pair of message and signature was not previously obtained from the oracle. This is experiment is easier for $\mathcal{A}$ to win, therefore SUF-CMA is the stronger property.

2.3. Hash Functions

Hash functions map inputs of arbitrary length to fixed-length outputs. Furthermore, hash function must have other properties to be useful for cryptographic applications.

This section gives an overview of these security properties and the constructions for hash functions. Sections 2.3.1 and 2.3.2 introduce One-Way Functions and Collision-Resistant Hash Functions, respectively. Further security properties are given in Section 2.3.3. Sections 2.3.4 and 2.3.5 present two constructions to build hash functions for arbitrary input data from building blocks that only accept fixed-size inputs.

2.3.1. One-Way Functions

A One-Way Function (OWF) is a function $f : \{0,1\}^* \to \{0,1\}^*$ that can be efficiently computed but cannot be efficiently inverted. The formal definition presented below is based on Katz and Lindell [40].

The requirement that f is efficiently computable can be easily formalized: we required the existence of a polynomial-time algorithm that computes $f(x)$ for a given x. The hardness of the function inversion is defined over an experiment.

Inverting Experiment For a given function $f : \{0,1\}^* \to \{0,1\}^*$, an algorithm $\mathcal{A}$, and a security parameter λ, the inverting experiment $\mathsf{Invert}_{\mathcal{A},f}(n)$ is defined as:

- Sample $x \xleftarrow{\$} \{0,1\}^\lambda$ randomly using the uniform distribution and compute $y = f(x)$,
- Compute $x' \leftarrow \mathcal{A}(1^\lambda, y)$,
- Output 1 if $x = x'$ and 0 otherwise.

One-Way Property A function $f : \{0,1\}^* \to \{0,1\}^*$ is one-way if it fulfills the following two requirements:

1. A polynomial-time algorithm M_f exists that computes f. That is, $M_f(x) = f(x)$ for all $x \in \{0,1\}^*$.
2. For every PPT algorithm $\mathcal{A}$, the success probability in the inverting experiment is negligible. That is, a negligible function negl exists such that

$$Pr[\mathsf{Invert}_{\mathcal{A},f}(\lambda) = 1] \leq \mathsf{negl}(\lambda).$$

2.3.2. Collision-Resistant Hash Functions

Collision-Resistant Hash Functions (CRHFs) are functions for which it is hard to find collisions. That is, distinct values x, x' that are mapped to the same value in the function's range. Below, we give a formal definition based on Katz and Lindell [40].

Keyed Hash Functions A (keyed) hash function is a tuple of PPT algorithms (Gen, H) that fulfill:

- Gen takes the security parameter 1^λ as input and outputs a random key k. The security parameter 1^λ is assumed to be implicitly contained in k.
- H is a deterministic algorithm that takes a key k and a bit string x as input and returns a value $H_k(x) \in \{0,1\}^{l(\lambda)}$.

H is called a fixed-length hash function or compression function if $H_k(x)$ is only defined for strings $x \in \{0,1\}^{l'(1^\lambda)}$.

Collision-Finding Experiment For a given hash function $\Pi = (\mathrm{Gen}, H)$, an algorithm $\mathcal{A}$, and a security parameter λ, the collision-finding experiment $\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(\lambda)$ is defined as:

1. Generate a function key $k \leftarrow \mathrm{Gen}(\lambda)$.
2. Execute $\mathcal{A}$ on input k. The algorithm $\mathcal{A}$ returns two bit strings x, x'.
3. Output 1 if $x \neq x'$ and $H_k(x) = H_k(x')$, and 0 otherwise.

Collision-Resistant Hash Functions A hash function $\Pi = (\mathrm{Gen}, H)$ is collision-resistant if, for every PPT algorithm $\mathcal{A}$, a negligible function negl exists such that

$$\Pr\left[\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda).$$

2.3.3. Other Security Properties

Apart from collision resistance, two weaker security properties for hash functions are commonly used: Preimage resistance and second-preimage resistance.

Preimage Resistance Given $k \leftarrow \mathrm{Gen}(1^\lambda)$ and $y \in \{0,1\}^{l(\lambda)}$, the success probability of all PPT algorithms in finding x with $H_l(x) = y$ is negligible.

Second-Preimage Resistance Given $k \leftarrow \mathrm{Gen}(1^\lambda)$ and $x \in \{0,1\}^\lambda$, the success probability of all PPT algorithms in finding x' with $x' \neq x$ and $H_k(x) = H_k(x')$ is negligible.

Note that collision resistance implies second-preimage resistance which in turn implies preimage resistance [40]. The notion of preimage resistance for hash functions is essentially equivalent to the one-way property introduced in Section 2.3.1.

Birthday Attacks Birthday attacks are generic attacks on the collision resistance of a hash function. Boneh and Shoup [7] and Katz and Lindell [40] give a detailed presentation and analysis. For this thesis, it is only important to know that for a hash function with range $\{0,1\}^l$, it is possible to find a collision with only $O(2^{l/2})$ evaluations of H [7].

In terms of concrete security, this means that if finding a collision for H should be as hard as the exhaustive search over all n-bit keys, the hash function must have an output length of at least $2n$ bits [40].

This type of attack can only be used to find collisions, not to find preimages or second preimages. There are no other generic attacks on the preimage or second-preimage resistance of a hash function that require fewer than 2^l evaluations of H [40].

In summary, hash functions with an output length of l bits can provide a security level of l against preimage and second-preimage attacks, but only $l/2$ against collision attacks.

2.3.4. Merkle-Damgård Constrution

The Merkle-Damgård construction can be used to construct a CRHF for arbitrary-length inputs from a collision-resistant compression function. We require the compression function (Gen, h) to have an input length of $2n$ and an output length of n. This restriction is not strictly necessary for the construction, but it allows for a simpler presentation.

Additionally, a padding function $\mathrm{pad} : \mathbb{N} \to \{0,1\}^*$ is required. The function pad must fulfill the following properties:

1. $i + |pad(i)|$ must be a multiple of n for all $i \in \mathbb{N}$,
2. $x \parallel pad(|x|) \neq x' \parallel pad(|x'|)$ for all distinct $x, x' \in \{0,1\}^*$.

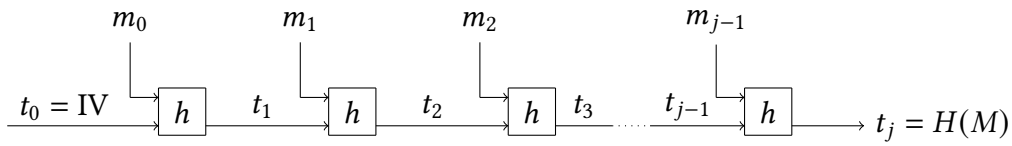


Figure 2.1.: Illustration of the Merkle-Damgård iterated hash function. Based on [7].

To calculate the Merkle-Damgård hash for a message M, it is first padded to block length using pad: $M' = M \parallel pad(|M|)$. The padded message is then split into $j = \frac{|M'|}{n}$ bit strings of length n: $M' = m_0 \parallel \cdots \parallel m_{j-1}$.

The message blocks are processed iteratively. With each processed block, the internal state t is updated. The initial state is a fixed initialization vector: $t_0 = \mathrm{IV}$. For each block, the state is updated using the compression function:

$$t_{i+1} = h_s(t_i \parallel m_i) \quad i \in [j].$$

The Merkle-Damgård hash $H(M)$ is defined as $H(M) = t_j$. Boneh and Shoup [7] propose a padding function and prove that the Merkle-Damgård construction with this padding function yields a CRHF.

2.3.5. Sponge Constructions

Sponge constructions present an alternative approach to building hash functions in practice. Unlike Merkle-Damgård, sponge constructions require no collision-resistant compression function, but only a permutation $\pi : \{0,1\}^n \rightarrow \{0,1\}^n$. The presentation below is based on Boneh and Shoup [7].

However, it is not known how to generically prove the collision resistance of the resulting hash function based on a concrete security property of π.

In addition to supporting variable input sizes, sponge constructions can also generate hash values of variable length.

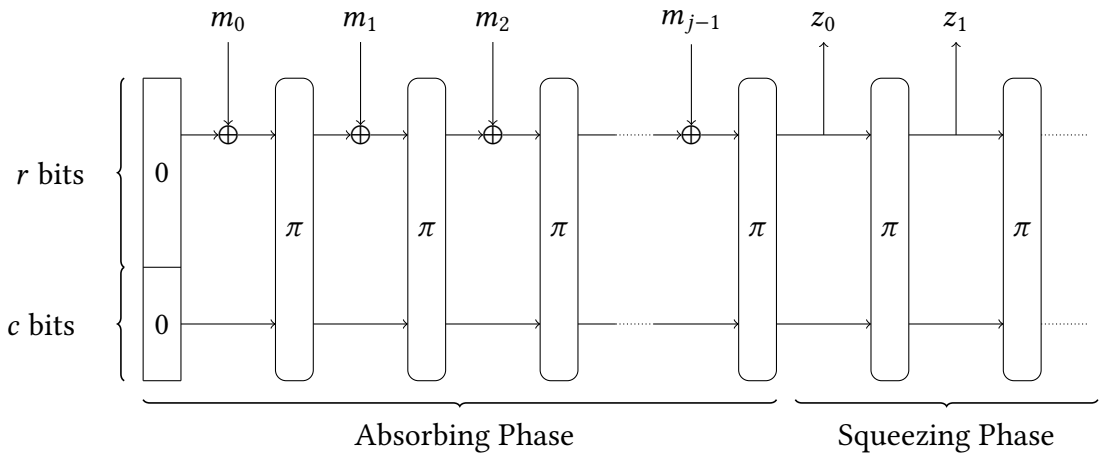


Figure 2.2.: Illustration of the sponge construction. Based on [7].

A sponge construction is parameterized by its rate r and its capacity c. Rate and capacity must fulfill $r + c = n$.

Similar to Merkle-Damgård, the message M is padded to a multiple of r using a padding function pad. Let $M' = M \parallel pad(|M|)$ and split M' into $j = \frac{|M'|}{r}$ blocks of length r: $M' = m_0 \parallel \cdots \parallel m_{j-1}$.

The sponge construction operates on an internal state of n bits. The initial value is usually $t_0 = 0^n$.

In the absorbing phase, each message block is then processed to update the state:

$$t_{i+1} = \pi(t_i \oplus (m_i \parallel 0^c)) \quad i \in [j]$$

After all message blocks have been processed, the resulting state t_j is used in the squeezing phase to compute the hash value. To generate a l-bit hash value, set $k = \lceil l/r \rceil$ and compute

$$\begin{aligned} t_{j+i+1} &= \pi(t_{j+i}) \qquad i \in [k-1], \\ z_i &= \mathsf{trunc}_r(t_{j+i}) \qquad i \in [k], \\ z &= \mathsf{trunc}_l(z_0 \parallel \cdots \parallel z_{k-1}) \end{aligned}$$

The resulting hash value of the sponge construction is z.

2.4. Hash-Based One-Time Signatures

In this section, we present two hash-based One-Time Signature Schemes (OTSs): the Lamport-Diffie One-Time Signature Scheme (LD-OTS) in Section 2.4.1, and the Winternitz One-Time Signature Scheme (WOTS) in Section 2.4.2. After that, we introduce two variants of the latter in Sections 2.4.3 and 2.4.4.

2.4.1. Lamport One-Time Signature Scheme

The Lamport one-time signature scheme, sometimes also referred to as Lamport-Diffie One-Time Signature Scheme (LD-OTS), was originally proposed by Lamport in 1979 [47]. In contrast to traditional digital signatures, a one-time signature scheme can only be used to sign exactly one message per generated key pair. In the following, we present the three algorithms of LD-OTS: Key generation, signing, and verification. We follow the notation used by Buchmann et al. [10].

Prerequisites Let n be the security parameter, a positive integer. LD-OTS requires two functions: a one-way function f and a cryptographic hash function g:

$$\begin{aligned} f &: \{0,1\}^n \to \{0,1\}^n, \\ g &: \{0,1\}^* \to \{0,1\}^n. \end{aligned}$$

Key Generation The secret key X is an ordered set of $2n$ bit strings of length n which are randomly sampled:

$$X = (x_{n-1}[0], x_{n-1}[1], \ldots, x_0[0], x_0[1]) \xleftarrow{\$} \{0,1\}^{(n,2n)}.$$

The public key Y is derived by applying the one-way function f to each element individually:

$$\begin{aligned} y_i[j] &= f(x_i[j]) \quad i \in [n],\ j \in \{0,1\}, \\ Y &= (y_{n-1}[0], y_{n-1}[1], \ldots, y_0[0], y_0[1]). \end{aligned}$$

Public and secret key have a size of $2n^2$ bits each.

Signing A message $M \in \{0,1\}^*$ is first hashed using the cryptographic hash function g: $d = g(M)$. Let $d_{n-1}, \ldots, d_0$ denote the individual bits of the hash value, i.e. $d = (d_{n-1}, \ldots, d_0)$. Each bit is signed individually and the signature σ_i for the bit d_i is the value $x_i[d_i]$. The signature for the message M is, therefore, defined as

$$\sigma = (x_{n-1}[d_{n-1}], \ldots, x_0[d_0]) \in \{0,1\}^{(n,n)}.$$

We note that the signature's size is n^2 bits in total and signature generation requires no evaluation of f.

Verification To verify a signature $\sigma' = (\sigma'_{n-1}, \ldots, \sigma'_0)$ for a message M' and a public key Y, the message is hashed $d' = g(M') = (d'_{n-1}, \ldots, d'_0)$. The signature is considered valid if and only if

$$f(\sigma'_i) \stackrel{?}{=} y_i[d'_i] \qquad \forall i \in [n].$$

Verification of an LD-OTS signature requires n evaluations of f.

Security The security of LD-OTS depends on three factors: the one-way property of f, the collision resistance of g, and the one-time use of key pairs. If f is not a one-way function, the secret key (or parts thereof) can be derived from the public key. Intuitively, a key pair can be used only once because the signature process reveals parts of the secret key. If only one message is signed, the revealed parts of the secret key can only be used to sign messages with exactly this hash value. However, if a key is re-used, this will reveal additional parts of the secret key. This information can be used to sign messages with certain other hash values. In the worst case, assume two messages M, M' that fulfill $d_i \neq d'_i$ for all $i \in [n]$. Their signatures σ, σ' can be used to trivially reconstruct the entire secret key X. This can, in turn, be used to generate signatures for arbitrary messages.

Note that this means that LD-OTS is not EUF-CMA secure. Instead, it is necessary to introduce new security properties for OTSs. These are Existential Unforgability under Single-Message Attack (EUF-SMA) and Strong Unforgability under Single-Message Attack (SUF-SMA) which are defined similar to their CMA counterparts. The only difference is that the attacker $\mathcal{A}$ can only send one query to the oracle instead of a polynomial number of queries. This restriction models the one-time use of keys. Katz and Lindell [40] prove that LD-OTS indeed fulfills EUF-SMA if f is one-way. Furthermore, Chia et al. [12] claim SUF-SMA security if instantiated with a one-way function.

Example Assume $n = 4$. In this case, both keys X and Y consist of 8 bit strings. We depict this scenario in Figure 2.3. To sign a message M with hash value $d = 1101$, the secret key elements $x_3[1]$, $x_2[1]$, $x_1[0]$, and $x_0[1]$ are used as the signature. To illustrate the consequences of key reuse, we assume another message M' with hash value $d' = 1010$ is signed. The signature for this message is $x_3[1]$, $x_2[0]$, $x_1[1]$, and $x_0[0]$. Observe that the two signatures reveal all parts of the secret key except $x_3[0]$. An attacker can now use this information to trivially generate forgeries for all messages whose hash value starts with 1.

Key Generation:

$$\begin{pmatrix} x_3[0] & \boxed{x_2[0]} & x_1[0] & \boxed{x_0[0]} \\ \boxed{x_3[1]} & x_2[1] & \boxed{x_1[1]} & x_0[1] \end{pmatrix} \xrightarrow{f} \begin{pmatrix} y_3[0] & y_2[0] & y_1[0] & y_0[0] \\ y_3[1] & y_2[1] & y_1[1] & y_0[1] \end{pmatrix}$$

Secret Key X — Public Key Y

Sign first message M:

$g(M)$:	1	1	0	1
σ:	$\boxed{x_3[1]}$	$\boxed{x_2[1]}$	$\boxed{x_1[0]}$	$\boxed{x_0[1]}$

Sign second message M':

$g(M')$:	1	0	1	0
σ':	$x_3[1]$	$x_2[0]$	$x_1[1]$	$x_0[0]$

Figure 2.3.: Example of the LD-OTS key generation and signature with key reuse for $n = 4$

2.4.2. Winternitz One-Time Signature Scheme

The Winternitz One-Time Signature Scheme (WOTS) is a similar one-time scheme that was presented by Merkle [50]. It can be considered an improvement of LD-OTS and has a generally smaller key and signature size. It allows for a trade-off between the size of keys and signatures and the run-time of the operations. The presentation below is based on [10, 9].

Prerequisites As with LD-OTS, we require a security parameter n, a one-way function f, and a cryptographic hash function g. Let $w = 2^i$ be the Winternitz parameter for some $i \in \mathbb{N}_+$.

Note that this definition deviates from Buchmann et al. [10] who define the Winternitz parameter as $w_l = \log w$. Both definitions are used in the literature. To avoid confusion, the definition above is used for the rest of this thesis as in [9, 36].

Let

$$l_1 = \left\lceil \frac{n}{\log w} \right\rceil, \qquad l_2 = \left\lfloor \frac{\log(l_1(w-1))}{\log w} \right\rfloor + 1, \qquad l = l_1 + l_2$$

as in [9].

Key Generation The secret key X consists of l bit strings of length n chosen at random:

$$X = (x_{l-1}, \ldots, x_0) \xleftarrow{\$} \{0,1\}^{(n,l)}.$$

The iterated application of f is defined as

$$f^i(x) = \begin{cases} x & i = 0 \\ f^{i-1}(f(x)) & i \neq 0 \end{cases}.$$

Similar to LD-OTS, the public key Y is derived from X using the one-way function f. However, f is applied $w-1$ times to each x_i:

$$y_i = f^{w-1}(x_i) \quad i \in [l]$$

$$Y = (y_{n-1}, \ldots, y_0)$$

The key generation requires $l(w-1)$ evaluations of f and both secret and public keys have a size of ln bits [10].

Signing To sign a message M, a hash value $d = g(M)$ is computed . The hash d is then split into l_1 bit strings $b_{l_1-1}, \ldots, b_0$ of length $\log w$. If $\log w$ does not divide $n = |d|$, d is padded with zeroes on the left. This corresponds to a base-w encoding of d interpreted as an integer.

$$d = b_{l_1-1} \parallel \cdots \parallel b_0$$

To prevent trivial forgeries, a checksum is calculated and signed along the message blocks $b_{l_1-1}, \ldots, b_0$. To calculate this checksum, the b_i are interpreted as integers in $[w]$. Let

$$C = \sum_{i=0}^{l_1-1} (w - 1 - b_i).$$

The checksum C is encoded to its base-w representation: $C = (c_{l_2-1}, \ldots, c_0)$. This is equivalent to converting C to its binary representation and splitting it into bit strings of length $\log w$. Due to $C \leq l_1(w-1)$, the length of the base-w representation is not greater than l_2 [11]. Set $(b_{l-1}, \ldots, b_{l_1}) = (c_{l_2-1}, \ldots, c_0)$.

The signature of M is defined as:

$$\sigma_i = f^{b_i}(x_i) \quad 0 \leq i < l$$

$$\sigma = (\sigma_l, \ldots, \sigma_0)$$

While the size of the signature is always $l \cdot n$, the number of evaluations of f is $\sum_{i=0}^{l-1} b_i$ and therefore the runtime of the signing operation depends on the message M.

Verification To verify a signature $\sigma' = (\sigma'_l, \ldots, \sigma'_0)$ for a message M', the message is encoded into blocks $b_l, \ldots, b_0$ as described for signing. The signature is considered valid if and only if

$$(f^{w-1-b_l}(\sigma'_l), \ldots f^{w-1-b_0}(\sigma'_0)) \stackrel{?}{=} Y$$

Again, the number of evaluations of f required for the verification depends on the message: $\sum_{i=0}^{l-1}(w-1-b_i)$. However, signing and verification require

$$\sum_{i=0}^{l-1} b_i + \sum_{i=0}^{l-1} (w-1-b_i) = l \cdot (w-1)$$

invocations of f in total for one message.

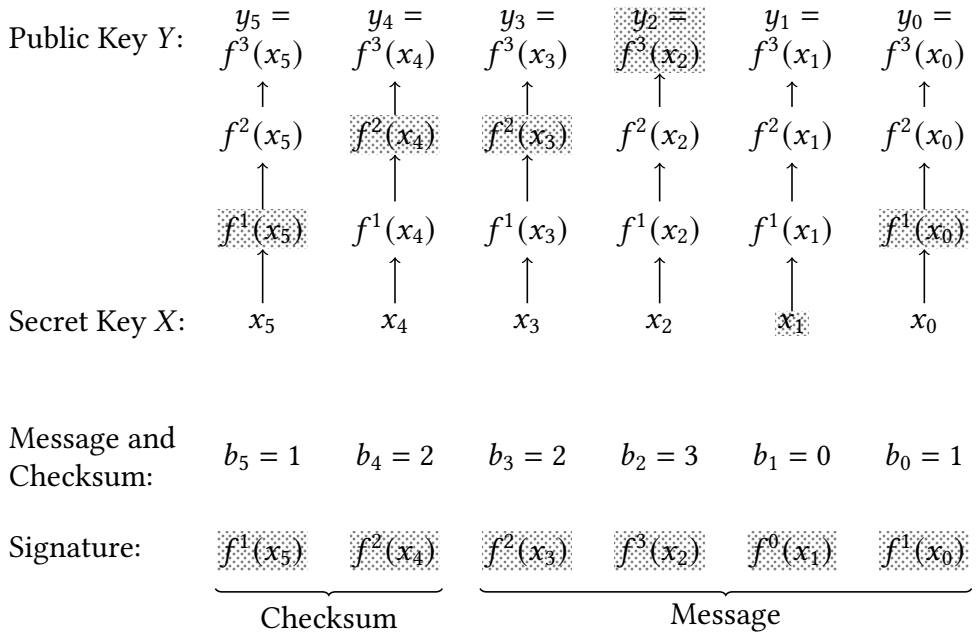


Figure 2.4.: Example of the WOTS key generation and signature for $n = 8$ and $w = 4$

Example Let $n = 8$ and $w = 4$. The message is then split into $l_1 = \lceil n/(\log w) \rceil = \lceil 8/2 \rceil = 4$ blocks. In this scenario, the checksum C can reach a maximum value of $l_1(w-1) = 4 \cdot (4-1) = 12$. Hence, $l_2 = \lfloor \log(l_1(w-1)/(\log w) \rfloor + 1 = 2$ checksum blocks are required.

Assume $d = 10110001_2 = 2301_4$. This bit string is split into $b_3 = 2$, $b_2 = 3$, $b_1 = 0$, and $b_0 = 0$. It follows that $C = (3-2) + (3-3) + (3-0) + (3-1) = 6_{10} = 0110_2 = 12_4$. Hence, $b_5 = 1$ and $b_4 = 2$. Figure 2.4 shows this scenario.

Key Extraction Figure 2.4 shows that the repeated application of f forms so-called hash chains. Each node in a hash chain has a value and the next node is derived by applying f once. The first node of a hash chain is always one of the x_i, i.e. one of the secret key elements. The chains' length is w and the last node is one of the y_i, i.e. one of the public key elements. The signature contains one element of each hash chain. Note that once a node is known, all following nodes can be calculated as this is done for verification. Hence, the verification does not only yield a true or false value, but it returns a public key Y'. This value is equal to the public key Y if (and only if) the signature is valid.

This property allows the compression of the public key: instead of using the entire $l \cdot n$ bit public key, the verifier can also use a smaller hash value of Y, for example, $h(Y)$. During verification, the extracted key Y' is hashed, and only the hash values are compared: $h(Y') \stackrel{?}{=} h(Y)$.

Domination-free encoding As noted above, once any signature element is known, all elements following a signature node in the respective hash chain can easily be calculated. In the example in Figure 2.4, these are the dotted nodes. For the security of the scheme, it

is essential that it is not possible to construct a signature for a different message from the known nodes. In WOTS, this is ensured by the addition of a checksum.

Let $P : \{0,1\}^n \rightarrow \{0,\ldots,w-1\}^l$ be the function that maps a message digest d to the hash chain positions $(b_l,\ldots,b_0)$ as described above. A vector $v = (v_{k-1},\ldots,v_0)$ dominates a vector $v' = (v'_{k-1},\ldots,v'_0)$ if for every $i \in [k]$ the condition $v_i \geq v'_i$ holds [7].

A function $Q : \{0,1\}^n \rightarrow [w]^l$ is considered domination-free if for every distinct $m_1, m_2 \in \{0,1\}^n$ the vector $Q(m_1)$ does not dominate $Q(m_2)$ [7]. Boneh and Shoup [7] prove that P as constructed above is indeed domination-free. Examples for different domination-free encodings are given in Section 4.2.3.

Security Boneh and Shoup [7] claim EUF-SMA security for WOTS as presented above. Buchmann et al. [11] present a detailed security analysis of WOTS. They propose a slightly modified scheme which uses a family of pseudo-random functions instead of a OWF and prove that this variant is EUF-SMA secure. By requiring additional properties of the function family, they are able to prove that WOTS is also strongly unforgeable.

Hülsing [34] introduces WOTS+, a variant that adds bitmasks to the chaining function and presents a an exact and tight proof that WOTS+ is SUF-SMA secure.

2.4.3. WOTS-BR

Perin et al. [57] propose two modifications of WOTS to provide faster verification at the cost of more expensive signing. There are application scenarios where the verification of a signature is used significantly more often than signing. One example is a signed device firmware that is only signed once but verified at each startup of the device. In these scenarios, it makes sense to provide the fastest possible verification and a higher signature cost is acceptable.

Both modifications are based on the observation that the sum of the cost of signing and verifying a signature is constant while the fractions vary depending on the message.

WOTS-R This variant introduces an R-tuple of nonces $\lambda = (\lambda^{(R-1)},\ldots,\lambda^{(0)})$. Instead of hashing M directly, all $d^{(r)} = g(M \parallel \lambda^{(r)})$ with $0 \leq r < R$ are computed. Each $d^{(r)}$ is split into l_1 bit strings of length $\log w$:

$$d^{(r)} = b^{(r)}_{l_1-1} \parallel \cdots \parallel b^{(r)}_0, \quad r \in [R].$$

Afterward, the nonce λ_{max} is chosen that maximizes the sum of all message blocks:

$$r_{max} = \arg\max_{r\in[R]} \sum_{i=0}^{l_1-1} b^{(r)}_i,$$

$$\lambda_{max} = \lambda_{r_{max}}.$$

The hash value $d^{(r_{max})}$ is then signed as in WOTS. The chosen nonce λ_{max} is included in the signature. A verifier can use this information and calculate $d^{(r_{max})}$ directly without testing all R nonces. The rest of the verification is the same as with WOTS.

WOTS-B This variant slightly modifies the signature checksum which is defined as $C = \sum_{i=0}^{l_1-1}(w-1-b_i)$. Due to $0 \leq C \leq l_1(w-1)$, a maximum of $n_c = \lceil \log(l_1(w-1)) \rceil$ bits are required to encode the checksum. The l_2 signature blocks can encode a total of $n_r = l_2 \cdot \log(w)$ bits. For many parameter choices, $n_r > n_c$ holds, and more bits are allocated for the checksum than are needed. Let $n_u = n_r - n_c$. The n_u unused bits are set to 0 in WOTS. As a consequence, only the $\log(w) - n_u$ lower-order bits of the first checksum block c_{l_2-1} are used. Therefore, its value is always small: $c_{l_2-1} < 2^{\log(w)-n_u}$. This means that there are always at least $w - 1 - 2^{\log(w)-n_u}$ evaluations of f required to verify the signature of the first checksum block.

To allow for faster verification, Perin et al. [57] suggest instead setting the unused bits in the first checksum block to 1. This effectively moves these evaluations of f from the verification to the signing operation. With this modification, the number of evaluations of f to verify the first checksum block is no more than $2^{\log(w)-n_u}$.

We remark that it would also be possible to shorten the hash chain for the first checksum block. Instead of w, its length should be the maximum value of the first checksum block. This has construction has two advantages over WOTS-B: It removes the unnecessary evaluations of f from the signing process and may further reduce verification cost in the case that the new chain length is not a power of two.

WOTS-BR WOTS-BR applies both modifications described above in one scheme.

Security Perin et al. [57] argue that the WOTS-B variant has no impact on the security of the scheme. Furthermore, they present a rough analysis based on the collision resistance of the underlying hash function. Bos et al. [8] give a security analysis of RapidXMSS, a XMSS variant that is built using WOTS-R.

2.4.4. WOTS+C

Kudinov et al. [46] propose a variant called WOTS+C that is primarily geared toward reducing the size of the signature. This is achieved by removing the checksum and other message blocks. To still provide a domination-free encoding, a constant sum of the message blocks is required. The message blocks to be removed have to be 0. To fulfill these conditions, the message M is repeatedly hashed with nonces.

Prerequisites A message block sum S and a number of zero-blocks z are required. Let

$$l_1 = \left\lceil \frac{n}{\log w} \right\rceil, \qquad l = l_1 - z.$$

Key Generation The key generation is analogous to WOTS, only that for the same parameters n and w, the key size l is smaller.

Signing To sign a message M, a nonce λ is selected and a hash value is computed: $d = g(M \parallel \lambda)$. The hash d is again split into l_1 bit strings of length $\log w$: $d = b_{l_1-1} \parallel \cdots \parallel b_0$. The hash is tested for the following properties:

1. $\sum_{i=0}^{l_1-1} b_i = S$,
2. $\forall i \in [z] : b_i = 0$.

If d does not fulfill both conditions, another nonce λ is chosen and tested. Once a suitable nonce was found, the signature σ is computed as:

$$\sigma_i = f^{b_{i+z}}(x_i) \quad i \in [l]$$

$$\sigma = ((\sigma_{l-1}, \ldots, \sigma_0), \lambda)$$

Verification To verify a signature σ' for a message M', the hash value $d' = g(M' \parallel \lambda)$ is calculated and split into blocks: $d' = b'_{l_1-1} \parallel \cdots \parallel b'_0$. Afterward, the verifier checks that the hash fulfills the two conditions above. If this is the case, the signature is considered valid if and only if

$$(f^{w-1-b'_{l+z}}(\sigma'_l), \ldots f^{w-1-b'_z}(\sigma'_0)) \stackrel{?}{=} Y.$$

Choice of parameters We present a detailed analysis of the effects of the choice of the parameters S and z on the scheme in Section 6.4.1.2.

Security Bos et al. [8] present security proofs for both the standalone use of WOTS+C and the use of WOTS+C as part of SPHINCS+C. In the standalone setting, they prove that WOTS+C is EUF-CMA secure if the underlying has function is multi-target extended target collision resistant [8].

2.5. Merkle Signature Scheme

The one-time signature schemes presented above have one major drawback: a new key pair is required for each signature. This means that for each message signed, a key pair needs to be exchanged in advance. This is not feasible in practice. Merkle trees present a solution to this problem: they can be used to authenticate a fixed number of one-time key pairs using one long-term public key. This is achieved by applying a binary hash tree. Below, we give a description of the Merkle Signature Scheme (MSS) based on [10].

Prerequisites Select the Merkle Tree height $h \in \mathbb{N}$, $h \geq 2$. Each MSS key pair can be used to sign up to 2^h messages. MSS requires an underlying OTS and a cryptographic hash function $g : \{0,1\}^* \rightarrow \{0,1\}^n$.

Key Generation First, 2^h key pairs (X_i, Y_i), $0 \leq i < 2^h$, for the chosen OTS are generated. Afterward, a binary hash tree is constructed. Each node of a binary hash tree has a value associated with it. In this case, this value is a bit string of length n. The value of the i-th leaf is the hash value of the public key $g(Y_i)$. The value of each inner node is calculated by hashing the values of its children.

Let $v_i[j]$ be the value of the j-th node on layer i. In this notation, the $v_0[j]$ are the leaf values and $v_h[0]$ is the value of the root node. More precisely, the node values are defined as:

$$v_i[j] = \begin{cases} g(Y_j) & i = 0 \\ g(v_{i-1}[2j] \parallel v_{i-1}[2j+1]) & i > 0 \end{cases}, \quad 0 \leq i \leq h,\ 0 \leq j < 2^{H-i}.$$

The MSS public key is the value of the root node: $Y_{MSS} = v_h[0]$. The secret key consists of the 2^h secret keys for the underlying OTS. Figure 2.5 illustrates a Merkle tree of height $h = 3$.

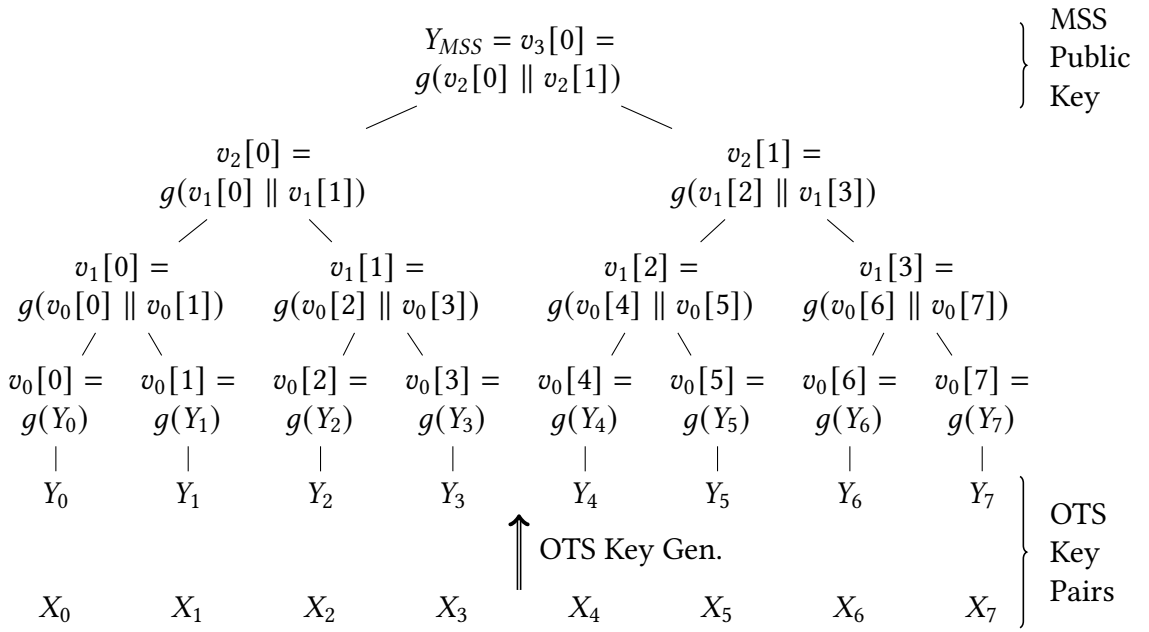


Figure 2.5.: Example of the MSS key generation for $h = 3$

Signing As a first step, an unused one-time key is selected. Let this be (X_i, Y_i). An OTS signature σ_{OTS} for the message M is generated using the selected key pair. Afterward, the so-called authentication path is calculated. It is used to prove that Y_i is part of the Merkle tree.

The authentication path consists of all sibling nodes on the path from the leaf $v_0[i]$ to the root node $v_h[0]$. Precisely, the authentication path is defined as:

$$A_s = (a_{h-1}, \ldots, a_0),$$

$$a_j = \begin{cases} v_j[s/2^j - 1] & \text{if } \lfloor s/2^j \rfloor \equiv 1 \mod 2 \\ v_j[s/2^j + 1] & \text{if } \lfloor s/2^j \rfloor \equiv 0 \mod 2 \end{cases}.$$

The MSS signature consists of four parts:

- The one-time signature σ_{OTS}
- The index of the one-time key i
- The one-time public key Y_i
- The authentication path A_s

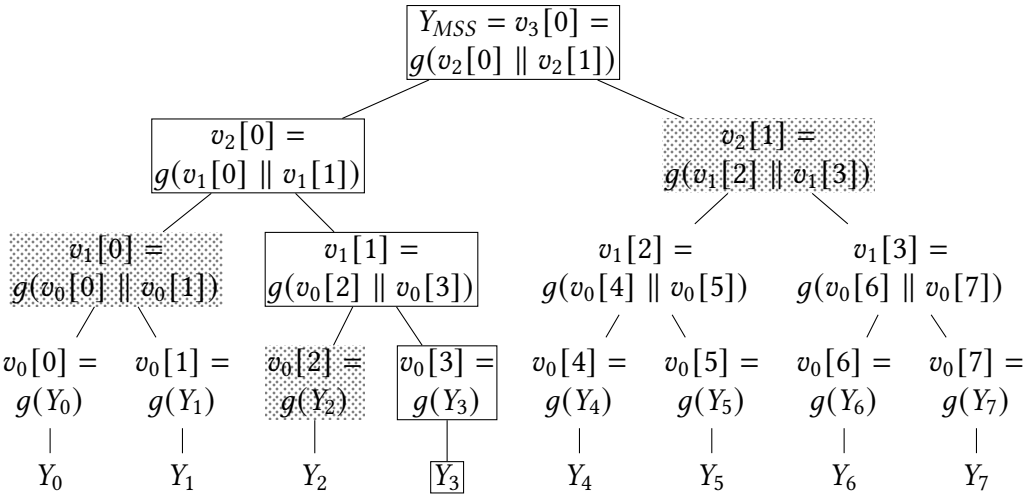


Figure 2.6.: Example of an MSS authentication path for the one-time public key Y_3

Figure 2.6 illustrates an authentication path in a Merkle tree of size $h = 3$. The framed nodes represent the path from the corresponding leaf node to the root. The dotted nodes are the elements of the authentication path. It includes the siblings of the nodes on this path. Together with the one-time public key Y_3, the authentication path provides all information required for a verifier to recalculate all nodes on this path up to and including the root node.

Verification Verification consists of two steps: first, the one-time signature σ'_{OTS} is verified with Y'_i. In the next step, the nodes on the path from the corresponding leaf to the root are re-calculated using Y'_i and the authentication path. The resulting value of the root node is then compared to the MSS public key Y_{MSS}. More specifically, the values $(p'_h, \ldots, p'_0)$ are computed where $p'_0 = g(Y'_i)$ and

$$p'_j = \begin{cases} g(a'_{j-1} \parallel p'_{j-1}) & \text{if } \lfloor s/2^{j-1} \rfloor \equiv 1 \mod 2 \\ g(p'_{j-1} \parallel a'_{j-1}) & \text{if } \lfloor s/2^{j-1} \rfloor \equiv 0 \mod 2 \end{cases}, \quad 1 \leq j \leq h.$$

The MSS signature is considered valid if (and only if)

$$p'_h \stackrel{?}{=} Y_{MSS}.$$

Key Extraction If the chosen OTS allows key extraction it is not necessary to include the one-time public key Y_i into the MSS signature. Instead, the key extraction is performed on the signature resulting in Y'_i. The path to the root node is calculated starting with $p'_0 = g(Y'_i)$. This allows for smaller signatures for OTS allowing key extraction. This is the case for WOTS, but not for LD-OTS.

State As described above, the signer has to ensure that each of the OTS key pairs is only used once. This has two consequences: firstly, each MSS can only be used for a fixed number of signatures. Secondly, this means that the signer has to keep track of which keys have already been used. This introduces a state which has to be updated with each created signature. Correct state management is essential to the security of MSS as improper state management can lead to key reuse - which breaks the security of the underlying OTS and the entire MSS.

Security MSS introduces a state that has to be updated with each signature. The definitions of digital signatures, EUF-CMA, and EUF-SMA presented in section 2.2 do not cover this case. To overcome these limitations, we extend the definitions of EUF-CMA and EUF-SMA for stateful signature schemes: The signature oracle $\mathrm{Sign}_X(\cdot)$ should keep track of the state and update it with each requested signature such that it is not possible to request signatures for two messages starting from the same state.

Coronado [15] proves that MSS is EUF-CMA secure under the assumption that the underlying OTS is EUF-SMA secure and the hash function is collision-resistant.

Pseudo-Random Key Generation The secret key for MSS consists of the one-time secret keys and the current state. As mentioned above, one one-time key has to be used for each message to be signed in the MSS. For long-living keys, this means that numerous OTS keys are required. In practice, $h \geq 15$ is not uncommon. This however leads to large storage requirements to store the OTS keys if they are chosen at random.

Instead, the OTS keys may be pseudo-randomly generated from a significantly smaller seed. This leads to considerably lower storage requirements for the signer [10].

Tree Traversal During the key generation, all nodes in the tree must be computed. During a key's lifetime, each node is part of at least one authentication path. However, it is not feasible to store all nodes of a tree as this increases the key size linearly in the number of signatures and therefore exponentially in h. Not storing any nodes is however also problematic as each signature will require about as much computation as the key generation.

To address this problem, several tree traversal algorithms have been proposed. For an overview, refer to [10]. In general, they extend the state to also include certain tree nodes

which are computed and discarded with each signature. The current state of the art is the so-called BDS algorithm [10].

2.6. Hypertrees

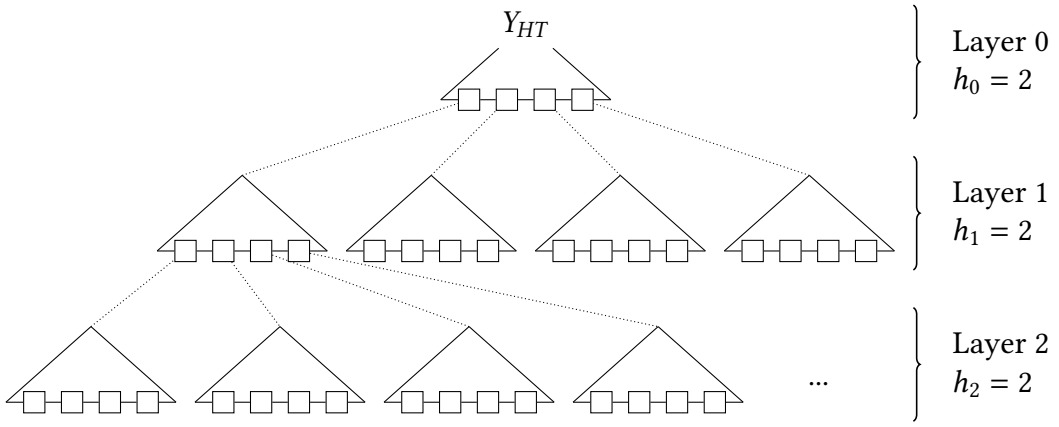


Figure 2.7.: Example of a hypertree structure with layer count $d = 3$ and $h_0 = h_1 = h_2 = 2$.

Hypertrees, also called Multitrees or Tree Chaining, are a technique to achieve keys for a large number of signatures at a greatly decreased key generation cost. However, this comes at the cost of slower signing and verification operations and larger signatures. Instead of using a single large MSS tree that has to be traversed entirely when a key is generated, a hypertree is a tree that consists of MSS trees. The intermediate trees are used to sign the root of a tree on the next lower level. The trees at the lowest level are used to sign messages. Buchmann et al. [10] present the Chained Merkle Signature Schemes (CMSS), a simple signature scheme based on a hypertree.

Let d be the number of layers of the hypertree. The layer i of the hypertree consists of a certain number of MSS trees of height h_i. In total, the hypertree can be used to sign $2^{h_0+\cdots+h_{d-1}}$ messages. There is exactly one MSS tree on layer 0. On layer $i > 0$, there are $2^{h_0+\cdots+h_{i-1}}$ trees.

The public key of the hypertree Y_{HT} is the value of the single root node of the single MSS tree on layer 0. Figure 2.7 shows an example of the hypertree structure for a $d = 3$ and $h_0 = h_1 = h_2 = 2$. Each of the triangles represents an MSS tree of height 2 and can be used to create up to four MSS signatures.

To sign a message M using a hypertree scheme, an unused leaf Y_{d-1} of an MSS tree on the bottom layer $d - 1$ is selected. The message M is signed using this key resulting in the OTS signature σ_{d-1}. As in MSS, the authentication path A_{d-1} to the root of the MSS tree r_{d-1} is constructed. However, the root of the tree on the bottom layer is not known to a verifier.

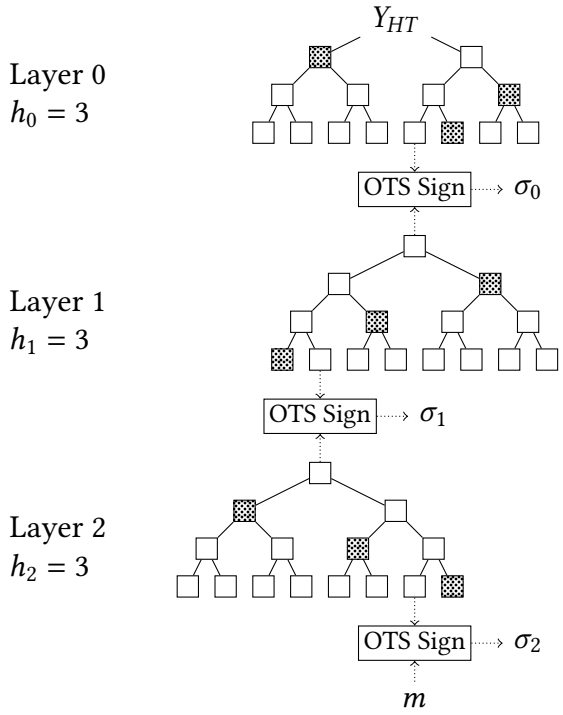


Figure 2.8.: Example of a hypertree signature with $d = 3$ and $h_0 = h_1 = h_2 = 3$. Dotted notes represent the authentication paths.

Hence, r_{d-1} is signed using the corresponding leaf of the tree on the layer above Y_{d-2}. This yields the signature σ_{d-2}, the authentication path A_{d-2}, and the root of this tree r_{d-2}. This process is repeated until a signature was created using the tree on layer 0. Its root r_0 is known to the verifier as it is the public key of the hypertree scheme Y_{HT}. The complete hypertree signature is

$$\begin{aligned}\sigma_{HT} = (&\sigma_{d-1}, Y_{d-1}, A_{d-1},\\ &\sigma_{d-2}, Y_{d-2}, A_{d-2},\\ &\ldots,\\ &\sigma_0, Y_0, A_0).\end{aligned}$$

If the chosen OTS allows key extraction, the Y_i can be reconstructed by the verifier from the message or root of the tree below. In this case, it is not necessary to include them in the signature.

Verification is straightforward: The message is hashed and verified using Y'_{d-1}. If the scheme supports key extraction and Y'_{d-1} is not transferred, it is extracted from the signature and the message. Using Y'_{d-1} and A'_{d-1}, r'_{d-1} is computed. It is used to compute r'_{d-2} in the same manner. This process is repeated for each layer until r'_0 is known. The signature is considered valid if and only if $r'_0 \stackrel{?}{=} Y_{HT}$.

Coronado [15] claims EUF-CMA security for CMSS with two layers. However, this result can also be applied to the generalized setting with multiple layers.

2.7. Hash-Based Few-Time Signatures

In this section, we introduce three Few-Time Signature Schemes (FTSs). Unlike in OTSs, a FTS key may be used to sign multiple messages. However, the security level decreases with each signature.

2.7.1. HORS Signature Scheme

Reyzin and Reyzin [60] propose the signature scheme Hash to Obtain Random Subset (HORS). In contrast to WOTS and its variants, HORS is a so-called Few-Time Signature Scheme (FTS). This means that each HORS key can be used for up to r signatures (for small r).

Prerequisites HORS requires a one-way function f and parameters k and t with $k < t$.

Additionally, HORS requires an encoding function $H : \{0,1\}^* \to \mathbb{P}_k([t])$. This encoding function maps a message of arbitrary length to a set of integers. The set is not empty and has a maximum of k elements which are all integers in $[t]$.

Informally, Reyzin and Reyzin [60] describe the security requirements for H as follows: It should be infeasible to find messages $m_0, \ldots, m_r$ such that $H(m_0) \subseteq H(m_1) \cup \cdots \cup H(m_r)$. We abstain from presenting the formal definition of subset-resilience. This property allows r messages to be signed with one key.

Key Generation The secret key X consists of t bit strings of length n which are chosen at random:

$$X = (x_{t-1}, \ldots, x_0) \xleftarrow{\$} \{0,1\}^{(n,t)}.$$

The public key Y is then derived by applying the one-way function f to each element:

$$y_i = f(x_i) \quad i \in [t],$$

$$Y = (y_{t-1}, \ldots, y_0).$$

Signing The message M is encoded using $s = H(M)$. Let $s_{j-1}, \ldots, s_0$ be the members of s in a certain order (for example, in ascending order).

As a signature, the secret key elements specified by s are revealed:

$$\sigma_i = x_{s_i} \quad i \in [j],$$

$$\sigma = (\sigma_{j-1}, \ldots, \sigma_0).$$

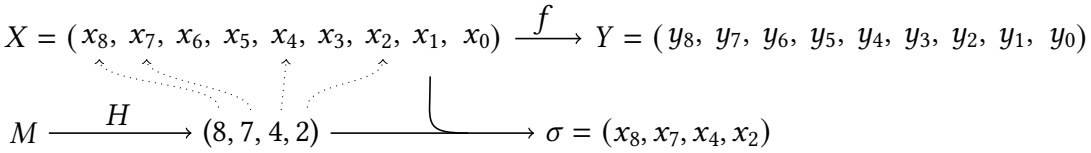


Figure 2.9.: Example of a HORS key and signature with $k = 4$ and $t = 10$

Verification Verification of a signature $\sigma' = (\sigma'_{j'-1}, \ldots, \sigma'_0)$ is straightforward. As above, let $\{s'_{j''-1}, \ldots, s'_0\} = H(M')$. If $j' \neq j''$ abort. The signature is considered valid if and only if:

$$f(\sigma'_i) = Y_{s'_i} \quad \forall i \in [j'].$$

Example Suppose $k = 4$ and $t = 9$. To generate a key, 9 random bit strings are sampled and hashed to retrieve the public key. Figure 2.9 depicts this scenario. Suppose for a given encoding function H and a message M to be signed, we have $H(M) = \{8, 7, 4, 2\}$. The signature for this message is $\sigma = (x_8, x_7, x_4, x_2)$.

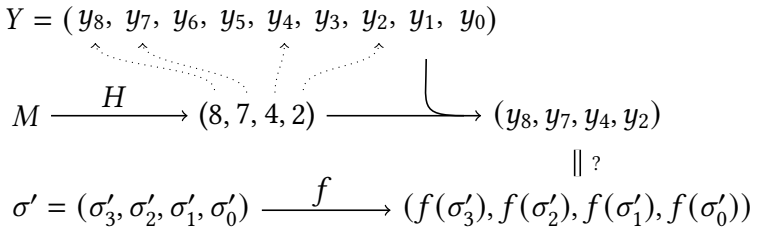


Figure 2.10.: Example of a HORS signature verification with $k = 4$ and $t = 9$

The verification process for this message and a signature $\sigma' = (\sigma'_3, \sigma'_2, \sigma'_1, \sigma'_0)$ is illustrated in Figure 2.10. Each of the signature elements is hashed using f and the result is compared to the tuple $(y_8, y_7, y_4 s, y_2)$.

Construction of H Reyzin and Reyzin [60] present the following construction for an encoding function H. It requires a hash function g with an output length of n where $n = k \cdot \log t$. For a message M, divide $d = g(M)$ into k bit strings of length $\log t$:

$$d = b_{k-1} \| \cdots \| b_0$$

Interpret each b_i as an integer in $[t]$ and return $\{b_{k-1}, \ldots, b_0\}$. Note that this set is not-empty and has k or fewer elements.

Reyzin and Reyzin [60] give no proof that this construction does indeed fulfill the required property in the standard model.

Runtime and sizes Key generation requires t evaluations of f. Singing requires no evaluation and verification k evaluations. The public key size is $t \cdot n$ and signatures have a size of $k \cdot n$. For a given n, the choice of t and k allows for a trade-off between the size and generation time of keys and signatures.

Reyzin and Reyzin [60] suggest the following parameters for $n = 160$: $k = 16$, $t = 2^{10} = 1024$ or $k = 20$, $t = 2^8 = 256$. Even for the second choice, the public key consists of 256 bit strings of length n. This is significantly larger than WOTS.

HORS also does not allow key extraction. This means that for use in a Merkle tree, the HORS public key must be included in the Merkle tree signature. This leads to comparably large signatures.

2.7.2. HORST Signature Scheme

Bernstein et al. [4] propose HORS with Trees (HORST) to address the limitations of HORS described above. It was introduced as part of the signature scheme SPHINCS.

Their main improvement comes from building a binary hash tree on top of the secret key elements, similar to a Merkle tree. Only the root of this tree is used as the public key (instead of all leaves). Each signature element has an accompanying authentication path that authenticates the path from the corresponding leaf to the root.

In the following, we give a slightly simplified definition of HORST that is intended to illustrate the structure of the scheme. Therefore, we omit the bitmasks for randomized hashing used by Bernstein et al. [4].

Prerequisites As above, let f be a one-way function and g a cryptographic hash function with output length n. HORST requires parameters k and $t = 2^\tau$ for some $\tau \in \mathbb{N}_+$ with $k\tau = n$. Select $z \in \mathbb{N}_+$ such that $k(\tau - z + 1) + 2^z$ is minimal. If this is the case for two successive values, the larger one is chosen.

Key Generation The secret key X consists of t bit strings of length n. Bernstein et al. [4] chose them pseudo-randomly. For the sake of simplicity, we present them as randomly sampled:

$$X = (x_{t-1}, \ldots, x_0) \xleftarrow{\$} \{0,1\}^{(n,t)}.$$

The one-way function f is applied to each of the secret key elements:

$$y_i = f(x_i) \quad i \in [t].$$

Afterward, a binary hash tree of height $\log t$ is constructed that has the y_i as leaves. We omit a formal definition as this is analogous to the one on section 2.5. Let Y be the root of this tree which is used as the public key.

Signing To sign a message M, divide its hash $d = H(M)$ into k bit strings of length $\log t$: $d = b_{k-1} \parallel \cdots \parallel b_0$. Let $A^{(i)} = (A^{(i)}_{\log(t)-1}, \ldots, A^{(i)}_0)$ be the authentication path for y_i in binary hash tree described above. The signature element σ_i is defined as the b_i-th secret key element with the first $\log(t) - z$ elements of its authentication path:

$$\sigma_i = (x_{b_i}, (A^{(b_i)}_{\log(t)-1-z}, \ldots, A^{(b_i)}_0)) \quad i \in [k].$$

Additionally, let σ_k contain all nodes of the binary hash tree on layer $\log(t) - z$: $\sigma_k = (v_{\log(t)-z}[0], \ldots, v_{\log(t)-z}[2^z - 1])$.

$$\sigma = (\sigma_k, \ldots, \sigma_0)$$

Verification Compute $b'_{k-1}, \ldots, b'_0$ as described above. Then for each $i \in [k]$ calculate the corresponding node on layer $\log(t) - x$ above the leaf with index b'_i using $f(\sigma'_i)$ and the partial authentication path as given in the signature. The result is compared to the value provided in σ'_k. If they are equal for all i, the elements of σ'_k are used to compute the root of the binary hash tree Y'. The signature is considered valid, if and only if $Y' \stackrel{?}{=} Y$.

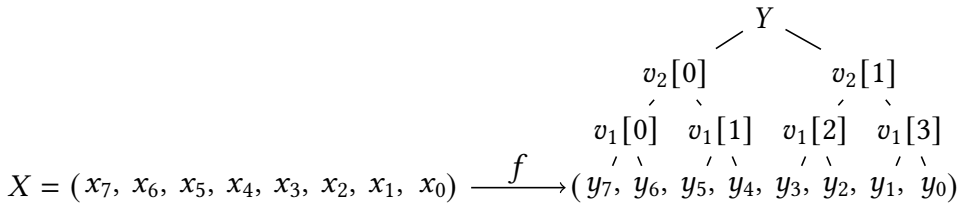


Figure 2.11.: Example of a HORST key with $n = 9$, $t = 8$, and $k = 3$

Example Suppose $n = 9$, $t = 8$ and $k = \frac{n}{\log t} = 3$. With these parameters, the term $k(\log(t) - z + 1) + 2^z$ is minimal for $z = 2$. The key generation is illustrated in Figure 2.11. Eight bit strings are sampled and hashed. From the hashed values, a binary hash tree of height $\log t = 3$ is constructed. Its root Y is used as the public key.

Figure 2.12 shows the signature generation process. Because of $z = 2$ all nodes on layer one will be included in $\sigma_3 = (v_1[0], v_1[1], v_1[2], v_1[3])$. Hence, the authentication path for each leaf will only have a length of $\log t - z = 1$. This means that for this choice of parameters, the authentication path only consists of the respective neighboring leaf. This is independent of the message to be signed.

Assume $g(M) = 110011001$. This hash value is split into 3 blocks of length 3. Interpreting the blocks as integers yields $(6, 3, 1)$. The leaf y_6 has the authentication path $A^{(6)} = (y_7, v_2[1], v_1[1])$. Because of $b_2 = 6$, we have $\sigma_2 = (x_6, (y_7))$. The other signature elements σ_1 and σ_0 are computed in the same way. With all elements known, the signature can be assembled: $\sigma = (\sigma_k, \ldots, \sigma_0)$.

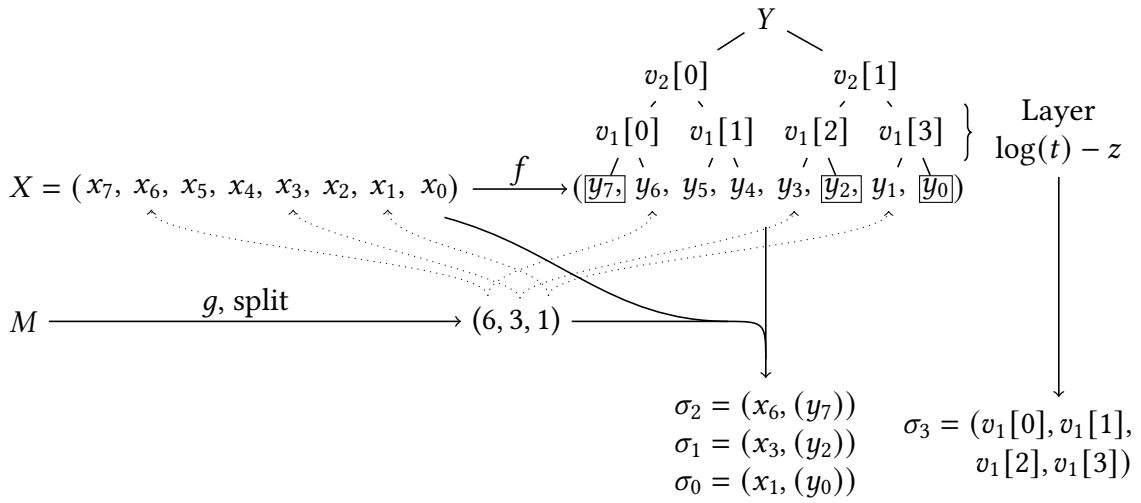


Figure 2.12.: Example of a HORST signature with $n = 9$, $t = 8$, $k = 3$, and $z = 2$

Sizes The secret key has a size of $t \cdot n$, but this can be trivially reduced by using a pseudo-random key generation. The public key consists of a single value of size n. Each signature consists of k secret key elements, their corresponding authentication paths of length $\log(t) - z$, and the 2^z nodes on layer $\log(t) - z$. In total, a signature's size is $(k(\log(t) - z + 1) + 2^z)n$ bits.

Note that depending on the message, there may be redundant information in the signature. If $b_i = b_j$ holds, then $\sigma_i = \sigma_j$, and one of the elements could be omitted in the signature. Additionally, the authentication paths for two included leaves may merge. In this case, the last elements of the authentication path will be redundant in the signature. These observations allow compressing the signature further.

Key Extraction In contrast to HORS, HORST allows key extraction from the signature. This makes HORST a good candidate for signature schemes based on Merkle trees. Its signature size is larger than HORS, but this is more than compensated by the key extraction capability.

2.7.3. FORS Signature Scheme

Bernstein et al. [5] present the Forest of Random Subsets (FORS) signature scheme as an improvement upon HORST. Instead of using the b_i as indices into one tree, FORS uses k trees and each b_i is used as an index into one tree.

Prerequisites As for HORS, the parameter x is not needed.

Key Generation Choose kt bit strings of length n as the secret key grouped into sets of t values:

$$X^{(i)} = (x_{t-1}^{(i)}, \ldots, x_0^{(i)}) \xleftarrow{\$} \{0,1\}^{(n,t)} \quad i \in [k],$$
$$X = (X^{(k-1)}, \ldots, X^{(0)}).$$

For each $X^{(i)}$, construct a binary hash tree. Let $Y^{(i)}$ denote its root. Then compute the public key:

$$Y = g(Y^{(k-1)}, \ldots, Y^{(0)})$$

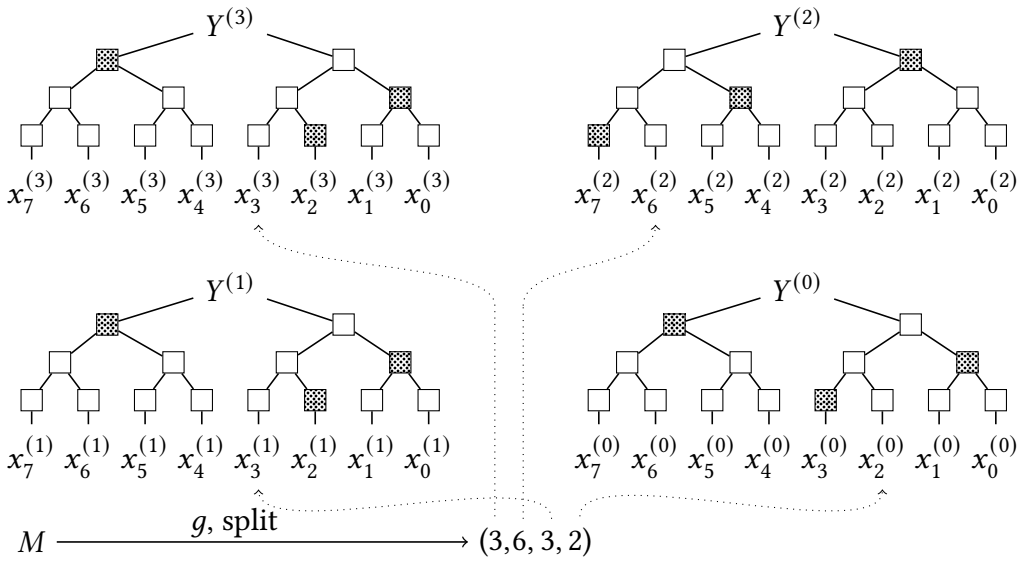


Figure 2.13.: Example of a FORS key and signature with $k = 4$ and $t = 8$

Signing The message hash $d = g(M)$ is split into $b_{k-1}, \ldots, b_0$ as above. Let $A_i^{(j)}$ be the authentication path of the node with index i in the hash tree j. The signature is defined as:

$$\sigma_i = (x_{b_i}^{(i)}, A_{b_i}^{(i)}) \quad i \in [k],$$
$$\sigma = (\sigma_{k-1}, \ldots, \sigma_0).$$

Figure 2.13 shows the structure of a FORS key and the signature for a message m. The dotted nodes in the trees represent the authentication path of the respective leaf used in the signature.

Verification For each tree i, the root Y_i' is recomputed using the index b_i', the hashed secret key element $f(x_{b_i}'^{(i)})$ and the provided authentication path $A_{b_i}'^{(i)}$. The signature is considered valid, if and only if

$$Y \stackrel{?}{=} g(Y'^{(k-1)}, \ldots, Y'^{(0)}).$$

Improvement over HORST One of the shortcomings of HORS is that, in extreme cases, a signature may only comprise a single secret key element. This is the case, if all message blocks are equal. If this element is known from a previous signature, a signature for such a message can be easily forged. In FORS, this is not possible as even if all the message blocks are equal, different secret key elements are required for the signature [3].

3. Specification and Standardization

While the previous chapter presented theoretical constructions for one- and few-time signature schemes, hash functions, and Merkle tree constructions, this chapter presents concrete signature schemes and hash functions that are used in practice. It is structured as follows: First, we present three hash-based signature schemes and how they are built from the constructions presented in the previous chapter. Secondly, we present the cryptographic hash functions that are used to implement hash-based signatures. Lastly, we close the gap between both and describe how the cryptographic hash functions are used to implement the functions required in the signature schemes.

3.1. Signature Schemes

This section introduces three fully specified HBSs for practical use: XMSS, LMS, and SPHINCS$^+$. We describe their structure and how they relate to the constructions in the previous chapter.

3.1.1. XMSS

The first signature scheme with the name eXtended Merkle Signature Scheme (XMSS) was proposed in 2011 by Buchmann et al. [9]. It uses the Merkle tree construction with the WOTS one-time signature. In the following, we describe the main differences between XMSS and the standard MSS as presented in Section 2.5 [9]:

Pseudo-Random Key Generation The WOTS keys are pseudo-randomly generated from a single seed. This effectively reduces the secret key to just this seed.

Bitmasks in Tree The nodes in the hash tree are defined as $v_i[j] = g((v_{i-1}[2j] \oplus b_l[i]) \parallel (v_{i-1}[2j+1] \oplus b_r[i]))$. The bitmasks $b_l[i]$ and $b_r[i]$ are bitstrings of length n that are randomly sampled and are fixed for each layer and part of the public key. This change allows dropping the requirements to g from collision resistance to second-preimage resistance.

L-Trees In MSS, a WOTS secret key $Y_i = (y_{i,l-1}, \ldots, y_{i,0})$ is compressed as $v_0[i] = g(y_{i,l-1} \parallel \cdots \parallel y_{i,0})$. In XMSS, a so-called L-Tree is applied. An L-Tree is a binary hash tree that has the leaf values $y_{i,l-1}, \ldots, y_{i,0}$. However, l is not necessarily a power of two. To construct a hash tree, all nodes without a right sibling are lifted until they become the right sibling of another node. The L-Tree also uses bitmasks as described above.

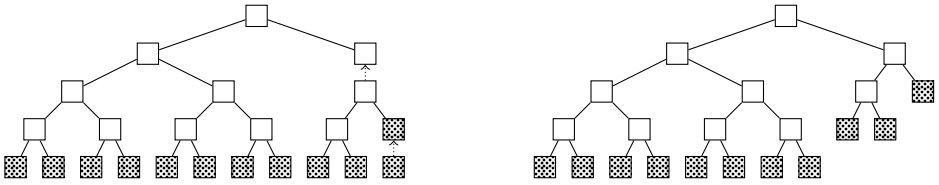


Figure 3.1.: Example of an L-Tree with 11 nodes. Construction (left) and resulting tree (right). Dotted squares represent leaf nodes, dotted arrows represent the lifting of a node.

Public Key The public key contains the root node of the Merkle tree. Additionally, it contains all bitmasks used in the Merkle tree and the L-Trees.

Security Buchmann et al. [9] prove that XMSS is EUF-CMA if it is instantiated with a second-preimage resistant function family and a pseudorandom function family, and argue that these requirements are minimal. That is, the existence of any digital signature scheme implies the existence of both a second-preimage resistant function family and a pseudorandom function family. This means that as long as any digital signature scheme exists, a secure instantiation of XMSS exists. The security requirements of XMSS are minimal in this sense.

RFC 8391 In 2018, Hülsing et al. published RFC 8391 [36]. In this standard, a signature scheme also called XMSS is fully specified for implementation. However, this RFC does not standardize XMSS as described above. Instead, it also includes further improvements originally published in [34, 35]. The main differences between XMSS as in [36] and XMSS as in [9] are:

Domain Separation Instead of using two functions f and g as MSS, RFC 8391 uses four domain-separated functions: F for Winternitz chaining, H for tree node calculation, H_{msg} for the initial message hash, and PRF for the pseudo-random generation of bitmasks and prefixes. Nevertheless, all functions may be implemented using the same underlying cryptographic hash function (see subsection 3.3.1).

Addressing RFC 8391 introduces an addressing scheme: Each hash function invocation has a unique address assigned to it. This address only depends on the position of the hash in the XMSS structure, but not on keys or messages. For example, the address to compute the next element of a Winternitz chain includes the index of the WOTS key in the Merkle tree, the number of the chain in the key, and the current position in that chain, because the hash invocation can be uniquely identified by this information.

Bitmasks and Prefixes For each invocation of F and H, a prefix and bitmask are used. Both are pseudo-randomly derived from a public seed using PRF and the address. This leads to a further strengthening of the scheme.

For the remainder of this thesis, we will use XMSS to refer to the signature scheme specified in RFC 8391 [36].

Hypertree RFC 8391 [36] also specifies a hypertree variant of XMSS called eXtended Merkle Signature Scheme Multi-Tree ($XMSS^{MT}$). Hypertree constructions are presented in section 2.6.

Parameter Choice RFC 8391 [36] specifies the Winternitz parameter as $w = 16$. The security parameter n depends on the chosen hash function and is either 256 or 512 bits. In Special Publication (SP) 800-208 [14], NIST published additional parameter sets which use $n = 256$ or $n = 192$. The Merkle tree height h is either 10, 16, or 20. This means that the largest XMSS tree supports up to 2^{20} signatures.

3.1.2. LMS

The Leighton-Micali Signature Scheme (LMS) was first published by Leighton and Micali in 1995 as a patent [48]. In 2019, McGrew et al. published RFC 8554 [49] that specifies LMS for implementation building on this patent.

Similar to XMSS, LMS uses an OTS in combination with a Merkle tree. The OTS is called LM-OTS, but in practice this scheme is equivalent to WOTS. Overall, the high-level structure of XMSS and LMS are the same. In the following, we describe the main differences between LMS and XMSS or MSS:

Functions LMS only requires one second-preimage resistant function H and no pseudo-random function. Unlike XMSS, there is no explicit domain separation. Implicit domain separation is achieved by the input formats of H (see subsection 3.3.2).

Addressing and Bitmasks There is no explicit addressing in LMS. However, the position of each hash evaluation is encoded in the input data for the hash function.

Prefixes There are no pseudo-random bitmasks or prefixes in LMS. Instead, a constant prefix 16-byte prefix I is used for all hash function calls.

OTS Key Compression As in MSS, the WOTS public key is compressed using one hash function call. MSS does not use L-Trees.

Security Katz [39] provides a security analysis of LMS under the assumption that the underlying hash function behaves like a random oracle. Fluhrer [23] presents another concrete security analysis which assumes that the underlying hash function is based on a Merkle-Damgård construction and its compression function behaves like a random oracle. In contrast to XMSS, there is no security analysis in the standard model and all existing results rely entirely on the random-oracle model.

Parameter Choice McGrew et al. [49] differentiate between n, the hash output length used in LM-OTS, and m, the hash output length used in the Merkle tree. As it is hard to find a scenario in which chosing $n \neq m$ is reasonable, we assume $n = m$ for the rest of this thesis for simplicity and comparability with XMSS. RFC 8554 [49] only specifies parameter sets with $n = m = 32$. SP 800-208 [14] additionally defines parameter sets with $n = m = 24$.

Additionally, LMS requires a Winternitz parameter w with $w \in \{2, 4, 16, 256\}$. Note that RFC 8554 [49] does not use w as defined in this thesis, but instead its base-2 logarithm.

LMS supports a Merkle tree height h of 5, 10, 15, 20, or 25. The smallest LMS key can be used for up do $2^5 = 32$ signatures, while the largest size supports up to $2^{25} = 33554432$ signatures.

Hypertree RFC 8554 [49] also specifies a hypertree variant of LMS, called Hierarchical Signature Scheme (HSS). It applies the generic hypertree construction described in section 2.6.

3.1.3. SPHINCS$^+$

Unlike XMSS and LMS, SPHINCS$^+$ is a stateless signature scheme. Like traditional digital signature schemes, the private key does not change over time and no additional state is needed. We recall that the purpose of the state in a stateful HBS is to keep track of which OTS keys were already used. As illustrated in Section 2.4.1, key reuse can completely break the security of an OTS.

The main idea behind transforming stateful into stateless HBS is to use a large number of OTS keys such that the probability of key reuse is negligible. This was proposed by Goldreich [25, Section 6.4.2.3]. For a security parameter n, the author presents an authentication-tree structure with 2^n OTS keys for signing documents. The leaf is either chosen at random or pseudo-randomly depending on the message to achieve a deterministic signing algorithm.

Bernstein et al. [4] elaborate on the idea of pseudo-random leaf selection and suggest the usage of a CRHF H with an output length of n bits. To sign a message M, a hash value $h = H(M)$ is computed, interpreted as an integer in $[2^n]$ and used as the leaf index. This reduces the problem of OTS key reuse to the collision resistance of H.

However, it is not feasible to construct a Merkle tree of height n for usual values of n. A different approach is required to authenticate the chosen OTS key. Goldreich [25] suggest a binary authentication tree. Each node contains an OTS key that is used to sign the concatenated OTS public keys of its children. The signature contains an OTS key and a signature of this key for every of the n layers of the tree. This leads to large signature sizes. For common parameters, the signature size is larger than 1 MB [4].

Another possibility is the use of a hypertree, as proposed in [4] as part of the stateless signature scheme SPHINCS. Additionally, SPHINCS does not use an OTS to sign messages,

but the FTS HORST instead (see Section 2.7.2). This significantly reduces the security impact of key reuse. Hence, the tree height can be reduced without lowering the security level.

Building on SPHINCS, Bernstein et al. [5] propose SPHINCS$^+$. The scheme was submitted to the NIST Post-Quantum Cryptography (PQC) competition [51]. Therefore, an exhaustive specification of SPHINCS$^+$ [3] exists.

Structurally, SPHINCS and SPHINCS$^+$ are quite similar. Both use a hypertree whose OTS leafs are used to sign FTS keys which in turn are used to sign the messages. Figure 3.2 illustrates the structure of SPHINCS and SPHINCS$^+$.

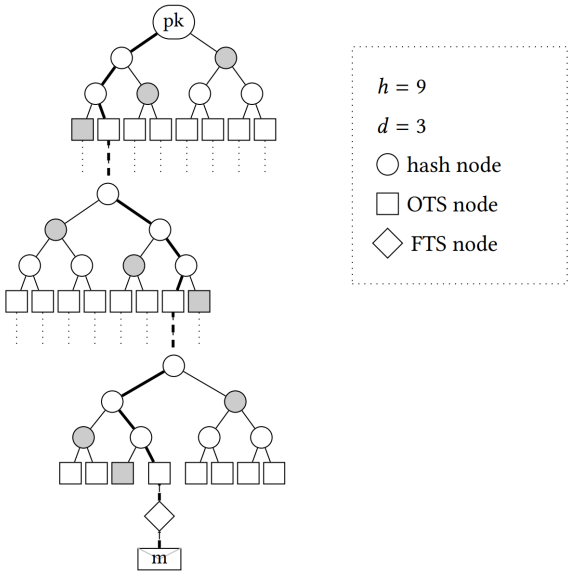


Figure 3.2.: Example of the structure of SPHINCS and SPHINCS$^+$. Source: [5].

In the following, we describe the main differences between SPHINCS and SPHINCS$^+$. For an exhaustive list, see Aumasson et al. [3].

FTS SPHINCS$^+$ uses FTS FORS instead of HORST. We briefly describe the advantages of FORS over HORST in Section 2.7.3.

OTS SPHINCS$^+$ uses a WOTS+ as proposed by Hülsing [34]. OTS keys are compressed with one hash call, SPHINCS$^+$ does not use L-Trees.

Hypertree The SPHINCS$^+$ hypertree is essentially a fixed input-length variant of XMSS^{MT} [3].

Verifiable leaf selection While the verifier cannot verify the leaf choice in SPHINCS, SPHINCS$^+$ provides a publicly verifiable leaf selection. This further strengthens the scheme [5].

Parameter Selection SPHINCS$^+$ defines parameter sets for three different security parameters: $n \in \{128, 192, 256\}$. For each possible security parameter, there are two different SPHINCS$^+$ parameter sets. One is optimized for small signatures (suffix `s`) and the other is optimized for fast signing (suffix `f`). Both provide roughly the same level of security [3, Table 3].

3.2. Hash Functions

In this section, we present three hash function families that are used to instantiate the signature schemes introduced in the previous section.

3.2.1. SHA-2

SHA-2 is a family of hash functions specified by NIST in the Federal Information Processing Standard (FIPS) 180-4 [17]. It consists of the hash functions SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. We focus on SHA-256 and SHA-512 because only these are used in the signature schemes covered in this thesis. Additionally, the other hash functions are structurally equivalent to SHA-256 or SHA-512 and only differ in the chosen initialization vector and a truncation of the final hash value.

Both SHA-256 and SHA-512 use a Merkle-Damgård construction with a compression function. SHA-256 uses a block size of 512 bits (64 bytes) and works on an internal state of 256 bits (32 bytes). The size of the hash value is also 256 bits (32 bytes). For SHA-512, all numbers are doubled: the block size is 1024 bits (128 bytes), and states and hash values are 512 bits (64 bytes) [17].

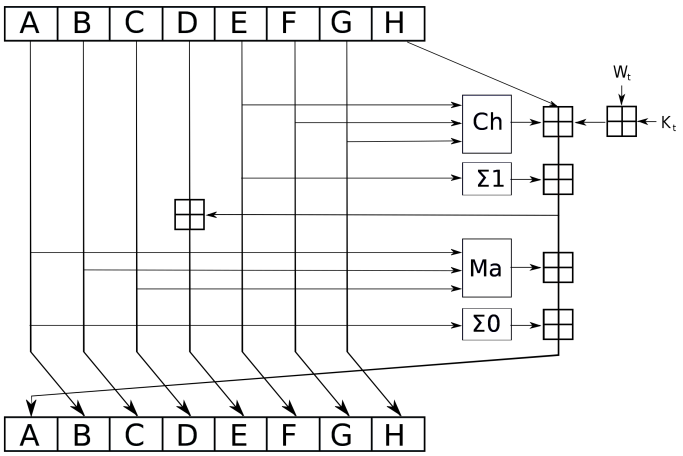


Figure 3.3.: SHA-2 compression function (one round). Source: [41].

Message Padding Before a message is split into block-sized pieces, it has to be padded to ensure that the message length is a multiple of the block size. Generally, the padding consists of a 1-bit, a variable number of 0-bits, and the message length in bits as a 64-bit (SHA-256) or 128-bit (SHA-512) integer. The number of 0-bits is chosen minimally such that the total length is a multiple of the block length [17].

Merkle-Damgård Processing Afterward, the message is split into blocks of the respective block length. Starting from an initial state, the state is updated for each message block using the compression function as described in subsection 2.3.4. The resulting state is used as the hash value.

Compression Function The compression functions for SHA-256 and SHA-512 are structurally the same and only differ in the word size they operate on. The function works in two phases: the message schedule phase and the state update phase [22]. In the first phase, the message block is scheduled, that is, split and extended into 64 words $W_0, \ldots, W_{63}$. The state update phase consists of 64 iterations. The structure of one such iteration is illustrated in Figure 3.3. Initially, the working variables $A, \ldots, H$ are initialized with the current state. The i-th iteration incorporates the scheduled message word W_i and the round constant K_i. After 64 iterations, the working variables are added to the state [22].

3.2.2. SHA-3

SHA-3 is another family of hash functions specified by NIST and intended as a successor for SHA-1 and SHA-2. It is specified in FIPS 202 [19] and is a subset of the Keccak family [6]. Unlike its predecessors, SHA-3 is not based on a Merkle-Damgård construction, but on a sponge construction as presented in subsection 2.3.5. We focus on the Extendable-Output Functions (XOFs) SHAKE128 and SHAKE256 as only these are used in the schemes covered in this thesis.

All SHA-3 hash functions are based on the Keccak-p[1600, 24] permutation, which operates on bit strings of $n = 1600$ bits. SHAKE128 uses a capacity $c = 256$ and SHAKE256 uses $c = 512$ [19]. The resulting rates are $r = 1344$ and $r = 1088$, respectively. Recall that, for sponge constructions, the rate r is the block size that is absorbed for each call of the permutation.

Message Padding All SHA-3 hash functions use a 10∗1 padding scheme. An additional suffix is appended to the message for domain separation before the padding. For SHAKE128 and SHAKE256, this is 1111 [19]. In total, the padding pattern is 111110*1.

3.2.3. Haraka

The Haraka v2 hash function was proposed by Kölbl et al. in 2017 [45]. Traditional hash functions like SHA-2 and SHA-3 are geared towards optimal performance on larger inputs. However, in hash-based signature schemes, input sizes are generally rather small (often less than 128 bytes). Haraka was specially designed to perform well on small inputs.

Kölbl et al. [45] specify two permutations as part of the Haraka family:

$$\pi_{256} : \{0,1\}^{256} \rightarrow \{0,1\}^{256},$$
$$\pi_{512} : \{0,1\}^{512} \rightarrow \{0,1\}^{256}.$$

The two hash functions are derived from these permutations using a Davies-Meyer construction:

$$\text{Haraka256} : \{0,1\}^{256} \rightarrow \{0,1\}^{256},\ x \mapsto \pi_{256}(x) \oplus x,$$
$$\text{Haraka512} : \{0,1\}^{512} \rightarrow \{0,1\}^{256},\ x \mapsto \text{trunc}_{\text{H}}(\pi_{512}(x) \oplus x)$$

where $\text{trunc}_{\text{H}} : \{0,1\}^{512} \rightarrow \{0,1\}^{256}$ is a truncation function specified in [45].

Permutations The permutations are iterated constructions and built upon the Advanced Encryption Standard (AES) round function specified in FIPS 197 [20]. The AES round function operates on 128 bits (32 bytes) of data and requires a round key of 128 bits. For Haraka, the round keys are replaced with fixed round constants.

Each Haraka round consists of two rounds of AES and a permutation. To cover the Haraka block of 256 or 512 bits, the AES round function is used two or four times, respectively. The following permutation permutes words of 32 bits. Figure 3.4 illustrates the structure of the Haraka round function.

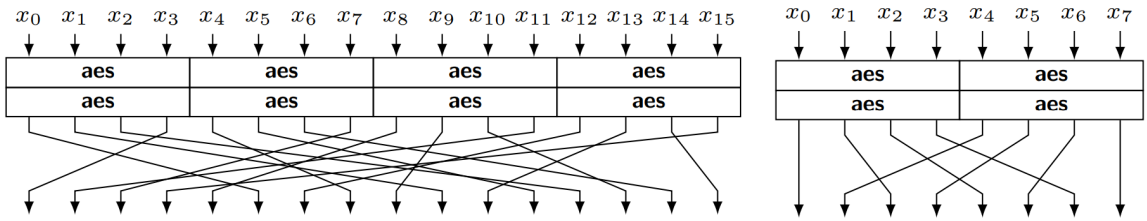


Figure 3.4.: Illustration of the Haraka round function for π_{512} (left) and π_{256}. Source: [45].

Security Kölbl et al. only claim second-preimage resistance for the Haraka construction as described above. However, collision resistance can be achieved by adding one more round to the permutations [45].

Sponge Construction Aumasson et al. [3] specify a sponge construction named HarakaS based on the permutation π_{512}. It uses a rate $r = 256$ and a capacity $c = 256$. It is used to support other input sizes and allow for longer hash values if required. Sponge constructions are presented in subsection 2.3.5.

Round Constants Kölbl et al. [45] define round constants used as the round keys for the AES round function evaluations. They propose round constants derived from π_{512} but also note that other choices would equally qualify as long as they do not contain strong symmetries [45].

Aumasson et al. [3] suggest deriving the round constants from a seed `seed`. More precisely, they derive the 40 120-bit constants using HarakaS:

$$RC_0, \ldots, RC_{39} = \mathsf{HarakaS}(\mathtt{seed}, 5120).$$

Note that for this operation, the default round constants are used. The derived round constants are then used in the functions $\mathsf{Haraka256}_{\mathtt{seed}}$, $\mathsf{Haraka512}_{\mathtt{seed}}$, and $\mathsf{HarakaS}_{\mathtt{seed}}$.

3.3. Hash Function Implementation

This section connects both preceding sections and covers how the presented hash functions are used to instantiate the signature schemes.

3.3.1. XMSS

Name	Hash Function	n	Defined in
SHA2_256	SHA-256	256	[36, 14]
SHA2_512	SHA-512	512	[36]
SHAKE_256	SHAKE128	256	[36]
SHAKE_512	SHAKE256	512	[36]
SHA2_192	SHA-256	192	[14]
SHAKE256_256	SHAKE256	256	[14]
SHAKE256_192	SHAKE256	192	[14]

Table 3.1.: Hash functions and n specified for XMSS

Table 3.1 shows the combinations of hash functions and the security parameter and hash output length n specified in RFC 8391 [36] and NIST SP 800-208 [14]. In the following, we describe how the four hash functions F, H, H_{msg}, and PRF are implemented.

Let Hash and n be a combination of hash function and output length as specified in Table 3.1. Let l_{td} be the type discriminator length as specified in Table 3.2.

n	l_{td}
192	32
256	256
512	512

Table 3.2.: Type discriminator length l_{td} for XMSS

If the chosen hash function is an XOF, let Hash have the output length n bits. For example, for SHAKE256, we define Hash as $\mathsf{Hash}(x) = \mathsf{SHAKE256}(x, n)$. For SHA-256 with $n = 192$, Hash is defined as $\mathsf{Hash}(x) = \mathsf{trunc}_{192}(\mathsf{SHA\text{-}256}(x))$.

The four functions used in XMSS are now defined as [36, 14]:

$$
\begin{aligned}
F(\mathsf{Key}, M) &= \mathsf{Hash}(toByte(0, l_{td}/8) \parallel \mathsf{Key} \parallel M),\\
H(\mathsf{Key}, M) &= \mathsf{Hash}(toByte(1, l_{td}/8) \parallel \mathsf{Key} \parallel M),\\
H_{msg}(\mathsf{Key}, M) &= \mathsf{Hash}(toByte(2, l_{td}/8) \parallel \mathsf{Key} \parallel M),\\
\mathrm{PRF}(\mathsf{Key}, M) &= \mathsf{Hash}(toByte(3, l_{td}/8) \parallel \mathsf{Key} \parallel M).
\end{aligned}
$$

Table 3.3 gives the input lengths for the underlying hash function. We observe that for all functions except H_{msg}, the input lengths are constant and known in advance. The input length for H_{msg} depends on the message to be signed.

For F and H, the value Key is always a pseudo-random value derived by PRF from a public seed SEED and the current address. In addition to this, the data to be hashed is XORed with a bitmask that is also computed using PRF, SEED and the address before it is passed to F or H as M.

For PRF, the value Key is usually the public seed SEED used for the computation of keys and bitmasks or the private seed SK_SEED used for pseudo-randomized message hashing. Additionally, PRF may be used with another secret seed S for pseudo-random key generation. For H_{msg}, the value Key consists of the pseudo-random value r derived from S, the root of the XMSS tree, and the index of the leaf to be used.

Function	|Key|	$\|M\|$	Total (Bits)
F	n	n	$l_{td} + 2n$
H	n	$2n$	$l_{td} + 3n$
H_{msg}	$3n$	variable	variable
PRF	n	256	$l_{td} + n + 256$

Table 3.3.: Hash function input length for XMSS

For the optimizations implemented and evaluated in this thesis, we focus on the parameter sets defined in NIST SP 800-208 [14].

3.3.2. LMS

Name	Hash Function	n, m (bits)	Defined in
SHA2_N32	SHA-256	256	[49, 14]
SHA2_N24	SHA-256	192	[14]
SHAKE_N32	SHAKE256	256	[14]
SHAKE_N24	SHAKE256	192	[14]

Table 3.4.: Hash functions and n specified for LMS

RFC 8554 [49] only specifies one choice of hash function and n. Additional options are defined in SP 800-208 [14]. Table 3.4 provides an overview over the hash functions and their respective output sizes defined by both publications. Analogous to XMSS, SHA2_N24 computes a full SHA-256 hash and truncates it to 192 bits.

Operation	Inputs (size)				Total size
WOTS Chaining	I (16),	q (4),	i (2),	j (1), data ($n/8$).	$n/8 + 23$
Key Compression	I (16),	q (4),	0x8080 (2),	data ($ln/8$).	$ln/8 + 22$
Tree Leaf	I (16),	r (4),	0x8282 (2),	HASH ($n/8$).	$n/8 + 22$
Tree Intermediate	I (16),	r (4),	0x8383 (2),	L_CHILD ($m/8$), R_CHILD ($m/8$).	$m/4 + 22$
PRF Key Gen	I (16),	q (4),	i (2),	0xFF (1), SEED ($m/8$).	$m/8 + 23$
Message Hash	I (16),	q (4),	0x8181 (2),	C ($n/8$), message.	variable

Table 3.5.: Hash operations in LMS with input sizes in bytes.

Table 3.5 shows the operations which the hash function H is used for and the respective inputs with their length. The input I is a 16-byte prefix that is used for every operation. It is randomly sampled at the time of key generation. The parameter q identifies the leaf of the LMS tree (that is, the WOTS key), i the Winternitz chain of an OTS key, and j the position in a Winternitz chain. The value r identifies the tree node that is computed and C is a random string sampled for each message.

Note that again the input sizes for almost all operations are fixed and only depend on the parameter choices. The input size for the message hash is not fixed as LMS can sign messages of arbitrary length.

The defined input format achieves implicit domain separation: There are no two different operations that can pass the same inputs to the hash function. This is because the value i contains the number of the current hash chain. For $n = 256$ and $w = 2$, the number of hash chains reaches its maximum at 265 chains. Hence, i never has the same value as any of the constants. The value of j has a maximum of $w - 2$ and therefore cannot reach 255 either.

3.3.3. SPHINCS$^+$

The SPHINCS$^+$ proposal [3] contains the specification for implementing the scheme with three cryptographic hash functions: SHAKE, SHA-2, and Haraka. Additionally, there are two variants per hash function. The `robust` variant applies pseudo-random bitmasks to the input before hashing, similar to XMSS. The `simple` variant omit these bitmasks, as in LMS. This significantly speeds up the scheme. However, the security analysis of the `simple` variant relies on the random oracle model. For a more detailed comparison, see Aumasson et al. [3, Section 8.1.6].

In the following, we present the hash function implementations for each cryptographic hash function as defined in [3, Section 7.2].

SHAKE The function $\mathsf{mask}(M, \mathrm{ADRS})$ is used to calculate the bitmasked input. As the `simple` variants do not use any bitmasks, mask is defined as $\mathsf{mask}(M, \mathrm{ADRS}) = M$. For the `robust` variants, the function is defined as:

$$\mathsf{mask}(M) = M \oplus \mathsf{SHAKE256}(\mathrm{PK.seed} \parallel \mathrm{ADRS}, |M|).$$

The Hash functions used by SPHINCS$^+$ are implemented as presented in Table 3.6.

SPHINCS$^+$ Function	Underlying Function	Inputs				Output Size
H_{msg}	SHAKE256	R,	PK.seed,	PK.root,	M.	m
PRF	SHAKE256	PK.seed,	ADRS,	SK.seed.		n
PRF_{msg}	SHAKE256	SK.prf,	OptRand,	M.		n
F	SHAKE256	PK.seed,	ADRS,	$\mathsf{mask}(M_1)$.		n
H	SHAKE256	PK.seed,	ADRS,	$\mathsf{mask}(M_1)$,	$\mathsf{mask}(M_2)$.	n
T_l	SHAKE256	PK.seed,	ADRS,	$\mathsf{mask}(M)$.		n

Table 3.6.: Hash operations in SPHINCS$^+$ with SHAKE

SHA-2 For $n = 128$, define SHA-X as SHA-256 and as SHA-512 for all other values of n. Let b be the block length of SHA-X. The function MGF1 refers to the mask generation function as specified in RFC 2437 [38]. By HMAC, we refer to the message authentication code defined in NIST FIPS 198-1 [18]. As an optimization, the address ADRS can be compressed from a 32-byte value to only 22 bytes. We denote the compressed address by ADRS^c.

For SHA-2, we define two masking functions for the `robust` variants:

$$\mathsf{mask}_F(M) = M \oplus \mathsf{MGF1}_{\mathsf{SHA\text{-}256}}(\mathrm{PK.seed} \parallel \mathrm{ADRS}^c, |M|/8),$$
$$\mathsf{mask}(M) = M \oplus \mathsf{MGF1}_{\mathsf{SHA\text{-}X}}(\mathrm{PK.seed} \parallel \mathrm{ADRS}^c, |M|/8).$$

For the `simple` variants, let both functions be the identity function. Furthermore, define a padding function pad as:

$$\mathsf{pad}_x(M) = M \parallel 0^{x-|M|}.$$

and let $\text{seed}_{512} = pad_{512}(\text{PK.seed})$ and $\text{seed}_b = pad_b(\text{PK.seed})$.

The hash functions in SPHINCS$^+$ are implemented as in Table 3.7.

SPHINCS$^+$ Function	Underlying Function	Inputs			Output Size
H_{msg}	$\mathsf{MGF1}^{\mathsf{SHA\text{-}X}}$	R,	PK.seed,	$\mathsf{SHA\text{-}X}(R\|\|\text{PK.seed}\|\|\text{PK.root}\|\|M)$.	m
PRF	SHA-256	seed_{512}	ADRS^c,	SK.seed.	
PRF_{msg}	$\mathsf{HMAC}^{\mathsf{SHA\text{-}X}}_{\text{SK.prf}}$	OptRand,	M.		
F	SHA-256	seed_{512}	ADRS^c,	$\mathsf{mask}_F(M_1)$.	
H	SHA-X	seed_b	ADRS^c,	$\mathsf{mask}(M_1\|\|M_2)$.	
T_l	SHA-X	seed_b	ADRS^c,	$\mathsf{mask}(M)$.	

Table 3.7.: Hash operations in SPHINCS$^+$ with SHA-2

For $n < 256$, the outputs of F, H, PRF, and PRF_{msg} are truncated to the required length.

Note that the value of the public seed PK.seed is the same for every hash function call for one SPHINCS$^+$ key. This seed is always padded to the block length. As a result, the first block of the input data to SHA-256 or SHA-512 is always the same.

Haraka The last specified cryptographic hash function for SPHINCS$^+$ is Haraka as introduced in Section 3.2.3. As for SHA-2, we start by defining two masking functions for the `robust` variants:

$$\mathsf{mask}_F(M) = M \oplus \mathsf{Haraka256}_{\text{PK.seed}}(\text{ADRS}),$$
$$\mathsf{mask}(M) = M \oplus \mathsf{HarakaS}_{\text{PK.seed}}(\text{ADRS}, |M|).$$

The hash functions are implemented as shows in Table 3.8.

SPHINCS$^+$ Function	Underlying Function	Inputs			Output Size
H_{msg}	$\text{HarakaS}_{\text{PK.seed}}$	R,	PK.root,	M.	m
PRF	$\text{Haraka512}_{\text{PK.seed}}$	ADRS,	SK.seed.		
PRF_{msg}	$\text{HarakaS}_{\text{PK.seed}}$	SK.prf,	OptRand,	M.	n
F	$\text{Haraka512}_{\text{PK.seed}}$	ADRS,	$\mathsf{mask}_F(M_1)$.		
H	$\text{HarakaS}_{\text{PK.seed}}$	ADRS,	$\mathsf{mask}(M_1)$,	$\mathsf{mask}(M_2)$.	n
T_l	$\text{HarakaS}_{\text{PK.seed}}$	ADRS,	$\mathsf{mask}(M)$.		n

Table 3.8.: Hash operations in SPHINCS$^+$ with Haraka

4. Related Work

This chapter gives an overview of other work on the topic of the optimization of HBSs. Optimization techniques can generally be divided into two categories: implementation optimizations and scheme variations. Implementation optimizations aim at improving and accelerating the implementation of a specific HBS. Scheme variations are incompatible changes to the signature scheme itself. These variations often improve the signature size, signature cost, or verification cost. This chapter first covers implementation optimization and then gives an overview of proposed scheme variations.

4.1. Implementation Optimization

This section presents previous work that aims to improve HBSs. It is structured as follows: First, we cover optimization techniques for SHA-2 and describe the efficient implementation of Haraka. We continue by presenting research on how to apply these optimizations to HBSs and conclude with an overview for other scheme-specific optimizations.

4.1.1. Optimizing SHA-2

As the computation of hash values is crucial for many cryptographic and non-cryptographic applications, a large amount of previous work on accelerating hash functions exists. This section summarizes optimization strategies for the SHA-2 hash function family on the Intel x86 platform.

Core x86 Optimizations In 2012, Gueron [27] proposed changes to the OpenSSL SHA-2 implementation that only uses the core x86 instruction set on second-generation Intel Core processors. They claim a speedup factor of 1.22 for the SHA-256 update function. This result shows small implementation details can have a major impact on the performance of SHA-2.

SIMD Message Scheduling As described in Section 3.2.1, the SHA-2 compression function consists of two phases: The message schedule and state update phase. Gueron and Krasnov [28] observe that only the state update phase depends on the previous state. The message schedule phase only depends on the message block. This means that if multiple message blocks are known, the message schedule phase can be parallelized.

Gueron and Krasnov [28] propose the use of SIMD instructions on the Intel x86 platform. Depending on the processor family, they observe a performance improvement of up to 31% compared to the best implementation at the time. Additionally, this approach is expected to deliver better performance on later SIMD instruction sets.

Guilford et al. [30] elaborate on this idea and provide additional guidance on how to implement SHA-2 efficiently using SIMD message scheduling.

Multi-Message Hashing In their 2010 white paper, Gopal et al. [26] present the idea of performing cryptographic operations with predominant data dependencies on multiple messages in parallel using SIMD instructions. They give AES-CBC encryption and HMAC-SHA1 as examples of such operations.

Gueron and Krasnov [29] apply this approach to SHA-256: Multiple independent messages are hashed in parallel using SIMD instructions. On a platform with the Intel Advanced Vector Extensions (AVX), four messages can be hashed in parallel. Using this approach, they achieve a speed-up of 3.42x compared to OpenSSL at the time of writing and 2.25x compared to SIMD message scheduling.

An implementation of vectorized multi-message hashing with SHA-2 is available as part of the OpenSSL library [73, File `sha256-mb-x86_64.pl`].

SHA New Instructions Intel published the specification of the SHA-NI in 2013 [31]. These are an extension to the x86 instruction set that provides hardware acceleration for certain operations in the compression functions of SHA-1 and SHA-2.

Faz-Hernández et al. [22] present benchmark results for hashing with SHA-NI on an AMD Zen platform. Their implementation provides a speedup of about 4 for 64-byte messages compared to OpenSSL's x86 implementation without SHA-NI.

It is important to note that even if the specification for SHA-NI was released in 2013, it is still not available in many computers in use at the time of writing this thesis. This is because the first mainstream Intel Core processor architecture to support SHA-NI was Ice Lake, which was released in 2019. This architecture was, however, only used in mobile and server processors. Support in desktop processors was introduced as late as 2021 [77, 76, 78].

Pipelining SHA-NI Multi-message hashing can also be used to increase the efficiency of hash implementations that make use of SHA-NI. This is achieved by pipelining SHA-NI instructions for the hash instances. Interleaving the execution of multiple instances allows better utilization of the processor pipeline by reducing the adverse effects of data dependencies.

Faz-Hernández et al. [22] implement and benchmark multi-message hashing by pipelining SHA-NI instructions. Their results show an 18% improvement for only two simultaneously hashed messages compared to the sequential computation of the message hashes. For four instances, the result improves to 21%.

4.1.2. Implementing Haraka

The Haraka hash functions were explicitly designed for the use on platforms with AES hardware acceleration [45]. The Intel x86 platform, for example, provides the `aesenc` instruction as part of the AES New Instructions (AES-NI). This instruction can be used to efficiently compute the AES round function that is used in Haraka. Additionally, the permutation that is used after applying the AES round function to the state was specifically designed to be efficiently implementable using the instructions `punpckldq` and `punpckhdq` on x86.

To further increase the pipeline utilization, Kölbl et al. [45] also propose, implement, and benchmark multi-message hashing with Haraka. Their results show an improvement of up to 25.8% when using four parallel messages with Haraka_{512} on Skylake.

4.1.3. Application to Hash-Based Signature Schemes

Faz-Hernández et al. [22] implement XMSS and XMSS^{MT} with the acceleration techniques for SHA-2 described above: Multiple message hashing and SHA-NI. They use both SIMD instruction sets Streaming SIMD Extensions (SSE) and Intel Advanced Vector Extensions 2 (AVX2) for multiple message hashing. With SSE, four messages can be hashed in parallel, and eight messages with AVX2. Additionally, they provide a sequential and a pipelined implementation using SHA-NI. Their source code is available on GitHub [21].

Furthermore, they present the following benchmark results for their XMSS implementation on an AMD Zen CPU: Compared to a core x86 implementation, XMSS key generation is faster by a factor of 1.72 and 1.18 for SSE and AVX2, respectively. For the sequential and the pipelined SHA-NI implementation, they achieve an improvement of 4.01 and 4.41.

The observation that the AVX2 implementation performs worse than SSE is due to a platform limitation on AMD Zen. Zen emulates a 256-bit AVX2 instruction by executing two 128-bit vector instructions. On other platforms, this limitation does not exist. As an example, they also present benchmarks for the Intel Kaby Lake platform. Here, they achieve an acceleration factor of about 2.2 for SSE and 4.0 for AVX2.

As part of their proposal for SPHINCS, Bernstein et al. [4] describe an efficient implementation for SPHINCS-256, a specific instantiation of SPHINCS which uses the BLAKE cryptographic hash function and the ChaCha permutation to implement the hash functions of the scheme. They suggest applying multi-message hashing for eight messages and using this to vectorize the computation of eight independent WOTS public keys and the corresponding L-Tree. Their code is available in the public domain [61].

Kölbl [44] implements and evaluates SPHINCS with additional hash functions: SHA-2, Keccak, Simpira, Haraka, and ChaCha. Multi-message hashing is implemented for every hash function. SHA-2 uses SHA-NI, if available. Additionally, an implementation for ARM is provided that utilizes the Neon SIMD instruction set and the SHA-2 hardware

acceleration. Kölbl presents benchmark results across multiple platforms. The used source code is published on GitHub [43].

As part of their submission to NIST, Rijneveld et al. [62] provide a reference implementation for the SPHINCS$^+$ scheme. There is no precise description of the optimizations applied in this implementation, only that it uses architecture-specific optimizations such as AES-NI and AVX2 [5]. Additionally, they implement multi-message hashing for all hash functions.

Hanson et al. [32] improve this implementation by utilizing the SHA-NI. In addition to using AVX2 for parallelizable hashes, they use SHA-NI for non-parallelizable hashes. Through this optimization, they achieve an 8% improvement in signing and 23% in verification.

Before optimizing SPHINCS$^+$, Hanson et al. investigate the performance of multi-message hashing on an Intel Tiger Lake processor. They compare a sequential implementation, a presumably pipelined SHA-NI implementation, an AVX2, and an AVX-512 implementation for 16 parallel hashes. Compared to the sequential implementation, AVX2 reduces the time for 16 hashes by 56.3%, SHA-NI by 53.9%, and AVX-512 by 81.6%. These results show that there is additional optimization potential by using AVX-512 for the parallelizable hashes instead of AVX2. Even if AVX-512 is not available on an older processor, SHA-NI performs slightly better than AVX2. However, Hanson et al. leave this optimization potential for SPHINCS$^+$ unconsidered.

4.1.4. Scheme-Specific Optimizations

Wang et al. [74] propose a so-called software-hardware co-design for XMSS. Their work contains designs for hardware accelerators as well as two software optimizations. In the following, we focus on the latter. The authors publish their source code on GitHub [75].

XMSS Fixed Padding Wang et al. observe that, for most hash operations in XMSS, the input length to the underlying hash function (for example, `SHA-256`) is fixed and known in advance. For details on the input formats for XMSS, refer to Section 3.3.1.

SHA-2 applies padding to the input to ensure that its length is a multiple of the block length (see Section 3.2.1). This padding depends on the message length. As this is mostly known in advance, the authors propose to hard-code this padding to save the time required to calculate the padding at runtime. Though they limit their remarks to XMSS with SHA-2, this approach is applicable to every HBS scheme with fixed input sizes and hash functions with length-depending padding.

Their benchmark results show that this optimization accelerates all XMSS operations by a factor of 1.06 compared to the XMSS reference implementation [63].

XMSS PRF Caching Furthermore, Wang et al. observe that for XMSS with SHA-2 and $n = 256$, the first 512 bits of the input to SHA-256 for PRF are:

$$toByte(3, 32) \parallel \texttt{Key}.$$

As we elaborated in Section 3.3.1, the value Key is always one of three possible values per key pair. Because the block size of SHA-256 is exactly 512 bits, it is possible to only consume each of the three values once and cache the state of the hash function. If a new message with the same first block should be hashed, the first block can be disregarded and the corresponding cached state can be restored.

Because restoring a SHA-2 state is significantly faster than one evaluation of the SHA-2 compression function, Wang et al. observe a speedup of about 1.44 for all XMSS operations compared to the XMSS reference implementation with the fixed padding optimization.

We note that Wang et al. assume in both [74, 75] that Key is always the public seed SEED. However, there are three different possibilities. In their code, the other values of Key are passed to PRF in `expand_seed()` in `wots.c` and `xmssmt_core_sign()` in `xmss_core.c`. Nevertheless, PRF ignores the provided value and restores the single cached state.

This does not limit the applicability or effect of their optimization. However, we expect that their implementation [75] is not correct when PRF caching is used. Still, this can be fixed easily. We describe one possibility in Section 6.2.2.

SPHINCS Parallelism Sun et al. [67] parallelize SPHINCS. Unlike multiple message hashing, their work aims to parallelize using multiple cores instead of SIMD data parallelism. They optimize for both multi-core CPUs and GPUs. As part of their work, they present parallel versions of both WOTS and HORS as well as a parallelized version of SPHINCS. They present extensive benchmark results for both x86 CPUs and Nvidia GPUs.

4.2. Scheme Variants

While the previous section focused on how to optimize implementations for a certain HBS, this section covers different variants of HBSs proposed in the literature. These variants usually improve the signature or verification time, or the signature size.

Many proposals only focus on the underlying WOTS and can be used in a Merkle tree scheme of choice. Few proposals consider the entire scheme.

This section starts by presenting RapidXMSS and SPHINCS+C. Afterward, it gives an overview on the work of WOTS with constant-sum encodings and zNAF encodings.

4.2.1. RapidXMSS

Bos et al. [8] introduce RapidXMSS, a variant of XMSS that is geared towards fast signature verification at the cost of more expensive signature creation. RapidXMSS is built to be verifiable by existing XMSS implementations and mainly modifies the signature generation. Overall, they claim an expected improvement of the verification time by a factor of 1.44 at a signature generation time of one minute.

RapidXMSS is mainly based on WOTS-R as introduced in Section 2.4.3. The contribution of Bos et al. primarily consists of a security proof, a theoretical analysis of the variant, and an experimental verification thereof. Also, they propose to use a caching strategy to achieve signature performance independent of the message length: to calculate the hash of $M \parallel \lambda^{(r)}$ for all $\lambda^{(r)}$, M is consumed once and the resulting state is cached. For each $\lambda^{(r)}$, this state is restored and $\lambda^{(r)}$ is consumed. This optimization significantly improves the performance of the signing process, especially for longer messages M.

Bos et al. do not apply the WOTS-B optimization proposed by Perin et al. [57] to preserve compatibility with existing XMSS verification implementations.

4.2.2. SPHINCS+C

Kudinov et al. [46] propose SPHINCS+C which is a variant of SPHINCS$^+$ optimized for smaller signatures. They claim a 20% reduction in signature size for one choice of parameters while reducing signing time as well.

SPHINCS+C introduces WOTS+C (see Section 2.4.4) and a variant of FORS called FORS+C. Similar to WOTS+C, FORS+C utilizes message hashing with a counter. To compress the signature, FORS+C requires that the value of the last message block is 0. Therefore, it is not necessary to sign this block, compute a FORS tree and include the authentication path for this FORS tree in the SPHINCS+C signature. Kudinov et al. remark that the expected number of hashes required to find a suitable counter and the number of hashes saved by removing the last tree are equal. Therefore, the signature time is roughly unchanged while reducing the signature size and slightly reducing the verification time.

The primary changes between SPHINCS$^+$ and SPHINCS+C are the different OTS and FTS. In general, SPHINCS+C reduces the signature size while slightly reducing the signature time [46]. Note that the signature time is randomized in SPHINCS+C which is not the case for SPHINCS$^+$.

4.2.3. WOTS Encodings

We remarked in Section 2.4.2 that the mapping from a message's hash to the hash chain positions $(b_l, \ldots, b_0)$ by splitting the hash value into blocks and appending a checksum can be seen as a domination-free encoding. In the following, we present three other classes

of encodings proposed in the literature: run-length encoding, constant-sum encoding, and non-adjacent form encoding.

WOTS encoding functions are extensively covered by Perin [55]. We refer the interested reader there and focus on the essentials for the sake of brevity.

Run-Length Encoding The first alternative encoding is due to Steinwandt and Villányi [66]. Their encoding scheme performs a run-length encoding for the message hash. As this encoding is only possible for hash values with a certain structure, they use hashing with a counter to find a suitable hash. For their scheme, they claim a reduction of verification cost by 33% while increasing the key generation and signing cost by a factor of 2 and 7, respectively. However, they only compare their results to WOTS with $w = 4$. Nowadays, $w = 16$ is a more common choice that halves the signature size compared to $w = 4$. We conjecture that reducing the signature size by half for the scheme by Steinwandt and Villányi would significantly increase signing cost which would lead to an unfavorable comparison to WOTS. We also remark that their work was published in 2008 and to the best of our knowledge, there has been no follow-up research on run-length encodings for WOTS.

Constant-Sum Encodings Cruz et al. [16] propose a so-called constant-sum encoding. This idea is based on the observation that if the sum of the hash-chain positions for an encoding is always a constant for all messages, this encoding is always domination-free. Their encoding is optimized for faster verification and greatly increases the key generation and signing time. For a parameter choice with a signature size comparable to WOTS with $w = 16$, the proposed scheme reduces the (expected) verification cost by 47%. Additionally, the verification cost is guaranteed and does not depend on the message. However, the cost for key generation and signing is increased by a factor of 12.7 and 25.9, respectively.

Kaji et al. [37] improve upon this encoding. They observe that most messages are encoded to only high positions in the hash chains. Therefore, they propose to only construct the upper part of the hash chains. To ensure that construct only requires nodes from the upper part of the hash chains, they use hashing with a counter. Kaji et al. only provide a direct comparison of their scheme to WOTS for $w \geq 64$. For an expected number of eight different counter values to try, they reduce key generation and (expected) verification costs by 6.3% and 36.6%, respectively. Note that the signature cost does not include finding a suitable hash and computing the encoding.

Perin et al. [56] propose another variant that works with shorter hash chains and therefore allows faster key generation and, to some extent, signing. It is deterministic and does not require hashing with a counter. However, they do not provide a way to efficiently compute this encoding. The algorithm in their work is expensive to compute. Overall, the use of this encoding only presents a small improvement over WOTS because of this.

Zhang et al. [80] introduce SPHINCS-α, a variant of SPHINCS$^+$ that also includes new variants of WOTS and FORS. Their WOTS variant also uses a constant-sum encoding similar to [56]. Zhang et al. prove that their encoding is size-optimal, that is, no other domination-free encoding uses fewer code words.

We remark that WOTS+C (see Section 2.4.4) uses a constant-sum encoding in some sense. The constant sum is however not achieved by an encoding algorithm as above, but only through hashing with a counter.

Non-Adjacent Forms The idea of using non-adjacent forms as an encoding for WOTS is due to Roh et al. [64] and was later improved by Zheng and Jr. [81]. We refrain from describing the encodings and refer the reader to the literature. Overall, [81] achieve a reduction of the verification cost of 30-40% at the cost of a 5-70% increase in key generation cost and a 52-170% increase in signing cost for a verification-optimized parameter set.

5. Practical Foundations

This chapter covers the practical foundations of this work. It introduces the architecture of the Java platform and its relevant limitations, gives an overview of the other software used in this work, and presents the methodology used for benchmarking our optimizations.

5.1. Java Architecture and Limitations

Language Architecture Languages like C and Rust are so-called compiled languages. Their source code is compiled statically (i.e. without execution) into machine code. The resulting platform-specific binary can be executed later on the platform it was compiled for. Therefore, the resulting binary can be executed without any runtime dependencies, ignoring libraries.

Interpreted languages do not require explicit compilation before the execution. An interpreter interprets and executes the source code. This of course requires a language-specific interpreter to be present at runtime. However, code written in an interpreted language can usually be executed on any platform. One disadvantage of interpretation is that the process and therefore the executed code is generally slower than executing native code.

Java employs a hybrid model that tries to reap the benefits of both approaches: performant, yet platform-independent code. To achieve this goal, Java code is first compiled into a low-level, yet platform-independent intermediate language, the so-called Java byte code.

This byte code is interpreted by the Java Virtual Machine (JVM) at runtime. The overhead for interpretation is smaller than for classical interpreted languages because the code is already compiled and not raw source code.

Just-in-Time Compilation To achieve better performance, the Java runtime uses a so-called Just-in-Time Compiler (JIT) in addition to the interpretation mechanism. The JIT compiles byte code into native code at runtime. In general, the JVM does not compile all byte code immediately, but only code portions that are executed repeatedly. In these cases, the savings of native code outweigh the compilation cost.

Additionally, the JIT can observe the previous executions of the code and therefore has more information available about the code and its typical execution than a static compiler. This information can be used to further optimize the resulting native code.

Oaks [53, Chapter 4] gives a detailed introduction to the Java JIT.

Limitations Due to this platform-independent design, it is not possible for Java code to directly use platform-dependent features and instructions. For the use with HBSs, we are especially interested in SIMD instructions and SHA-NI.

The upcoming Java Vector API [65] may provide a way to implement a more efficient message scheduling or multi-message hashing in Java (see Section 4.1.1). As of writing this thesis, the Vector API is still in the incubator.

Nevertheless, there are two ways that Java code can profit from native features: via the Java Native Interface (JNI) and JIT intrinsics.

Java Native Interface JNI allows Java code on the JVM to interact with native code [54]. The native code is platform-specific and can therefore use any feature the platform provides.

However, calling native code from Java comes at a considerable cost. This cost is even greater when the parameters passed to the native code are not primitives. During the time an object or array is used in native code, it must be pinned and explicitly released later. Pinning however hinders the garbage collector from running and significantly affects the performance of the program [53].

JIT Intrinsics Java compiler intrinsics [24] change how the JIT handles certain methods: instead of compiling the Java byte code, a hard-coded native implementation will be executed. This native implementation of the method may make use of additional optimizations that the JIT cannot use on-the-fly. For example, there is an intrinsic for the SHA-2 compression function that uses SHA-NI, if available.

This approach is limited to a few methods in the Java API as introducing a new intrinsic requires changes to the Java runtime. Unlike JNI, intrinsics are not intended to be directly used, added, or modified by the user.

Newland [52] provides a list of currently existing intrinsics in OpenJDK.

5.2. Software

This section gives a brief overview of the cryptographic libraries and related software used in this thesis and highlights their important features for this work.

5.2.1. BouncyCastle

BouncyCastle [69, 68] is a popular cryptography library for Java and C# maintained by The Legion of the Bouncy Castle. It is published as open-source under the MIT license [68]. Its source code is therefore freely available and can be easily modified.

In the following, we focus on the Java library. It provides implementations for a wide variety of cryptographic schemes and protocols, including a growing number of post-quantum cryptography algorithms. Most importantly for this thesis, BouncyCastle provides implementations of the HBSs XMSS, XMSS^{MT}, LMS, HSS, SPHINCS, and SPHINCS^{+}.

To the best of the author's knowledge, BouncyCastle provides the only Java implementations for these schemes that are intended for production use. Another SPHINCS^{+} implementation in Java is due to Heimberger [33]. It appears to be primarily intended for evaluation purposes.

The crypto implementations provided by BouncyCastle can usually be used in two ways: First, via a custom lightweight API [69] or via a Java Cryptographic Architecture (JCA)/Java Cryptographic Extension (JCE) provider.

5.2.2. OpenSSL

OpenSSL [73] is an open-source crypto library. Unlike BouncyCastle, it is a native library and is primarily written in C. Most relevant for this thesis, it provides implementations for the hash function families SHA-2 and SHA-3.

Implementations The crypto library contains different implementations for SHA-2. There is a generic implementation written in C [73, File `crypto/sha/sha512.c`] as well as a highly optimized one in Assembler [73, File `crypto/sha/asm/sha512-x86_64.pl`] (or, more precisely, a Perl script that generates an implementation in Assembler). The optimized implementation uses SHA-NI, if available. Otherwise, it uses AVX2, AVX, or SSE to compute SHA-2 hashes.

OpenSSL also provides a multi-message implementation for `SHA-256` to hash multiple messages in parallel [73, File `crypto/sha/asm/sha256-mb-x86_64.pl`]. However, we were unable integrate this implementation into our code.

Similarly, there is a generic implementation for SHA-3 [73, File `crypto/sha/keccak1600.c`] and optimized implementations for x86 and multiple SIMD instruction sets [73, Files `crypto/sha/asm/keccak1600-*`].

All optimized implementations originate from the *CRYPTOGAMS* project by Polyakov [58].

Interfaces For SHA-2, there are two ways to access the implementation: a legacy interface [73, File `doc/man3/SHA256_Init.pod`] and the newer, more generic EVP interface [73, File `doc/man3/EVP_DigestInit.pod`]. SHA-3 is only accessible using the newer EVP interface.

The older SHA-2 interface is considered deprecated as of OpenSSL 3.0. The older legacy interface is still available on OpenSSL 3.0, but it is implemented as a wrapper for the EVP interface [73, File `crypto/evp/legacy_sha.c`]. However, the EVP interface provides significantly worse performance for small inputs [2, File `csrc/hash_template.cpp.template`].

As HBSs almost exclusively work with short inputs, it is not sensible to work with the EVP interface. Therefore, we will continue to work with OpenSSL 1.1.1 for this thesis because it still exposes the SHA-2 implementation directly via the legacy interface.

We were also able to experimentally confirm a significant drop in performance for the legacy interface on short inputs when moving from OpenSSL 1.1.1 to version 3.0.

5.2.3. eXtended Keccak Code Package

The eXtended Keccak Code Package (XKCP) [79] is a collection of implementations of the Keccak hash function family, which includes SHA-3, and therefore `SHAKE128` and `SHAKE256`.

For example, it includes Keccak implementations derived from *CRYPTOGAMS* [58] which is also used in OpenSSL for different SIMD instruction sets. Additionally, it contains several multi-message implementations.

XKCP includes different interfaces that can be used to interact with the Keccak implementations. We focus on the `SimpleFIPS202` interface, which provides simple and low-overhead access to the hash functions specified in FIPS 202 [19].

For this reason, XKCP may be preferential over OpenSSL for short-input hashing with SHA-3. Both use very similar implementations of Keccak but XKCP has an interface with a lower overhead.

5.2.4. Amazon Corretto Crypto Provider

The Amazon Corretto Crypto Provider (ACCP) is a JCA/JCE provider that makes high-performance crypto primitives available to Java applications [2]. This includes hashing with SHA-2.

ACCP does not implement SHA-2 on its own, instead, it relies on OpenSSL which is called via JNI. As we elaborated above, OpenSSL uses a highly optimized implementation of SHA-2. ACCP's JCA/JCE provider can now easily be used as a drop-in replacement for the default provider and increase hashing performance in any Java application using JCA/JCE.

5.3. Benchmarking Methodology

To evaluate the performance of our optimized HBS implementations in Java, we conduct a series of benchmarks. The following section describes the tools and environment used in our experiments.

5.3.1. Java Microbenchmark Harness

The Java Microbenchmark Harness (JMH) is a framework for writing and executing benchmarks in Java (and other JVM languages) [71].

Every JMH benchmark consists of a benchmark method that is invoked repeatedly and whose performance is measured. Optionally, a benchmark may include setup and teardown methods that can be run at different stages. Additionally, benchmarks can have parameters that are configurable at run-time.

A benchmark run is structured as follows: for each combination of benchmark method and respective parameters, a specified number of trials (also referred to as forks) is performed. For each trial, a new JVM is launched and used to perform a specified number of warm-up and measurement iterations. Warm-up iterations are intended to allow the JVM to do operations like class loading, JIT compilation, caching, or similar. Generally, any operation influencing the run times that is only done once in a long-running program. As the name suggests, the performance of the benchmark method is measured during the measurement iterations.

A so-called benchmark mode determines how this measurement works. For our benchmarks, we use the average time mode. In this mode, the benchmarking method is invoked repeatedly for each measurement iteration until a specified time interval has passed. Afterward, the total elapsed time is divided by the number of invocations of the benchmark method.

For a more detailed introduction to JMH and Java benchmarking in general, refer to [53, Chapter 2].

5.3.2. Benchmarking Environment

As the execution environment for the benchmarks, we chose the Amazon Elastic Compute Cloud (EC2) on the Amazon Web Services (AWS) cloud platform. EC2 provides virtual machines in a wide variety of configurations.

The choice to run on a cloud hyperscaler was made because of the following considerations:

Reproducibility Benchmarks run on EC2 are reproducible in the sense that the benchmarking setup (i.e. the benchmarked software including dependencies and operating system) can be easily reset after each run and reproduced at a later point. Additionally, we assume that the performance of an EC2 instance is reproducible in the sense that each instance with the same setup provides very similar performance.

Variety Many different instance types are available on EC2. Instance types vary primarily in the processor family and additional features like graphics processors or machine learning accelerators. In this work, we do not make use of any such additional

features. The range of available processors includes a range of different x86 processors by Intel and AMD as well as ARM processors [1].

Scalability AWS allows us to easily run multiple independent instances of the benchmark machine. This can be used to execute benchmarks in parallel while guaranteeing that they do not directly influence each other. Therefore, the development process can be accelerated compared to running all benchmarks sequentially on a single physical machine.

Cost Running on AWS is more cost-efficient than buying physical systems to benchmark on. This is exacerbated by the fact that some hardware features (most notably, SHA-NI) are only available on the latest generations of processors.

Instance type selection To minimize benchmark the run time, we evaluate different EC2 instance types by benchmarking the XMSS key generation for the `SHA2_16_256` parameter set with unmodified BouncyCastle to find the fastest instance type for our use case.

XMSS key generation in BouncyCastle is single-threaded, hence we are not expecting improvements for high core count instances. However, the influence of other background tasks on the benchmark results could decrease when multiple cores are available.

Key generation creates a relatively consistent workload over a longer time. As a result, burstable instance types should be avoided. They allow bursting CPU usage above the intended maximum for a limited time. On EC2, the low-end T4, T3, and T2 instances are burstable [1].

Lastly, we note that we will focus on the x86 because of its prevalence among CPUs for desktop and server use.

Instance type	CPU	Cores
`m5zn.large`	Intel Xeon Platinum 8252C (Cascade Lake)	1
`m5zn.xlarge`	Intel Xeon Platinum 8252C (Cascade Lake)	2
`m5zn.2xlarge`	Intel Xeon Platinum 8252C (Cascade Lake)	4
`m6i.large`	Intel Xeon Platinum 8375C (Ice Lake)	1
`m6a.large`	AMD EPYC 7R13	1
`m6g.large`	AWS Graviton2 (Neoverse-N1)	1

Table 5.1.: Evaluated EC2 instance types

Therefore, we choose the instance types with the most promising CPU platform from the general purpose category [1]. Table 5.1 shows the instance types with the corresponding size that we evaluated.

Figure 5.1 shows that there is a considerable difference in performance across different EC2 instance types. Comparing the `large` instances, we see that the `m5zn` is the fastest, followed by `m6i` and lastly `m6a`. Overall, the key generation takes about 72% longer on `m6a`

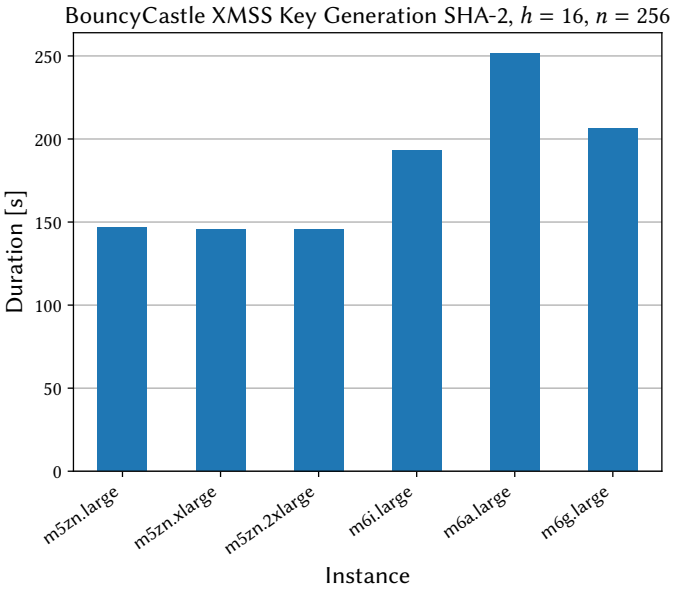


Figure 5.1.: XMSS key generation by EC2 instance type for the `SHA2_16_256` parameter set

compared to `m6i`. The ARM-based `m6g` is slower than `m6i` but still considerably faster than `m6g`.

Across the different sizes of the `m5zn` instance type, we see only insignificant differences. However, in other tests, we observe that the variance across multiple iterations is smaller when more than one core is available.

Hence, we pick the `m5zn` instance type as the primary benchmark machine and choose the `xlarge` size for single-threaded tasks. However, this instance does not support SHA-NI. Therefore, we decide to also run benchmarks on `m6i` to evaluate the impact of SHA-NI.

Benchmark Setup In the following, we briefly describe how we configure the EC2 instances for benchmarking. We follow the standard procedure for benchmarks and disable Simultaneous Multithreading (SMT). The instances run the Ubuntu 20.04 operating system and the OpenSSL release provided by it. A custom fork of OpenJDK 18 is used as described in Section 6.2.5.

Table 5.2 gives an overview of our benchmarking setup.

Component		Remarks
CPU `m5zn`	Intel Xeon Platinum 8252C	SMT disabled
CPU `m6i`	Intel Xeon Platinum 8375C	SMT disabled
Operating System	Ubuntu 20.04.5 LTS (Focal Fossa)	
JDK	Fork of OpenJDK 18+36	see Section 6.2.5
OpenSSL	1.1.1f	see Section 5.2.2
XKCP	Commit `64404be`	

Table 5.2.: Overview of the benchmarking setup

6. Implementation

This section presents the main parts of our contribution and is structured as follows: Fist, we present how we evaluate the XMSS reference implementation as a baseline. Second, we describe the different implementation optimizations we implement and evaluate in BouncyCastle. Third, we describe how paralellization can be used to accelerate HBS operations. Lastly, we give a theoretical analysis of two verification-optimized scheme variants and describe their experimental validation.

The source code used in this thesis is published as open-source [59].

6.1. XMSS Reference Implementation

To provide a performance baseline and an external comparison for our work, we use the XMSS reference implementation [63] accompanying RFC 8391 [36]. It is written in C and supports the parameter sets specified in [36, 14] (see Section 3.3.1).

For SHA-2, this implementation relies on OpenSSL which is used via the legacy interface (see Section 5.2.2). More precisely, it uses the `SHA256` function provided by OpenSSL. We discuss different interfaces for `SHA-256` in Section 6.2.4.2. The SHAKE hash functions are computed by a custom implementation in C [63, File `fips202.c`].

We observe that this custom implementation is slower than the SHAKE implementations contained in OpenSSL and XKCP. For more details, refer to Section 7.2.3.2. To achieve a fair comparison, we replace the custom implementation with XKCP.

The project already includes a benchmark that measures performance for key generation, signing, and verification [63, File `test/speed.c`]. We slightly modify this benchmark to test multiple parameter sets in one run and output the results in a more convenient format.

Additionally, we add a dedicated benchmark for the function `core_hash` [63, File `hash.c`]. This function performs the hash operation with the underlying hash function depending on the chosen parameter set. We evaluate the function with input sizes that correspond to the Winternitz chaining function F (see Section 3.3.1).

Apart from these changes, we leave the reference implementation unmodified. We present the benchmark results in Section 7.1.

6.2. Optimization Levels

This section describes the various different changes and optimizations we implement in BouncyCastle to improve the performance of the HBSs. We present optimizations to BouncyCastle's hash implementations, different variants to use native hash implementations using JNI and describe how we use JIT intrinics.

6.2.1. Hash Encapsulation

To allow for easily replaceable hash implementations, we use a class for each HBS that provides the scheme's hash function and implements them using some underlying hash function implementation.

For XMSS, the hash functions are already encapsulated in the class `KeyedHashFunctions` in BouncyCastle [68]. The same applies to SPHINCS$^+$ and the class `SPHINCSPlusEngine`. For LMS, we extract all hashing to a class called `LMSHash`.

All those classes are abstract and multiple implementations exist. Additionally, we introduce the interface `HashingProvider` and provide one implementation for each optimization level. This can be used to retrieve the concrete implementation of `KeyedHashFunctions`, `LMSHash`, or `SPHINCSPlusEngine` for this optimization level for a choice of hash function and output length.

Furthermore, we introduce a `HashingProviderProvider` through which one can configure and retrieve the hashing provider to be used (and therefore the optimization level).

6.2.2. BouncyCastle

Optimization Level	XMSS	LMS	SPHINCS$^+$
`bc`	yes	yes	yes

The `bc` optimization level refers to the hash implementations provided by and used in BouncyCastle by default. It does not include any explicit optimizations.

For all schemes, this optimization level uses the generic SHA-256 implementation in the class `SHA256Digest` and the SHAKE256 implementation in `SHAKEDigest` [68]. The implementation of the Haraka functions used in SPHINCS$^+$ can be found in the classes `HarakaS256Digest`, `HarakaS512Digest`, and `HarakaXof` [68].

All of these implementations are part of BouncyCastle, use plain Java, and implement BouncyCastle's `Digest` interface. A `Digest` exposes the operations typical for a message digest: an `update` method to consume several bytes and a `doFinal` method to calculate and retrieve the digest of the previously consumed bytes. Usually, the `Digest` holds an internal buffer of block size and the state. The state update function is invoked whenever enough bytes are provided via the `update` method. Additional bytes are stored in the internal

buffer. The `doFinal` method calculates the padding, invokes the state update function on the content of the buffer and the padding, and returns the hash value.

Optimization Level	XMSS	LMS	SPHINCS$^+$
`bc-optimized`	only `SHA-256`	only `SHA-256`	no

This optimization level applies a few, targeted optimizations to the existing `SHA-256` implementation in BouncyCastle. It uses the following three changes:

Buffering We described above that a typical implementation of a message digest uses a buffer of block size and invokes the state update function whenever this buffer is full. The `SHA256Digest` extends the `GeneralDigest` which has a buffer of one word (4 bytes). Whenever this buffer is full, it is passed on to the underlying `SHA256Digest` which in turn has a buffer of block size. As the input length is usually a multiple of 4 bytes, the `GeneralDigest` can be bypassed and the input can be directly passed to `SHA256Digest`.

Consuming Zero-Words For XMSS with $n = 256$, the hash type discriminator consists of seven zero-words (see Section 3.3.1). `SHA256Digest` clears its buffer after each call to the state update function. Therefore, it is not necessary to actually update the buffer when zero-words are consumed. It is sufficient to only increase the amount of data in the buffer.

PRF Caching We additionally implement the PRF caching optimization proposed by Wang et al. [74] and described in Section 4.1.4. We recall that this optimization is only applicable to XMSS with `SHA-256` and $n = 256$.

In Section 4.1.4, we described a problem in the implementation of this optimization provided by Wang et al.: they assume that the value `Key` is constant for every call to PRF for one XMSS key. This is, however, not the case. There are three possible values. We implement a lightweight hash map with 1024 entries indexed by 10 bits of the value `Key`. We save additional 32 bits of the address along the cached state. This brings the probability of two entries colliding to 2^{-10}. This only causes decaying performance. The probability of an undetected collision is 2^{-42} which we consider to be low enough for the purposes of this work. In the case of an undetected collision, the implementation will return the wrong results. However, this can be easily mitigated by storing and comparing all 256 bytes of the `Key` at little additional cost.

An alternative implementation could exploit that PRF is called from distinct contexts for the three possible values of `Key`. Therefore, it would be possible to pass not only the `Key` but also a cached state to PRF. This would eliminate the need for the hash-map structure described above but would require changes to the XMSS implementation itself. This may be the better approach for a productive implementation, but it would make generically swapping the underlying hash implementation harder.

6.2.3. Amazon Corretto Crypto Provider

Optimization Level	XMSS	LMS	SPHINCS$^+$
`corretto`	only `SHA-256`	only `SHA-256`	only `SHA-256`

The Amazon Corretto Crypto Provider (ACCP) provides a high-performance implementation of SHA-2 backed by OpenSSL and JNI as a JCA/JCE provider. We introduced ACCP in Section 5.2.4.

The `SHA-256` implementation provided by ACCP is accessible via the JCA/JCE interface `MessageDigest`. The HBS implementations by BouncyCastle however expect a `Digest` as described above. As the interfaces are quite similar, we write a simple adapter class that wraps the `MessageDigest` by ACCP into a `Digest`.

Instead of using the adapter class to instantiate the `KeyedHashFunctions`, `LMSHash`, and `SPHINCSPlusEngine`, we use alternative implementations of these classes to reduce overhead. However, as the interfaces of `Digest` and `MessageDigest` are almost identical, so are both implementations of these classes. The adapter classes are only used when the HBS implementation explicitly requires a `Digest`.

This approach can be applied to all signature schemes: XMSS, LMS, and SPHINCS$^+$. However, ACCP does not support SHA-3 or Haraka. Therefore this optimization level is limited to SHA-2 only.

6.2.4. JNI

Our evaluation of ACCP shows that native hash implementation can significantly speed up HBSs in Java. However, we also noted the limitations of ACCP: It currently only supports SHA-2 and it uses the generic JCA/JCE interface. This may be convenient for many applications but does not provide the best performance. This is partly due to the generic `update`/`doFinal` interface which is not necessary for our application. Additionally, we consider integrating additional optimizations and hash functions to be rather difficult.

For these reasons, we instead write our own replacement for ACCP which is specially designed for use with the HBSs in this thesis.

6.2.4.1. JNI Data Transfer

Through the Java Native Interface (JNI), Java code can interact with native code and vice-versa. As we sketched in Section 5.1, calling native functions from Java comes at a considerable cost. This cost consists of two parts: The cost of invoking the native function and the cost of transferring data between Java and native memory and back. While we cannot reasonably influence the cost of the method invocation, there are several ways to transfer data between Java and native memory. In the following, we describe these methods and our approach to benchmark them.

Benchmark Setting For hashing in HBSs, both input and input consist of binary data whose size varies depending on the choice of scheme and parameter set. The input of the hash function is typically between 46 bytes (LMS Tree Leaf for $n = 192$) and 128 bytes (XMSS H for $n = 256$). This range may not cover all hash function calls, especially since the initial message hash may have a longer input. However, most hash function calls in the HBSs in this thesis fall into this range. The output usually consists of 24 or 32 bytes.

We choose an input length of 64 bytes and an output length of 32 bytes for the benchmark. This means that a buffer of 64 bytes is passed from Java to native code where a trivial computation is performed: each element is added to one of its neighbors and the result is written to the output buffer which is passed back to the Java code.

We evaluated the following methods to transfer data between Java and native:

JNI GetByteArrayElements In this setting, two byte arrays are passed to the native method. One contains the input data and the output is written into the other. The methods `GetByteArrayElements` and `ReleaseByteArrayElements` are used to access these byte arrays from native code. The function `GetByteArrayElements` returns a pointer to the data contained in the array. This may or may not be a copy of the Java byte array. The function `ReleaseByteArrayElements` is used to free up allocated resources afterward and copy data back to the Java byte array, if necessary. For more details on these operations, see [54, Chapter 4].

JNI GetByteArrayElementsCritical This setting is very similar to the previous ones. It uses the functions `GetByteArrayElementsCritical` and `ReleaseByteArrayElementsCritical` to access the byte array. These behave similarly to the functions used above but introduce a so-called critical section. For details, refer to [54, Chapter 4]. We expect that this may have different performance characteristics.

JNI GetByteArrayRegion The functions `GetByteArrayRegion` and `SetByteArrayRegion` allow retrieving a copy of a byte array and write a native buffer back to a byte array. This may be beneficial because we only read from one byte array and write to the other. The methods above allow both reading and writing.

Direct ByteBuffer Unlike in the previous methods, the data is not passed as a byte array to the native method. Instead, a `java.nio.ByteBuffer` is used. More specifically, a direct `ByteBuffer`. In the native code, the function `GetDirectBufferAddress` can be used to directly retrieve the address of the memory associated with this buffer [54, Chapter 4]. This however comes with the drawback that the data must be explicitly copied into and out of this buffer on the Java side. We use the methods `put` and `get` of the `ByteBuffer` for this task.

Netty ByteBuf The Netty framework [70] provides a `ByteBuf` class that represents a buffer of bytes. For certain implementations of the `ByteBuf` class, it is possible to extract the memory address of the underlying memory that stores the data. This address can be passed via JNI as a primitive. The native code can then simply access the data using this address. Multiple `ByteBufAllocators` are available to create different `ByteBufs`. We evaluate both the `PooledByteBufAllocator` and the `UnpooledByteBufAllocator`.

Java Unsafe The `Unsafe` class allows low-level operations from Java that are considered to be unsafe [72, Class Unsafe]. For example, it provides the functionality to directly allocate native and copy memory between objects and native memory. We use this class to allocate two buffers for input and output and copy data between Java byte arrays and the buffers. The address of the buffers is passed via JNI to the native method which can directly access the memory. We remark that the use of the `Unsafe` class in production code should be avoided.

6.2.4.2. Chosing the Hash Implementations

This section describes which different native hash implementations and interfaces we evaluate to provide the best performance for the JNI hashing module.

Benchmark Setting We benchmark different implementations and interfaces for `SHA-256` and `SHAKE256` with input sizes of 52, 96, and 128 bytes as these are typical sizes for XMSS. The output size is 256 bits. This benchmark is native-only, it does not use any Java code.

We evaluate the following interfaces to OpenSSL for `SHA-256`:

OpenSSL direct This uses the function `SHA256` function provided by OpenSSL. This function takes an input buffer and an output buffer as a parameter and directly writes the hash value of the input buffer into the output buffer.

OpenSSL with CTX This variant uses the functions `SHA256_Init`, `SHA256_Update`, and `SHA256_Final` which work similarly to the `Digest` class in BC: `SHA256_Init` initializes a digest context of type `SHA256_CTX`, `SHA256_Update` consumes the provided message and updates the context accordingly, and `SHA256_Final` computes and consumes the padding and returns the hash value.

OpenSSL Fixed Padding This variant uses the fixed padding optimization proposed by Wang et al. [74] and introduced in Section 4.1.4. Our implementation is similar to the one provided by Wang et al. [75]. For a new hash, a new buffer is allocated and input data and the matching padding are copied into this buffer. The padding values are hard-coded into the program. This buffer is passed to `SHA256_Update`. The buffer's length is a multiple of the block size and therefore this invocation of `SHA256_Update` executes all required state updates. The internal buffer is empty and the current state is the hash value. Therefore, the hash value is then extracted from the `SHA256_CTX`. It is not possible to use the `SHA256`

function as it would add another padding to the message. We remark that Wang et al. [75] instead copy the input data into a static buffer that contains the appropriate padding at the end. This approach is however generally not thread-safe.

OpenSSL Custom Padding To evaluate the real benefit of the fixed padding compared to saved copy operations to the internal buffer, we evaluate another custom padding implementation. The input data is copied to a larger buffer and the required SHA-2 padding is computed on-the-fly. As described in Section 3.2.1, computing the padding is a rather simple task. Afterward, the padded message is passed to `SHA256_Update` and the hash value is extracted from the `SHA256_CTX`.

For `SHAKE256`, we evaluate the following implementations:

Custom Uses the custom SHA-3 implementation also used in the XMSS reference implementation [63]. It is written in plain C without any explicit optimizations.

OpenSSL This variant uses the SHA-3 implementation provided by OpenSSL and accessed via the EVP interface. For further details, see Section 5.2.2.

XKCP This evaluates a SHA-3 implementation of XKCP [79]. We remarked in Section 5.2.3 that these implementations are closely related to those used in OpenSSL. The main difference is the interface.

6.2.4.3. Implementation of jni-hash

This section describes the implementation of `jni-hash`, our JNI bridge for hashing in HBSs, in detail. We present the resulting optimization levels based on the insight gained in the previous sections.

Optimization Level	XMSS	LMS	SPHINCS$^+$
`jni`	yes	yes	`SHAKE256` and Haraka

The optimization level uses the best of the hash function implementations described above but does not apply any further optimizations. We implement `jni` for all parameter sets of the three HBSs except SPHINCS$^+$ with SHA-2. This can also be implemented. However, we refrain from it as we do not expect any additional insights.

Data Transfer We use the Netty unpooled `ByteBuf` to transfer data between Java and native code. For more details, see Sections 6.2.4.1 and 7.2.3.1. However, this makes additional handling necessary: as allocating buffers is costly, the `ByteBufs` should be reused whenever possible. Additionally, they must be explicitly freed.

Using `static` buffers is also not possible as this would not be thread-safe. Instead, we use Java `ThreadLocals`. They ensure that each thread operates on an independent instance. However, we can also use them to ensure that each thread only allocates one instance and uses it for its lifetime. We use `ThreadLocals` for the `KeyedHashFunctions` and `LMSHash` as described in Section 6.2.1. To explicitly release the `ByteBufs` before garbage collection, we use the Java `Cleaner`.

SHA-256 To implement SHA-256, we use OpenSSL with a custom padding function as in the previous section.

SHAKE256 We use the implementation provided by XKCP for `SHAKE256`. More specifically, we use the AVX2 implementation on `m5zn` and the AVX-512 implementation on `m6i`.

Haraka The previous sections did not discuss Haraka as there is only one existing native implementation and therefore no comparison is possible. This is the reference implementation by Kölbl [42]. Heimberger [33] provides a Java implementation for SPHINCS$^+$ that includes a JNI bridge to a modified version of the Haraka reference code. While it is written in C, the core implementation uses intrinsics for AES-NI to implement the permutation as suggested in [45] and presented in Section 4.1.2.

For `jni-hash`, we build upon the code provided by Heimberger [33] and slightly modify it to integrate it into our project. Additionally, we adapt the code to also use Netty `ByteBufs` for data transfer.

Optimization Level	XMSS	LMS	SPHINCS$^+$
`jni-fixed-padding`	only `SHA-256`	only `SHA-256`	no

The optimization level `jni-fixed-padding` is limited to `SHA-256` and uses OpenSSL with fixed padding as described in the previous section. This is only possible for those functions whose input size is fixed and known in advance. This is the case for the majority of hash function calls in the HBSs. In all other cases, the custom padding implementation is used. Additionally, the custom padding function is used for LMS key compression, as there is numerous possible input sizes depending on the choice of parameters.

This approach could also be applied to SPHINCS$^+$. However, we again do not expect any further insight from this.

Optimization Level	XMSS	LMS	SPHINCS$^+$
`jni-prf-caching`	only `SHA-256` with $n = 256$	no	no

This optimization level builds upon the previous one and additionally applies the PRF caching strategy introduced in Section 4.1.4. This optimization is only applicable to XMSS with SHA-256 and $n = 256$.

For caching the states, the hash map described in Section 6.2.2 is used.

6.2.5. Java

This section introduces the optimization levels `java` and `java-optimized` that use the default implementation for SHA-2 and SHA-3 provided by the JDK. The state update function of these implementations is an intrinsic candidate. This means that Java implementation may be replaced by an equivalent native implementation by the Java runtime. For an introduction to the Java JIT and its compiler intrinsics, see Section 5.1.

As there is no existing implementation for Haraka in the JDK with a compiler intrinsic, we implement and evaluate a proof of concept that uses an intrinsic for the Haraka permutations.

Optimization Level	XMSS	LMS	SPHINCS$^+$
`java`	only SHA-256	only SHA-256	yes

SHA-2 and SHA-3 For the HBS instantiations with SHA-256 and SHAKE256, we use the respective message digest implementation provided by the default SUN JCA/JCE provider. As the implementation of SHAKE256 [72, Class `sun.security.provider.SHAKE256`] only outputs one block and does not implement the squeezing phase of the sponge construction, we modify it accordingly.

Similar to ACCP, this implementation is provided as a `MessageDigest` instance. Hence, we use the same approach as for ACCP. As described in Section 6.2.3, the `MessageDigest` is used to instantiate the three hash function implementation classes and an adapter class.

This approach can also be used for SHAKE256 in XMSS and LMS. However, we do not implement this variant as we do not expect any further insight (see Section 7.2.4).

Haraka As the Haraka hash function is rather new and not widely used, there is no existing implementation in the JDK. For this reason, we decide to create a Java implementation of Haraka with a JIT intrinsic. This requires changes to the JDK which are described below.

Our evaluation showed that introducing entirely new intrinsics is too complex and therefore out of scope for this work. Instead, we repurpose existing functions with an intrinsic by entirely replacing their implementation. This, of course, destroys the functionality these methods previously provided. As our implementation is intended for evaluation purposes only, this is acceptable. Furthermore, this has no influence on performance compared to introducing new intrinsics.

We implement three functions for Haraka: the hash functions Haraka256, Haraka512, and the permutation π_{512} which is used for the sponge construction HarakaS. We choose the methods to repurpose based on the method parameters. All three functions require the Haraka round constants and a bit string as input and return a bit string. Therefore, we

choose methods that have two byte arrays as parameters: one contains the round constants and the other one is used for the input and the returned bit string.

Haraka Function	Java Method (Package `com.sun.crypto.provider`)	Type Signature
Haraka256	`AESCrypt::implEncryptBlock`	`([BI[BI)V`
Haraka512	`ElectronicCodeBook::implECBEncrypt`	`([BII[BI)I`
π_{512}	`AESCrypt::implDecryptBlock`	`([BI[BI)V`

Table 6.1.: Repurposed methods for the Haraka intrinsic. For an explanation of Java type signatures, see [54, Chapter 3].

Table 6.1 gives an overview of the methods we re-implemented for Haraka. All parameters except the byte arrays are ignored. As we noted earlier, the JIT usually does not compile code on its first execution. Instead, the Java byte code is interpreted and the JIT is only invoked if the code is executed repeatedly. Therefore, it is not only necessary to change the intrinsics for the methods in Table 6.1 but also to change their Java implementation to also compute the corresponding Haraka function.

As the Java implementation is only used before the JIT is invoked, its performance is not critical. Hence, we integrate the Haraka implementation by BouncyCastle. We implement the intrinsic by translating the Haraka reference implementation [42] to the C++ code generator syntax used in OpenJDK.

Optimization Level	XMSS	LMS	SPHINCS$^+$
`java-optimized`	only `SHA-256`	only `SHA-256`	no

This optimization level contains changes similar to `bc` and `bc-optimized`: Instead of using the `MessageDigest` interface, the underlying `SHA-256` implementation is used directly. Additionally, the PRF caching optimization for XMSS with $n = 32$ is integrated.

Like with `bc-optimized`, the `SHA-256` implementation is not accessed via the JCA/JCE interface, but directly instead. However, the class `sun.security.provider.SHA2`, which contains this implementation, is not publicly visible. It is possible to invoke all methods without regard to their visibility over the Java Reflection API using the `setAccessible` method. However, the use of this functionality is generally discouraged. Additionally, we observe a performance penalty when invoking the `SHA-256` implementation via reflection. In our experiments, the JCA/JCE API is faster than directly calling the implementation using reflection.

Instead, we change the visibility of this class by modifying the JDK's source code. This allows us to directly use the implementation provided by this class without reflection.

6.3. Parallelization

Modern processors provide multiple cores and threads that can be used to concurrently execute code. Current HBS implementations are usually single-threaded and therefore do not fully utilize the available resources. In the following, we present how we parallelize the HBS implementations in BouncyCastle. We focus on key generation, as it is the most expensive operation for XMSS and LMS by a large margin. We focus on XMSS and LMS as parallelization of SPHINCS was extensively investigated by Sun et al. [67] and the results are largely also applicable to SPHINCS$^+$.

6.3.1. XMSS

Key Generation The main task of the XMSS key generation is to traverse the entire Merkle tree and calculate the value of the root node, which is part of the public key. For this task, the values of all nodes in the tree must be calculated. The Treehash algorithm [10] is commonly applied here because of its low memory requirements. The values of non-root nodes are usually discarded and later recalculated whenever they are required.

The leaves are derived from the public one-time signature keys, which in turn can be individually derived from a private seed in constant time. This allows us to easily parallelize the key generation as follows.

The task is divided into 2^i independent subtasks. For each task j, we calculate the root of the subtree with height $h - i$ that has the leaf nodes $j * 2^{h-i}$ through $(j + 1) * 2^{h-i} - 1$ of the original tree. This can be achieved by utilizing the Treehash algorithm. After that, we merge results by calculating the root node of the distinct subtree of height i that has the same root as the original tree. All the leaves of this subtree have been calculated in the previous step.

Figure 6.1 shows how an XMSS tree of height $h = 3$ can be split into four tasks with $i = 2$. Each task computes the root of its respective subtrees. In Figure 6.1, these are the dotted nodes. In the next step, these are uses to sequentially compute the value of the root node.

In practice, the XMSS key generation in BouncyCastle does not only calculate the root node but also initializes the state of the BDS algorithm [10] for tree traversal. It is used to traverse the tree to calculate the nodes required for signing later. The following information is needed to initialize the BDS state during key generation:

- The authentication path of the first leaf,
- A list `retain` of nodes on high levels in the tree,
- A list of Treehash instances initialized with the third node for each layer (when the layer is not stored in `retain`).

After running the subtasks, we can simply merge the created BDS states by:

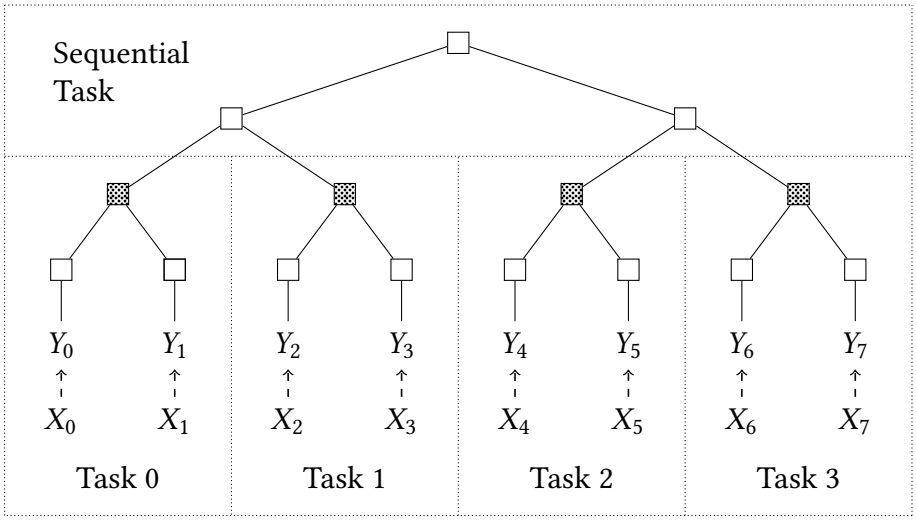


Figure 6.1.: Partitioning of the XMSS key generation with $h = 3$ into $2^i = 4$ independent tasks

- Using the authentication path of the first subtask,
- Merging the `retain` lists,
- Merging the Treehash instances by ignoring uninitialized instances. For each layer, there is no more than one initialized instance among the subtask results.

After merging the BDS states, we can continue to execute the key generation algorithm on the subtask roots and the merged BDS state. This yields the root node and a correctly initialized BDS state.

Signing and Verification The XMSS signing process consists of two main tasks: computing the WOTS signature and the authentication path. These are independent and can be done in parallel. Sun et al. [67] propose a parallelized version of WOTS. For each message block, the signature is computed in its own thread by applying the chaining function.

The WOTS verification can be parallelized in the same manner. However, the XMSS verification additionally includes the recalculation of the root node. This step depends on the WOTS key that is extracted during the WOTS verification. Therefore, these two tasks can not be executed in parallel.

However, creating threads comes at a cost. Splitting a task on many threads such that the amount of work per thread is small relative to the thread creation overhead leads to bad performance overall. Hence, parallelizing WOTS in BouncyCastle as suggested by Sun et al. [67] would require careful consideration and evaluation.

As noted above, we focus on key generation for this thesis and, therefore, do not present an implementation of parallelized signing and verification.

XMSS^{MT} Key Generation XMSS^{MT} uses a hypertree consisting of multiple XMSS trees. If large XMSS trees ($h \geq 10$) are used, XMSS key generation profits from the parallelization the same way XMSS does.

Some parameter sets use XMSS trees of height $h = 5$ [36, 14]. For example, the paramter `XMSSMT-SHA2_60/12_256` uses 12 layers of subtrees of height 5. Splitting the generation into $8 = 2^3$ subtasks results in the parallel generation of 8 subtrees of height 2 and sequential merging a subtree of height 3.

The cost of each subtask is rather small. As XMSS^{MT} requires the generation of multiple XMSS trees before the first message can be signed, it is possible to compute multiple trees in parallel. Each tree individually can be generated sequentially or, if sensible, in parallel with a smaller number of subtasks. This could reduce the thread creation overhead compared to sequentially executing multiple parallelized key generations.

6.3.2. LMS

Generally, the LMS key generation is structurally equivalent to XMSS and the observations in the previous section apply as well. However, BouncyCastle uses a different algorithm to traverse the LMS tree: Instead of using Treehash, it calculates a node's value by recursively calculating its children.

This can be easily parallelized using the Fork-Join pattern in Java: both child nodes are calculated concurrently and then merged to get the node's value. Due to synchronization overheads, this is only done for nodes on a high level, while nodes on a lower level are calculated sequentially. Testing showed that calculating nodes on level 5 (and lower) sequentially seems reasonable. However, even better performance might be achieved by optimizing this parameter through more extensive benchmarking.

6.4. Verification-Optimized Signatures

In the previous section, we presented strategies to optimize implementations of specific HBSs. This section focuses on verification-optimized scheme variants. They are designed to provide rapid signature verification at increased signature generation cost. This is especially useful in scenarios where only few signatures are created, but each signature is verified often.

One such scenario is digitally signed software and firmware: each new version could only be signed once, yet this signature is verified at each launch of the software or start-up of the device. Here, it is critical to optimize the signature verification, especially if the verification is done on a resource-constraint device. New signatures are only rarely created and may be computed on a powerful central server. Therefore, larger signature generation costs may be acceptable.

We focus on the underlying OTS as the verification of the OTS signature makes up the majority of the verification cost in a Merkle tree scheme. For common parameter choices, the verification of the Merkle tree itself only requires a few hash function evaluations. To the best of the author's knowledge, there are no proposals in the literature that explicitly optimize the Merkle tree for faster verification.

6.4.1. Theoretical Analysis

This section aims to provide a theoretical evaluation of WOTS-BR and WOTS+C for rapidly verifiable signatures.

6.4.1.1. WOTS-BR

WOTS-BR is introduced in Section 2.4.4 and was proposed by Perin et al. [57]. Bos et al. [8] integrate WOTS-R into XMSS and provide a detailed analysis. We focus on the WOTS-R modification and discuss the impact of the WOTS-B optimization at the end of this section.

Let $d = b_{l_1-1} \parallel \cdots \parallel b_0$ be the hash value of a message and define $S = \sum_{i=0}^{l_1-1} b_i$. The corresponding checksum is defined as:

$$\begin{aligned} C &= \sum_{i=0}^{l_1-1} (w - 1 - b_i) \\ &= l_1(w-1) - \sum_{i=0}^{l_1-1} b_i \\ &= l_1(w-1) - S. \end{aligned}$$

Let $(c_{l_2-1}, \ldots, c_0)$ be its base-w representation. The verification cost v of a signature is the required number of evaluations of the Winternitz chaining function f. It is the sum of the cost of verifying the message blocks v_m and the cost of verifying the checksum blocks v_c.

For a given message hash, the message block verification cost is

$$v_m = \sum_{i=0}^{l_1} (w - 1 - b_i) = l_1(w-1) - S$$

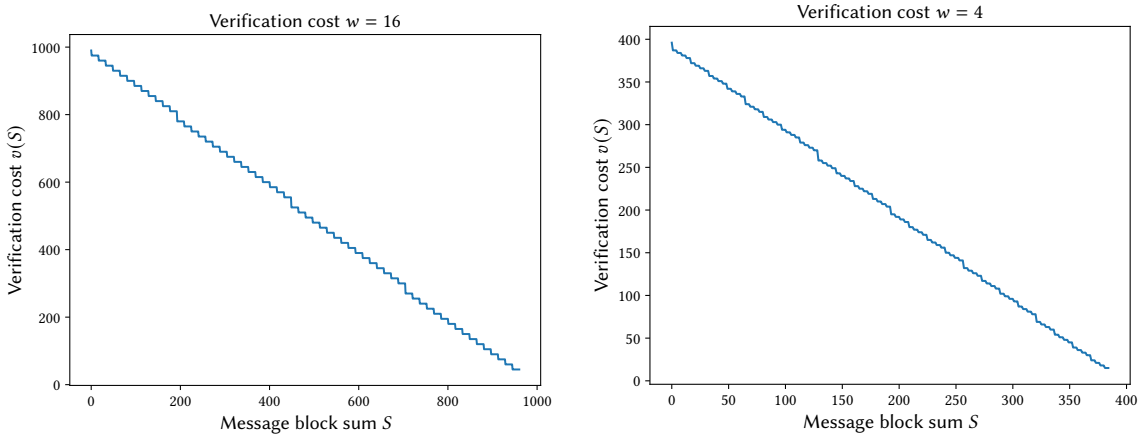
and the checksum verification cost is

$$v_c = \sum_{i=0}^{l_2} (w - 1 - c_i) = l_2(w-1) - \sum_{i=0}^{l_2} c_i.$$

The total verification cost is therefore

$$v = v_m + v_c = l(w-1) - S - \sum_{i=0}^{l_2} c_i.$$

Minimizing v We observe that the verification cost v only depends on the message block sum S. This is because the checksum is also fully determined by the value of S. Let $v(S)$ be the verification cost of a message with message block sum S. WOTS-R iterates over to multiple nonces to maximize S and therefore minimize $v_m(S)$. However, it is not immediately clear how $v_c(S)$ behaves and that the larger S leads to smaller $v(S)$, that is, that v is a monotonous function.



(a) $w = 16$ (b) $w = 4$

Figure 6.2.: WOTS Verification cost $v(S)$ by message block sum S for $n = 256$

Figure 6.2 shows the function v for two choices of parameters and for these, v is indeed a monotonous function. Furthermore, we can prove for all parameters that v is monotonous, that is, that $v(S+1) \leq v(S)$ holds for all S. Let C be the checksum for message block sum S and C' for $S+1$. We observe that there is only exactly one index i such that $c'_i = c_i + 1$ while for all other $j \neq i$ the condition $c'_j \leq c_j$ holds. It follows that $\sum_{i=0}^{l_2-1} c'_i \leq 1 + \sum_{i=0}^{l_2-1} c_i$ and therefore that $v(S+1) \leq v(S)$. For $c_0 < w-1$, the conditions $c_j = c'_j$ for all $j \neq 0$ and therefore $v(S+1) = v(S)$ holds. This means that v is a step function that changes in intervals of size w.

This result means that maximizing S yields in all cases the minimal message verification cost $v(S)$.

Modeling v Bos et al. [8] assume that the hash function g behaves like a random function. Hence, the b_i follow the uniform distribution $\mathcal{U}([0, w-1])$. They prove that the random

variable $X = \sum_{i=0}^{l_1-1}(b_i)/l_1$ has the mean $\mathbb{E}[X] = \frac{w-1}{2}$ and the variance $\mathbb{V}[X] = \frac{w^2-1}{12l_1}$. Therefore, they approximate the behavior of X and assume it follows the normal distribution $\mathcal{N}(\frac{w-1}{2}, \frac{w^2-1}{12l_1})$[1].

The sum of message blocks $S = l_1X$ therefore follows the distribution $\mathcal{N}(\mu_S, \sigma_S^2)$ for $\mu_S = \frac{l_1(w-1)}{2}$ and $\sigma_S^2 = \frac{l_1(w^2-1)}{12}$. To model hashing with R nonces, we define the random variables $S_{R-1}, \ldots, S_0$ with this distribution. As the nonce with the largest associated message block sum is chosen, we evaluate the expected verification cost for a message with sum $S_{max} = \max\{S_{R-1}, \ldots, S_0\}$.

Bos et al. show that the expected value of the message block verification cost is:

$$\begin{aligned}\mathbb{E}[v_m(S_{max})] &= \mathbb{E}[l_1(w-1) - S_{max}] \\ &= l_1(w-1) - \mathbb{E}[S_{max}] \\ &= \frac{l_1(w-1)}{2} - \Phi^{-1}\left(\frac{R-\alpha}{R-2\alpha+1}\right)\sqrt{\frac{l(w^2-1)}{12}}\end{aligned}$$

where Φ^{-1} is the inverse of the normal distribution $\mathcal{N}(0,1)$ and $\alpha = \frac{\pi}{8}$.

Calculating the expected verification for the entire signature for R nonces, $\mathbb{E}[v(S_{max})]$, is more challenging as it requires the expected verification cost for the checksum blocks, $\mathbb{E}[v_c(S_{max})]$:

$$\mathbb{E}[v(S_{max})] = \mathbb{E}[v_m(S_{max}) + v_c(S_{max})] = \mathbb{E}[v_m(S_{max})] + \mathbb{E}[v_c(S_{max})]$$

Bos et al. present two different approximations for $\mathbb{E}[v_c(S_{max})]$:

Random Checksum The checksum blocks $c_{l_2-1}, \ldots, c_0$ are assumed to follow the uniform distribution $\mathcal{U}([0, w-1])$ and to be independent of the message blocks $b_{l_1-1}, \ldots, b_0$. Therefore, the verification cost of the message signature blocks is independent of R and we have $\mathbb{E}[v_c(S_{max})] = l_2(w-1)/2$.

Worst-Case Checksum Bos et al. claim that maximizing the values of the message blocks leads to a lower average of the checksum blocks. Hence, the independence assumption above does not hold. Instead, $\mathbb{E}[v_c(S_{max})]$ is modeled as an upper bound for the verification cost of the checksum: $\mathbb{E}[v_c(S_{max})] = l_2(w-1)$.

This may be an acceptable approximation for large values of R as large iteration counts lead to high message blocks on average which leads to a low average of the checksum blocks and, therefore, to a verification cost close to the worst case.

[1] Bos et al. [8, Assumption 1] assume the normal distribution as $\mathcal{N}(\frac{w-1}{2}, \frac{w^2-1}{12})$. Yet, they prove $\mathbb{V}[X] = \frac{w^2-1}{12l_1}$ and present further calculations using this value. We expect this is merely a typing mistake in the variance used in the normal distribution.

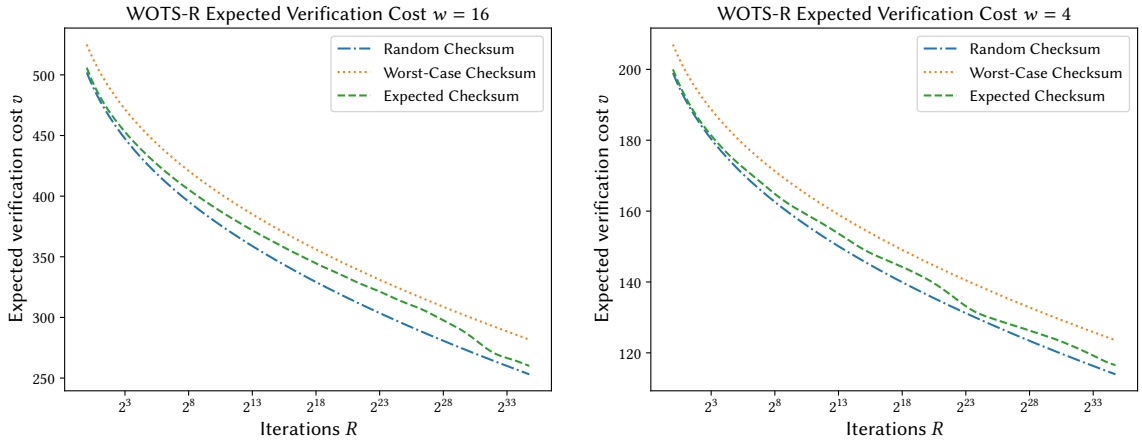
Expected Checksum Additionally, we present a way to specifically calculate $\mathbb{E}[v_c(S_{max})]$. Note that the random variable S_{max} takes values in $\{0, \ldots, l_1(w-1)\}$. Therefore, the definition of the expected value yields:

$$\mathbb{E}[v_c(S_{max})] = \sum_{s=0}^{l_1(w-1)} v_c(s) \cdot \Pr[S_{max} = s].$$

As S_{max} is a discrete random variable, we can determine the required probabilities as:

$$\begin{aligned}
\Pr[S_{max} = s] &= \Pr[S_{max} \leq s] - \Pr[S_{max} \leq s-1], \\
\Pr[S_{max} \leq s] &= \Pr[S_{R-1} \leq s] \cdot \ldots \cdot \Pr[S_0 \leq s] \\
&= (\Pr[S_0 \leq s])^R, \\
\Pr[S_0 \leq s] &= \Phi\left(\frac{s - \mu_S}{\sigma_S}\right)
\end{aligned}$$

where Φ is the distribution function of the normal distribution $\mathcal{N}(0,1)$.



(a) $w = 16$ (b) $w = 4$

Figure 6.3.: WOTS-R expected verification cost for different checksum models

Figure 6.3 shows the expected verification cost for the three models of the checksum. We observe that the verification cost with the expected checksum lies between the random and the worst-case checksum. For $w = 16$, it approaches the random checksum at about $R = 2^{31}$ and about $R = 2^{24}$ for $w = 4$.

WOTS-B Perin et al. [57] propose another modification of WOTS which we describe in Section 2.4.3. It fills the unused bits of the checksum with ones instead of zeroes. This increases c_{l_2-1} by a fixed amount and therefore reduces v_c by the same amount. The actual amount depends on the concrete choice of parameters. Overall, this optimization does not influence the verification cost beyond a constant reduction.

Let n_u denote the number of unused bits in the checksum. Setting these bits to one effectively increases the value of the highest-order checksum block c_{l_2-1} by $b = w - 2^{\log(w)-n_u}$. This change directly reduces the checksum verification cost v_c and thereby the overall verification cost by b.

Recall that $n_u = n_r - n_c = l_2 \log(w) - \lceil \log(l_1(w-1)) \rceil$. For $n = 256$ and $w = 16$, we have $l_1 = 64$, $l_2 = 3$, $n_u = 2$, and therefore $b = 12$. For $w = 4$, we have $l_1 = 128$, $l_2 = 6$, $n_u = 1$ and $b = 2$. Note that the impact of the WOTS-B optimization is smaller for smaller values of w as the number of checksum blocks l_2 can adapt better to the required number of checksum bits n_c.

6.4.1.2. WOTS+C

While the main intention behind WOTS+C is the reduction of the signature size, Kudinov et al. remark that, with the right choice of parameters, it could also be used for verification-optimized signatures [46]. In the following, we present a theoretical evaluation of WOTS+C for this purpose.

As introduced in Section 2.4.4, the scheme uses two parameters: a designated message block sum S and a number of zero-blocks z. A given message is hashed with nonces until the resulting hash value has the message block sum S and z zero-blocks on the left.

Given parameters S and z, we want to determine the expected number of nonces that have to be tested until a hash value is found that fulfills the requirements.

We define the sets $M_{w,l_1} = \{0, \ldots, w-1\}^{l_1}$ and $M_{w,l_1,S} = \{(m_{l_1-1}, \ldots, m_0) \in M_{w,l_1} \mid m_{l_1-1} + \cdots + m_0 = S\}$. Not that base-$w$ encoding of message hashes bijectively maps hash values to M_{w,l_1}. As a result, the number of possible hash values is equal to $|M_{w,l_1}| = w^{l_1}$.

The elements of $M_{w,l_1,S}$ directly correspond to the hash values with message block sum S. Assuming the hash function behaves like a random function, we find that the probability that a hash has the message block sum S is:

$$p_S = \frac{|M_{w,l_1,S}|}{|M_{w,l_1}|} = \frac{|M_{w,l_1,S}|}{w^{l_1}}.$$

Kudinov et al. [46] find that

$$|M_{w,l_1,S}| = \sum_{j=0}^{l_1} (-1)^j \binom{l_1}{j} \binom{(S+l_1) - jw - 1}{l_1 - 1}.$$

Additionally, they set $z_b = \log z$ and argue that z_b is the number of zero-bits the hash value must have. Therefore, the probability that a hash value has z zero-blocks is:

$$p_z = 2^{-z_b}.$$

Based on those two probabilities, Kudinov et al. [46] claim that the probability of finding a hash that fulfills both properties is:

$$p_{S,z} = p_S \cdot p_z.$$

By giving this formula, they implicitly assume that both events are statistically independent. However, this is not the case. Assume S' is the value that maximizes $p_{S',0}$. In this case, S' is also the expected value for the message block sum. According to the formula above, the same S' also maximizes $p_{S',l_1/2}$. This means that, when looking only at hash values where half the blocks are zero, the same message block sum S' still yields the highest probability. This is, however, not the case, especially, because the expected message block sum halves.

In the most extreme case, consider p_{1,l_1}. This is the probability that a given message hash has only zero-blocks and the sum of its blocks is one. Its value is zero, as this event is impossible. Still, the given formula yields a small, but non-zero probability.

Let $M_{w,l_1,S,z} = \{(m_{l_1-1}, \ldots, m_0) \in M_{w,l_1,S} \mid m_0 = \cdots = m_{z-1} = 0\}$. This is the set of the base-w encodings of all hashes that fulfill both properties.

We observe that

$$|M_{w,l_1,S,z}| = |M_{w,l_1-z,S}|.$$

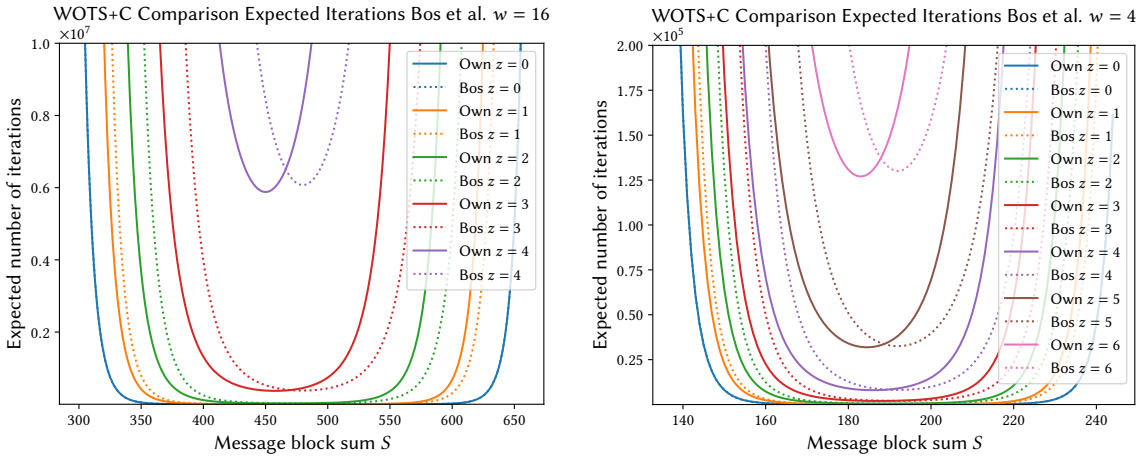
With this, we can correctly determine the probability $p_{S,z}$ as:

$$p_{S,z} = \frac{|M_{w,l_1,S,z}|}{|M_{w,l_1}|} = \frac{|M_{w,l_1-z,S}|}{w^{l_1}}.$$

Given $p_{S,z}$, the expected number of nonces that have to be tested until a matching hash is found can be trivially calculated as $\frac{1}{p_{S,z}}$.

Impact of Wrong Probability In this section, we discuss the impact of the flawed formula for $p_{S,z}$ given by Bos et al. [8]. First, we note that for $z = 0$, the event that the hash value has the required number of zero-blocks is always fulfilled. In this case, the calculation by Bos et al. is correct.

Figure 6.4 shows that the results diverge with growing values of z. Most importantly, the expected values S are different, that is, the value of S that minimizes the expected number of iterations. Bos et al. propose using WOTS+C for smaller signatures. Therefore, they chose the expected value of S. If this value is chosen based on the wrong assumption, the resulting scheme will perform worse than the preceding analysis suggested. For example, for $w = 16$ and $z = 4$, the value $S = 480$ would be chosen with an expected value of around 0.6×10^7 iterations. In reality, the expected value is about a third higher at around 0.8×10^7. Therefore, the scheme would perform about a third worse due to the parameter selection based on wrong assumptions.



(a) $w = 16$ (b) $w = 4$

Figure 6.4.: WOTS+C expected iterations by message sum for both models. For $z = 0$, both models are identical.

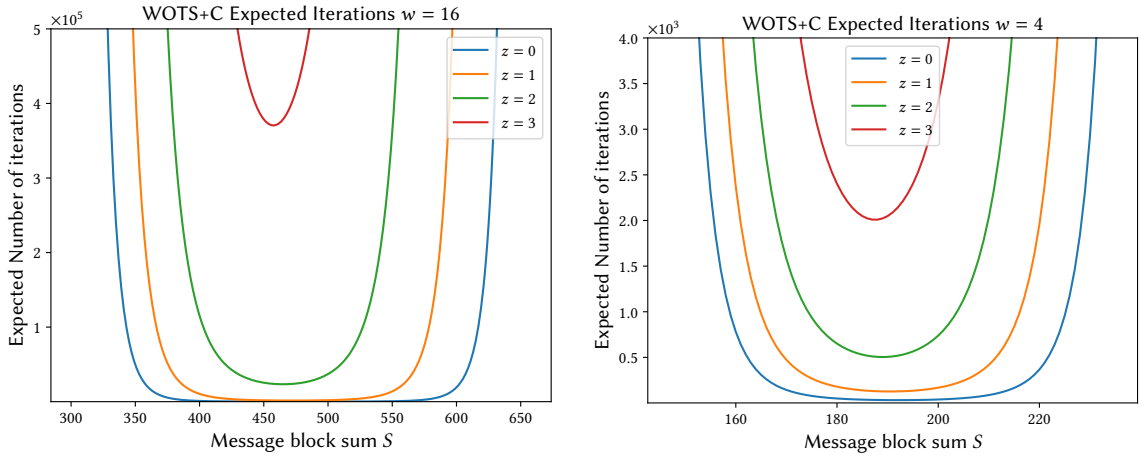
Bos et al. do not explicitly mention which value of z they use for SPHINCS+C. We assume that they chose $z = 0$. In this case, their formula is also correct. Therefore, we expect that the flawed assumption made by Bos et al. does not have any influence on their main contribution.

Performance impact of z In the following, we discuss the impact of the parameter z on the application of WOTS+C for verification-optimized signatures.

In this evaluation, we focus on the verification performance while reducing the signature size is not the main objective. Increasing z by one reduces the verification cost by $\frac{w-1}{2}$ on average as one signature block less has to be signed. At the same time, the size of the signature is reduced by n bits. However, the signing cost increases by a factor of approximately w. Note that this approximation is based on the same assumption of statistical independence discussed above.

However, this increased budget for signing could alternatively be used to instead increase S for an unchanged z. This would also reduce the signature cost. We want to evaluate which change provides the better result.

Additionally, we remark that a further possibility would be to increase z, which effectively decreases n and results in a smaller signature size. The reduction in the signature size could possibly be used to decrease the value of w while not increasing the signature size compared to the original signature size. This generally also improves the verification cost as it reduces the length of the Winternitz chains. For this thesis, we limit the Winternitz parameter w to powers of two. Therefore, this is not possible for reasonable choices of z, w, and l_1.



(a) $w = 16$ (b) $w = 4$

Figure 6.5.: WOTS+C expected iterations by message sum

Figure 6.5 shows the expected signature cost for parameters z and w. We recall that the verification cost is directly dependent on S: $v(S) = v_m(S) = l_1(w-1) - S$. As WOTS+C does not use an explicit checksum, we have $v_c(S) = 0$.

Therefore, we expect that increasing z by one and increasing S by $\frac{w-1}{2}$ lead to a similar reduction in the verification cost. Under this assumption, we find for the combinations of S, z, and w depicted in Figure 6.5 that it is generally better to increase S by $\frac{w-1}{2}$ than to increase z without changing S. At the same expected verification cost, it is possible to either increase the z by one or S by more than $\frac{w-1}{2}$.

We assume that this is generally applicable. Therefore, we set $z = 0$ for the rest of this thesis as this achieves the lowest verification cost.

6.4.1.3. Comparison

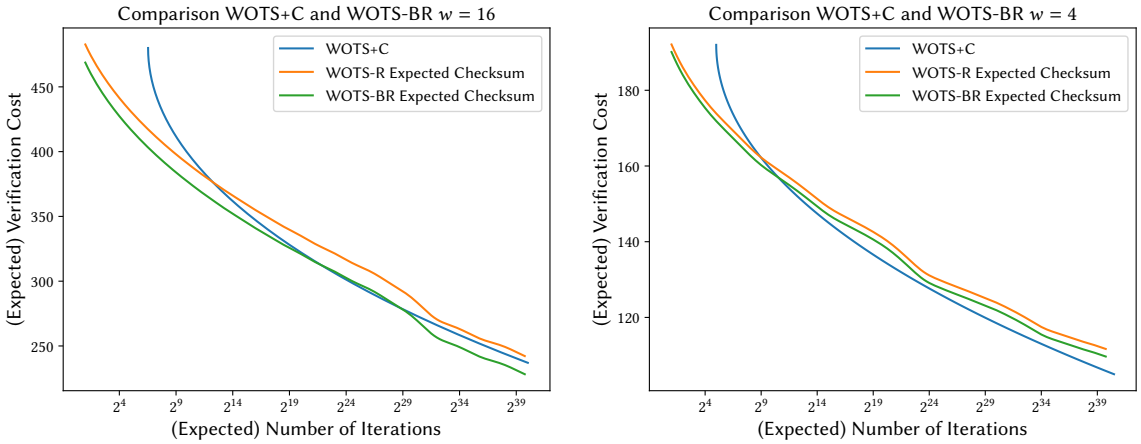
Comparability of the Iterations In WOTS-B, each iteration consists of computing the hash of the message with a new nonce and calculating the message block sum for this hash value. The resulting message block sum is then compared to the current best result and for most iterations, no further action is required.

In WOTS+C with $z = 0$, the iterations are very similar: a hash value with a new nonce and its message block sum are computed. The sum is then compared to S and if they are equal, no further iterations are required.

The iterations in WOTS-BR and WOTS+C are very similar. We, therefore, assume that iterations for both schemes have the same cost.

WOTS-BR To compare the schemes, we want to evaluate the message verification cost achieved on average for a certain number of iterations. For WOTS-R, this is straightforward once a checksum model has been chosen. This is depicted in Figure 6.3. For WOTS-BR, we simply reduce the expected message verification cost for WOTS-R by b.

WOTS+C For WOTS+C, we can calculate the expected number of iterations for a choice of S. As $v = l_1(w-1) - S$, we can compute the expected number of iterations by verification cost and plot this into the same diagram as the expected performance of WOTS-B and WOTS-BR.



(a) $w = 16$ (b) $w = 4$

Figure 6.6.: Comparison of WOTS+C, WOTS-R, and WOTS-BR

Figure 6.6 shows a comparison of the (expected) verification cost at a given (expected) number of iterations for WOTS+C, WOTS-R, and WOTS-BR. We recall that the expected verification cost for WOTS-R and WOTS-BR only differ by b. For small iteration counts, we find that both WOTS-R and WOTS-BR perform better than WOTS+C. This is mainly because even for the choice of S that minimizes the expected number of iterations, multiple iterations are required. This value is exactly the expected value of the message block sum for one hash value and therefore has the same verification cost as WOTS-R with $R = 1$ (neglecting the checksum).

However, for more than 2^{13} ($w = 16$) or 2^9 ($w = 4$) iterations, we expect WOTS+C to achieve better results than WOTS-R. Overall, the results of WOTS+C are close to or better than the over-optimistic random checksum model for WOTS-R. For more than 2^{32} iterations and $w = 16$, we find that the expected performance of WOTS-R approaches that of WOTS+C. We do not observe this behavior for $w = 4$.

For WOTS-BR and $w = 16$, we expect that WOTS-BR provides better results than WOTS+C for less than about 2^{20} iterations, comparable results for less than about 2^{30} iterations and better performance for even more iterations. We note that the last observation might not be practically relevant as such large signature costs may not be feasible in many application scenarios. For $w = 4$, the improvement of WOTS-BR over WOTS-R is smaller.

We find that, for more than about 2^{11} signatures, we expect WOTS+C to achieve better results than WOTS-BR.

Overall, we find that for most verification-optimized application scenarios, the scheme variant WOTS+C shows a better expected performance than WOTS-R. Comparing WOTS+C and WOTS-BR, it is harder to draw a general conclusion. For small choices of w, WOTS+C appears preferable. For larger values of w, a more detailed investigation based on the concrete choice of parameters is required.

Additionally, WOTS+C reduces the size of the signature by removing the checksum. Note that for a choice of parameters, WOTS-R and WOTS-BR guarantee the signing cost with variable verification cost while WOTS+C guarantees the verification cost with variable signing cost. Depending on the specific application scenario, one might be preferable over the other.

6.4.2. Practical Validation

This section describes how we implement WOTS-BR and WOTS+C to practically validate the theoretical results presented in the previous section.

6.4.2.1. WOTS-BR

Implementation We implement WOTS-BR as an extension of the WOTS implementation used by BouncyCastle for XMSS to allow for a possible integration of the WOTS variant into XMSS. Our implementation is parameterized by the number of iterations R and whether the WOTS-B padding optimization should be used. Additionally, we implement the option to select the message not by maximal message block sum S but by minimal total verification cost v. However, this option is unnecessary as described in Section 6.4.1.1 and we will therefore not evaluate it further.

As suggested by Bos et al. [8], the implementation initially invokes the update function of the hash function for the message. The resulting state is cached. For each nonce, the cached state is restored and the nonce is consumed to calculate the hash. Afterward, the message block sum S is calculated. If the S is larger than the current best value, S, the used nonce and the resulting digest are stored.

The implementation uses the 64-bit binary representation of the values $\{0, \ldots, R-1\}$ as nonces. Once all nonces have been used, the nonce with the maximum S is chosen and the resulting digest is signed by the regular WOTS implementation.

Benchmarking To evaluate WOTS-BR and our implementation, we measure two aspects: the runtime of the implementation and the verification cost of the generated signatures. We chose $R = 2^t$ for $t \in \{10, \ldots, 30\}$. For the majority of values of R, 1024 messages are signed. As the cost of the signing process grows exponentially in t, we reduce the number of messages signed for $t \geq 27$ to keep the runtime of the benchmark in a reasonable time frame.

We only benchmark WOTS-R and derive the message verification cost for WOTS-BR by subtracting b. For a given number of iterations R, the runtime difference of both schemes is negligible.

The used messages are generated pseudorandomly by hashing the number of the message.

6.4.2.2. WOTS+C

Implementation To validate the theoretical evaluation of WOTS+C, we implement the scheme similar to WOTS-BR as an extension of the WOTS implementation for XMSS in BouncyCastle. It uses the parameters S and z, yet we set $z = 0$ as elaborated above. Again, we use the state caching strategy to compute hashes for multiple nonces at a minimal cost.

Benchmark In the benchmarks for WOTS+C, we once more measure the runtime of the implementation. As the verification cost of the resulting message is directly determined by the parameter S, it is not necessary to measure it. Instead, we measure the number of nonces that has to be tested before a suitable hash value is found.

For $w = 16$, 1024 messages with S increasing in steps of 10 between 480 and 670. Additionally, 256 messages are signed for $S = 680$ and $S = 686$. For $w = 4$, we evaluate 1024 messages for S between 192 and 156 with a step size of 4 and 256 messages for $S = 260$, $S = 264$, and $S = 266$.

6.4.2.3. Comparison of Iterations

In Section 6.4.1.3, we assume that iterations of WOTS+C and WOTS-R require the same amount of time and are therefore comparable. We want to experimentally confirm this.

For this, we select a message and a value S and observe the number of required iterations. This value is then chosen as the parameter R for WOTS-R. In this configuration, both schemes make the same number of iterations and therefore should have a similar total runtime.

7. Evaluation

This chapter presents the results of the implementation and, therefore, largely follows the structure of the previous chapter: first, we present the benchmark results for the XMSS reference implementation. Second, we present the benchmark results for the different optimization levels in BouncyCastle. We continue by giving results for the parallelization of the HBS key generation and finish by presenting the validation of the evaluation of the verification-optimized scheme variants.

7.1. XMSS Reference Implementation

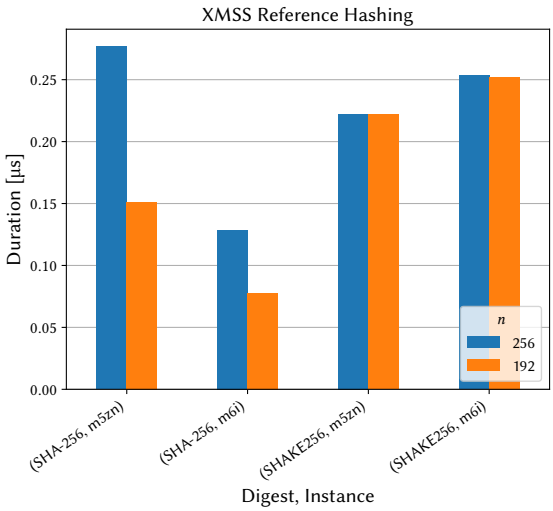


Figure 7.1.: Benchmark results for hashing with the XMSS reference implementation

Figure 7.1 shows the results of the hash benchmark with the XMSS reference implementation described in Section 6.1. We present benchmark results for two different EC2 instances: `m5zn` and `m6i`. Recall that our previous tests in Section 5.3.2 showed that the `m5zn` instances are generally faster, but do not support SHA-NI.

The input sizes are equivalent to the ones used for the Winternitz chaining function F as specified in Section 3.3.1. For $n = 256$, this is 768 bits, and 416 bits for $n = 192$.

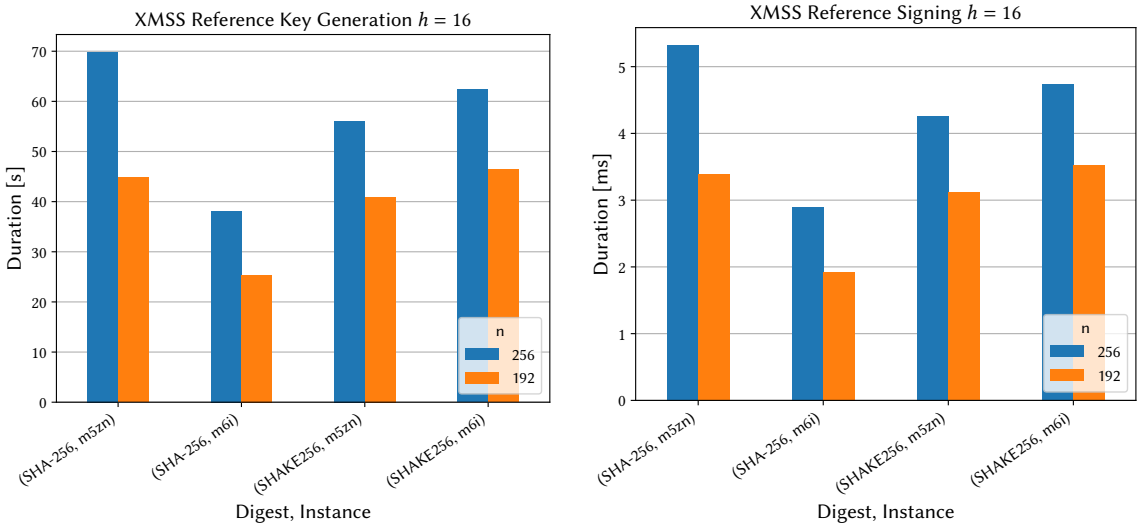
For SHA-256 with a block size of 512 bits, this means that two blocks have to be processed for $n = 256$ and only one for $n = 192$. Each processed block requires one evaluation of the

SHA-256 compression function. This is recognizable in the results we see: on both m5zn and m6i, there is almost a factor of 2 difference between the two choices for n.

SHAKE256 uses a block size of 1088 bits, hence for both choices of n only one block must be processed. This matches our benchmark results: on both instance types, $n = 192$ is only insignificantly faster.

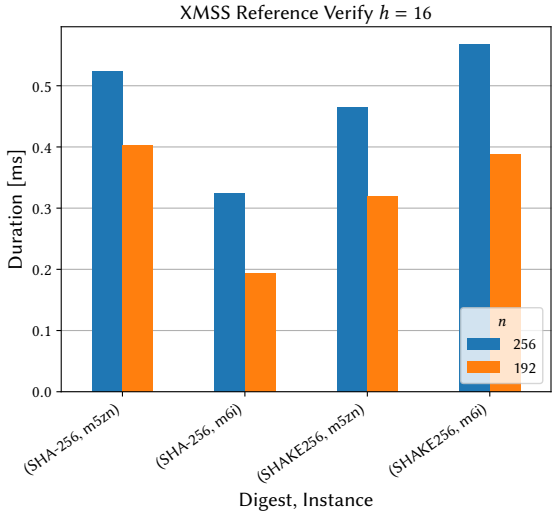
Comparing the results across instance types, we observe that m6i is slower than m5zn for SHAKE256. This coincides with our observations in Section 5.3.2.

For SHA-256, m6i only requires half the time of m5zn. The XMSS reference implementation relies on OpenSSL for SHA-256. OpenSSL uses SHA-NI if they are available. As this is the case on m6i, SHA-256 is significantly faster on this instance type.



(a) Key Generation

(b) Signing



(c) Verification

Figure 7.2.: Benchmark results for the XMSS Reference Implementation with $h = 16$

Figure 7.2 shows the benchmark results for key generation, signing, and verification with the XMSS reference implementation. We observe that the results for all operations are relatively similar.

The performance gain for SHA-256 when reducing n from 256 to 192 is lower than the results for hashing presented above. This is mainly due to two factors: the XMSS operations contain other operations whose run-time is independent of n, like the allocation and management of the addresses. Additionally, the hash results above take only the Winternitz chaining function F into account. Other functions may behave differently. For example, PRF requires the same number of blocks for both values of n.

For SHAKE256, we observe a different effect: Though hashing with F takes the same time for both choices of n, the operations are overall faster for $n = 192$. This is primarily due to non-hashing operations whose runtime still depends on n, e.g. copying data between buffers.

7.2. Optimization Levels

This section presents the benchmark results for the integration of the different optimizations proposed in Section 6.2 into BouncyCastle's implementation of HBS, and compares them to the default BouncyCastle implementation and the reference implementation (XMSS only).

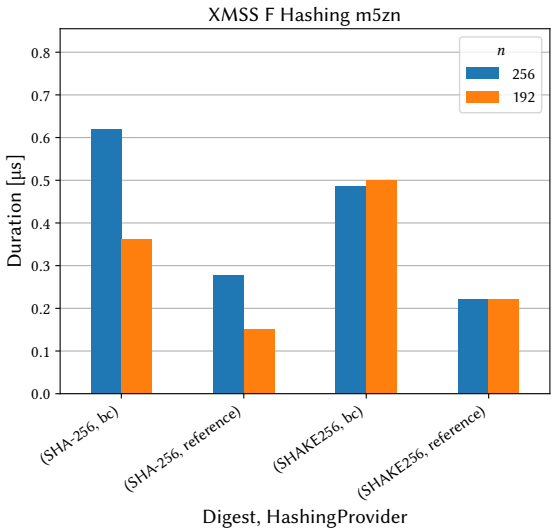
7.2.1. BouncyCastle

Level bc Figure 7.3 shows the benchmark results for XMSS for the reference implementation and unoptimized BouncyCastle. We observe that BouncyCastle behaves similarly to the reference implementation: For SHA-2, reducing n to 192 almost halves the cost of hashing. In the XMSS operations, this effect is also noticeable, yet at a lower factor than 2. For SHAKE, n has little influence on hashing with F, but overall, $n = 192$ performs better in the XMSS operations.

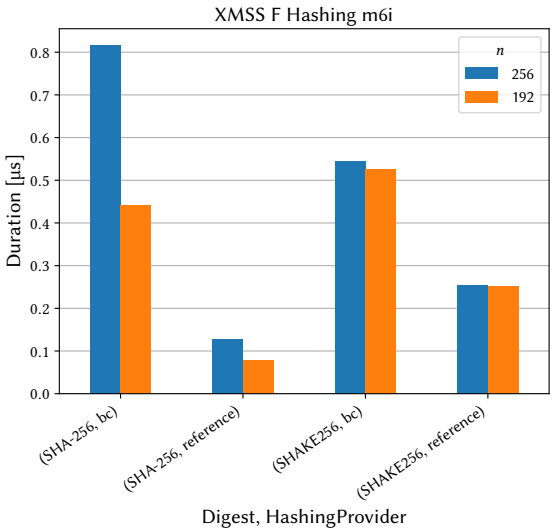
Comparing `bc` to the reference implementation, we observe that there is usually a factor of about two or larger between both implementations. This applies to hashing, key generation, and verification. For signing, this effect is slightly dampened. This might be caused by a different implementation or instantiation of the BDS algorithm for tree traversal which is an important part of the signing operation (see Section 2.5).

Furthermore, on `m6i`, the reference implementation with SHA-2 performs better by a factor of up to seven. This is due to the use of SHA-NI on this instance type.

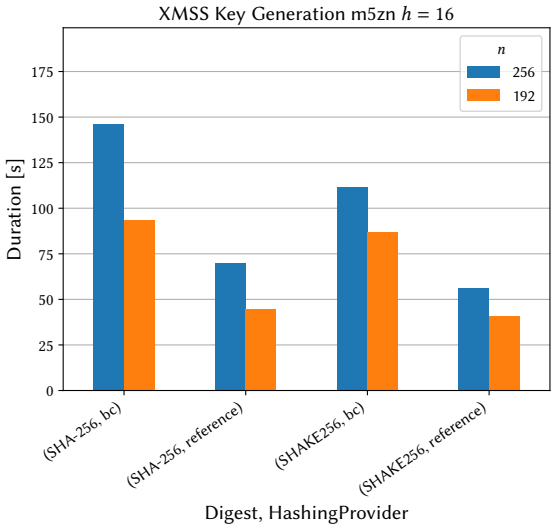
Overall, we draw two conclusions from these results: First, hashing and XMSS operations behave qualitatively similarly when comparing different hash implementations and instances for the same parameter set. Therefore, we will only present results for hashing for the rest of the optimization levels. Additionally, we show results for the key generation if



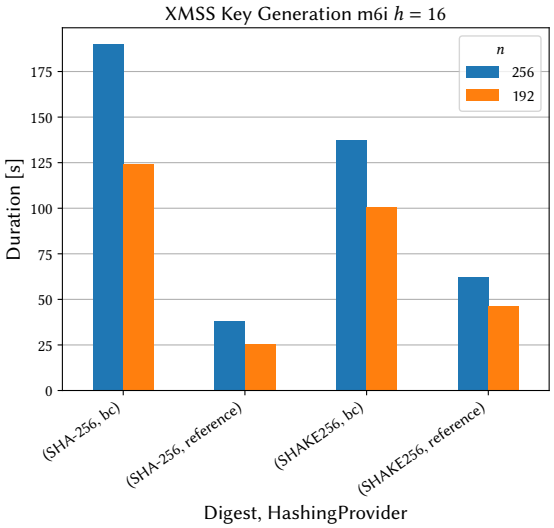
(a) Hashing on `m5zn`



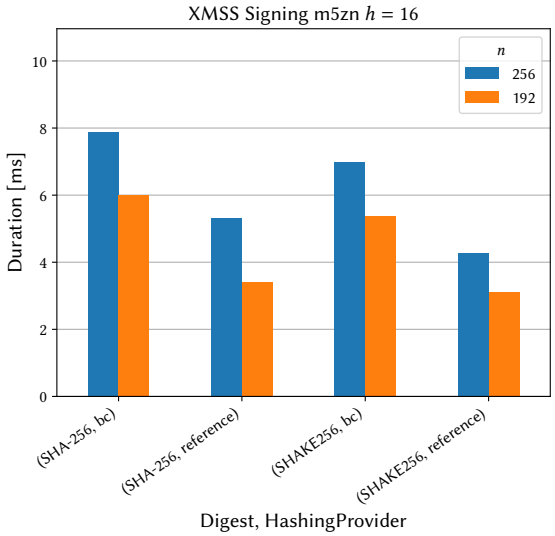
(b) Hashing on `m6i`



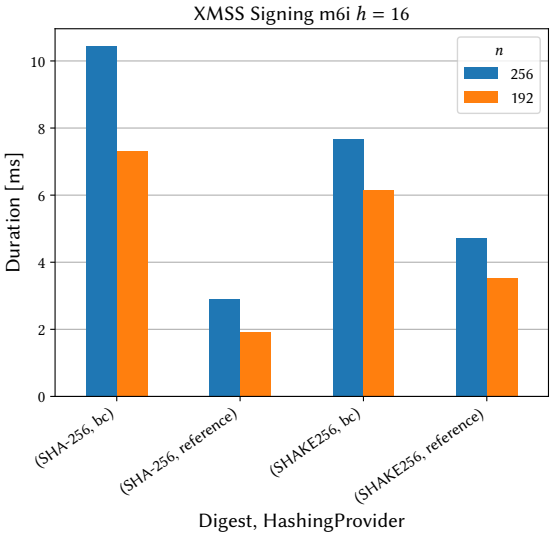
(c) Key Generation with $h = 16$ on `m5zn`



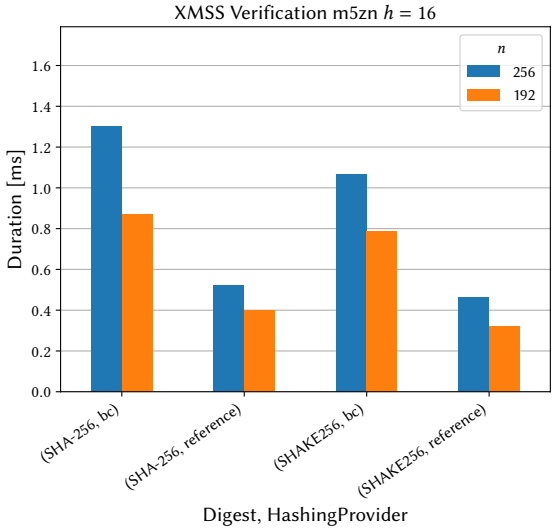
(d) Key Generation with $h = 16$ on `m6i`



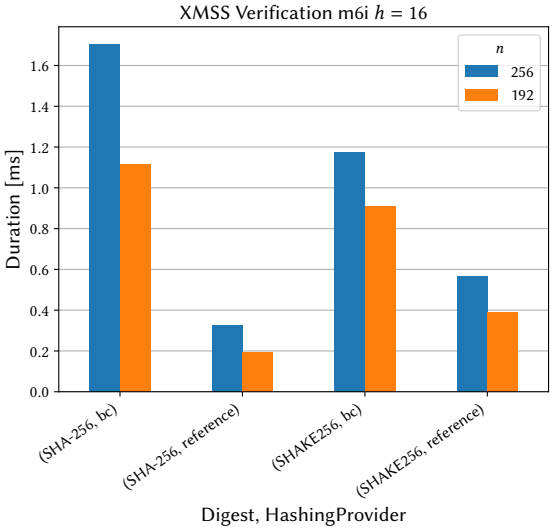
(e) Signing with $h = 16$ on `m5zn`



(f) Signing with $h = 16$ on `m6i`



(g) Verification with $h = 16$ on `m5zn`

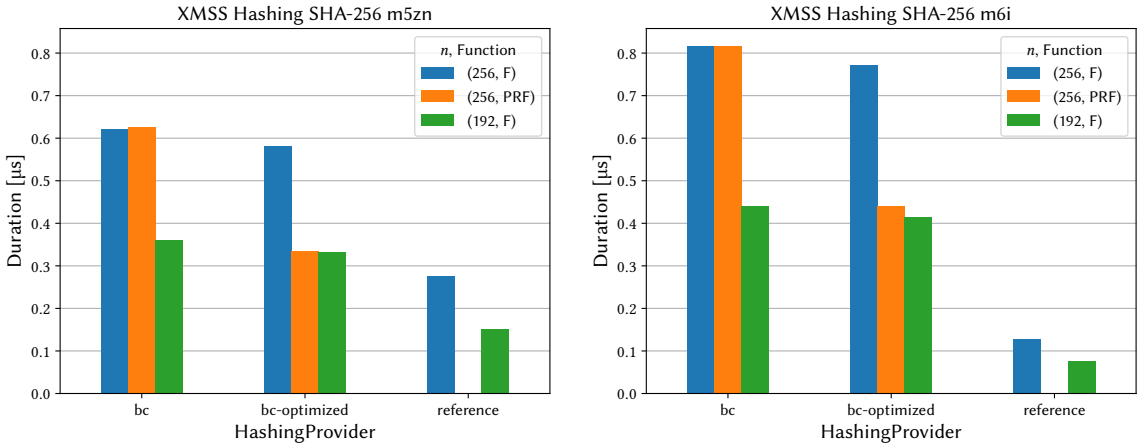


(h) Verification with $h = 16$ on `m6i`

Figure 7.3.: Benchmark results for the XMSS with the `bc` optimization level

they can not be directly inferred from the hashing results. Further results are available in Appendix A.

Secondly, we note that BouncyCastle performs significantly worse than the reference implementation and there is a large potential for optimization.



(a) `m5zn` (b) `m6i`

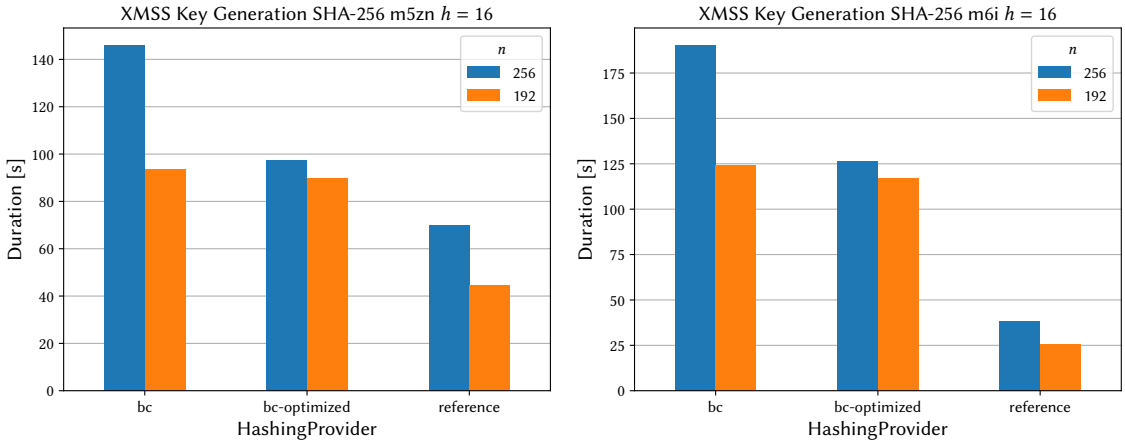
Figure 7.4.: Benchmark results for XMSS hashing with SHA-256 with the optimization level `bc-optimized`

Level bc-optimized Figure 7.4 shows the benchmark for hashing with the optimization level `bc-optimized`. This level only implements SHA-256. Unlike the previous hash benchmarks, we additionally simulate the function PRF for $n = 256$. Without the PRF caching optimization, PRF behaves exactly like F as the input lengths are equal. Therefore, we only expect a difference between F and PRF for `bc-optimized` and $n = 256$. For this reason, we do not provide benchmark results for PRF with $n = 192$ or `reference`.

For F, we find that the optimizations bring a small, yet still noticeable improvement of 6.3% on `m5zn` and 5.5% on `m6i` over `bc`. Nevertheless, the PRF caching optimization considerably speeds up PRF. While PRF for $n = 256$ takes as long as F for $n = 192$ with `bc`, it only takes as long as F for $n = 192$ for `bc-optimized`. This represents a reduction by 46% and is the behavior we expect from PRF caching: Without PRF caching, two SHA256 blocks must be evaluated for each invocation of PRF. Caching reduces this to one block. For $n = 192$, the function F only requires one block as well.

Figure 7.5 shows our measurements for key generation. For XMSS with $n = 192$ and LMS the optimizations only bring a minor improvement over `bc` of up to 5.5% and 2.1%, respectively. For XMSS with $n = 256$, we see significantly boosted performance due to the PRF caching. Overall, we observe that PRF caching allows using $n = 256$ with a level of performance comparable to $n = 192$.

The other optimizations have only little influence on the performance. However, the relatively minor changes made still have an effect. We conjecture that a SHA-256 imple-

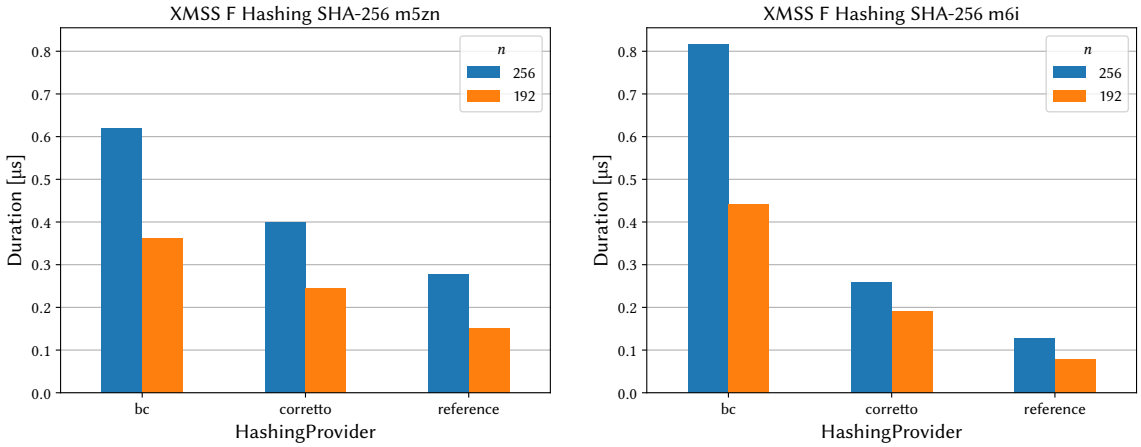


(a) `m5zn` (b) `m6i`

Figure 7.5.: Benchmark results for XMSS key generation with the `bc-optimized` optimization level and $h = 16$

mentation dedicated to short-input hashing for HBS might be able to provide another small improvement over the general-purpose implementation in BouncyCastle.

7.2.2. Amazon Corretto Crypto Provider



(a) XMSS Hashing on `m5zn` (b) XMSS Hashing on `m6i`

Figure 7.6.: Benchmark results for XMSS hashing with SHA-2 and the `corretto` optimization level

The benchmark results for hashing with `corretto` compared to `bc` and the XMSS reference implementation are visualized in Figure 7.6. We find that the optimization level `corretto` provides a significant improvement over `bc`. On `m5zn`, it reduces the runtimes by up to 36%. On `m6i`, this is even more noticeable: `corretto` provides an improvement by 68% for $n = 256$ over `bc`.

Even with this improvement, `corretto` still performs considerably worse than the reference implementation. On `m6i`, it is worse by a factor of two. Hence, there may still be room for further improvement.

Once more, the hashing performance is well reflected in the performance of the HBS key generation operations as shown in Appendix A. For example, `corretto` improves the XMSS key generation by 64% for $n = 256$ on `m6i`.

Overall, we observe that using ACCP brings a performance improvement, but the XMSS reference implementation is still considerably faster.

7.2.3. JNI

This section presents our benchmark results for JNI described in Section 6.2.4. Specifically, we start by giving the results for the JNI data transfer, the evaluated native hash implementations, and finally the achieved performance when using the combination of the best data transfer method and the best hash implementation in HBSs.

7.2.3.1. JNI Data Transfer

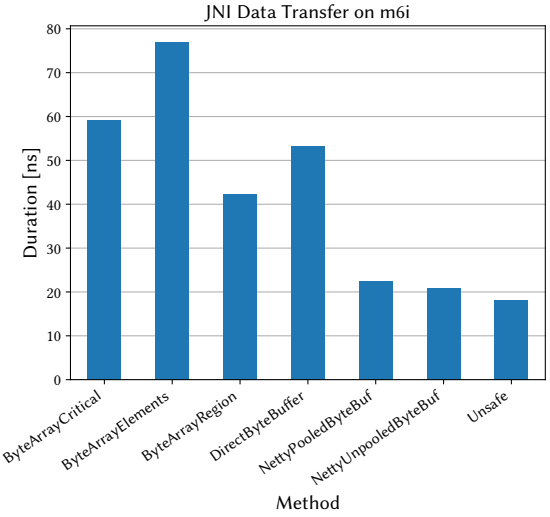


Figure 7.7.: Benchmark results for the JNI data transfer benchmark on `m6i`

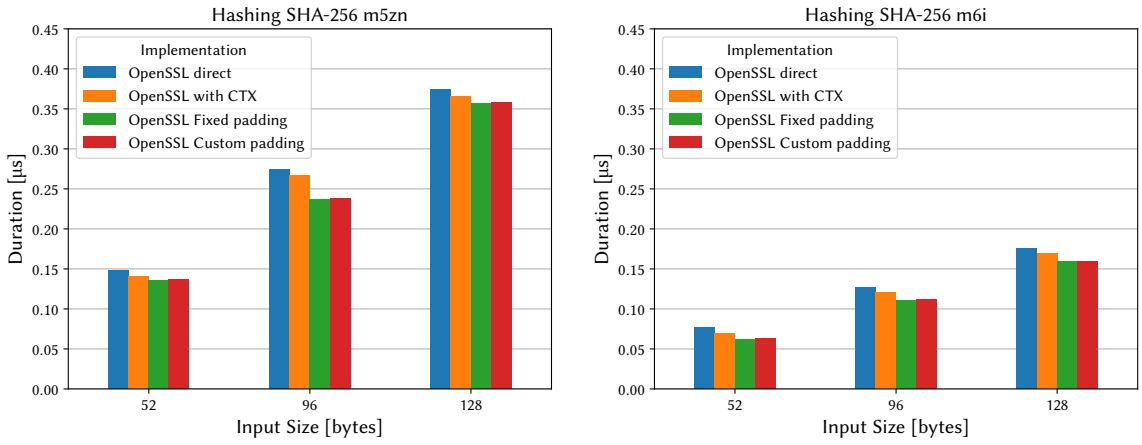
Figure 7.7 shows the results of the JNI transfer benchmark. The measurements include copying the data from a Java byte array to another buffer (if necessary), calling the native void, adding each pair of elements in the native code, returning to Java, and copying the data from the buffer to a Java byte array, if necessary. The initial allocation of the buffers is not part of the benchmark. However, the buffers may be reused and, therefore, the initial allocation is only required once.

Overall, we find significant differences in performance depending on the chosen transfer method. All three JNI methods and the Java `ByteBuffer` perform rather badly. Both Netty `ByteBufs` and the `Unsafe` approach are quite comparable. The fastest JNI method, `GetByteArrayRegion` and `SetByteArrayRegion`, takes more than twice as long as using `Unsafe`.

The Netty Pooled `ByteBuf` is backed by a Java direct `ByteBuffer`. However, data is not written to the `ByteBuffer` using the methods provided by it. Rather, the `Unsafe` class is used to read from and write to the memory backing the `ByteBuffer`. Additionally, the Netty `ByteBuf` does proper bounds checking. This explains why the performance of the Netty `ByteBufs` and `Unsafe` are similar.

We proceed using the Netty unpooled `ByteBuf` as it provides better performance than pooled `ByteBufs`. Additionally, it properly encapsulates the use of `Unsafe` and, therefore, it is not necessary to directly use `Unsafe`. We consider the small performance penalty over `Unsafe` to be acceptable for this benefit.

7.2.3.2. Choosing the Hash Implementations



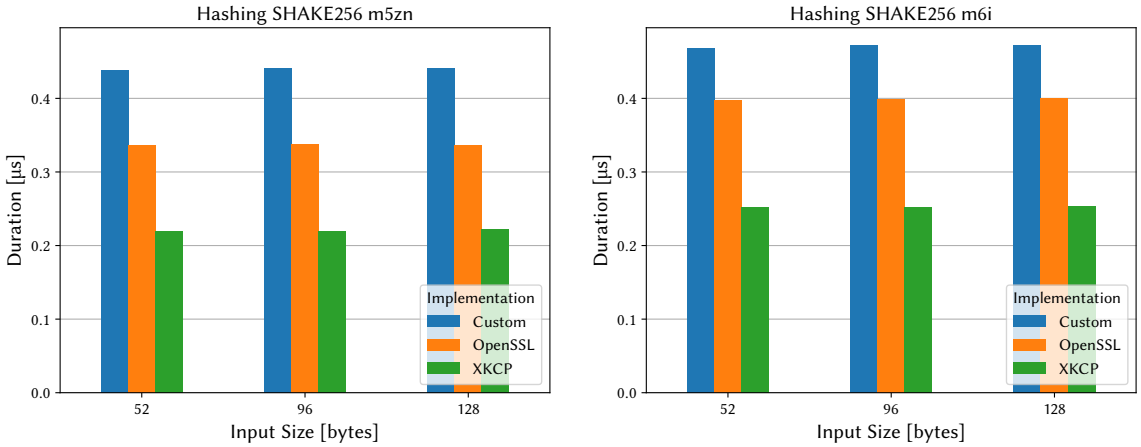
(a) `m5zn` (b) `m6i`

Figure 7.8.: Benchmark results for different OpenSSL interfaces and padding implementations for SHA-256

SHA-256 The benchmark results for SHA-256 are shown in Figure 7.8. One block has to be processed for an input size of 52 bytes, two blocks for 96 bytes, and three blocks for 128 bytes. This is well reflected in the results. We observe that the differences between the implementations are relatively small. Across all input sizes and platforms, the direct OpenSSL interface is the slowest, followed by OpenSSL with context. Both approaches with external padding perform better than using the finalize function provided by OpenSSL. This may be because, with external padding, it is not necessary to copy the data of the last block into the internal buffer until the finalize function is invoked. Additionally, fewer calls into OpenSSL are required.

Overall, the hard-coded padding performs slightly better than the custom padding that is calculated on the fly. However, hard-coded padding like this can only be applied if the input sizes are known in advance. We decide to further evaluate both approaches in separate optimization levels to observe their impact on the HBS.

We remark at this point that another possibility would be to compute the padding on the fly, but cache it for later use. This approach should provide a performance level similar to the hard-coded padding while eliminating the need for the input sizes to be known in advance.



(a) `m5zn` (b) `m6i`

Figure 7.9.: Benchmark results for different native SHAKE256 implementations

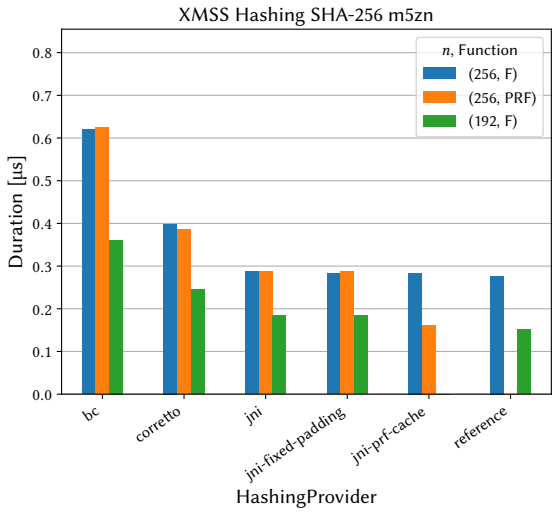
SHAKE256 Figure 7.9 depicts our benchmark results for SHAKE256. First, we note that the run times appear to be independent of the input size. Due to the block size of 1088 bits, only one block has to be processed for all the evaluated input sizes. There are significant differences in performance for the different implementations. The custom implementation is the slowest while XKCP only requires about half the amount of time on `m5zn`. The performance of OpenSSL is somewhere between these two, depending on the platform.

As OpenSSL and XKCP use a closely related underlying implementation, this shows the impact of the OpenSSL EVP interface on short-input hashing. Hence, we use XKCP for the further implementation.

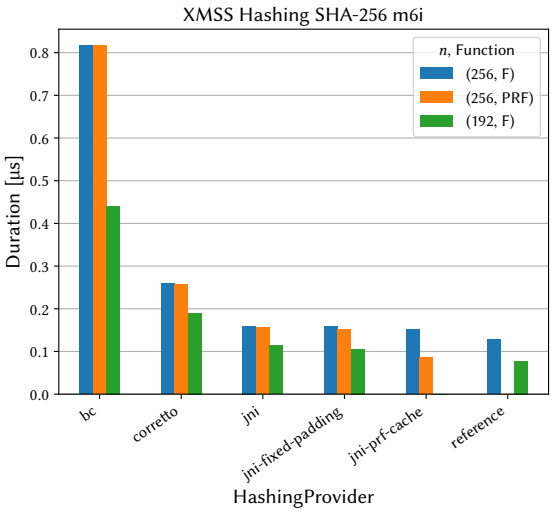
7.2.3.3. Implementation of jni-hash

SHA-2 Figure 7.10 shows the benchmark results for XMSS hashing and key generation with the optimization levels `jni`, `jni-fixed-padding`, and `jni-prf-cache`.

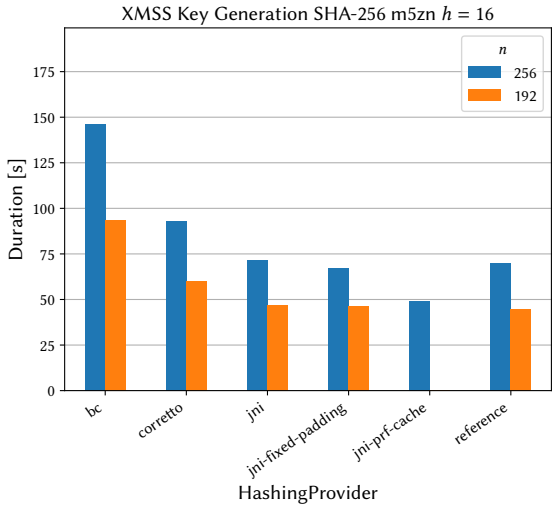
For hashing, we find that `jni` is significantly faster than `corretto`. Compared to `bc`, this optimization level reduces the runtime of a hash operation with F by up to 53.7% on `m5zn` and 80.4% on `m6i`. As in Section 7.2.3.2, the addition of the fixed padding optimization only



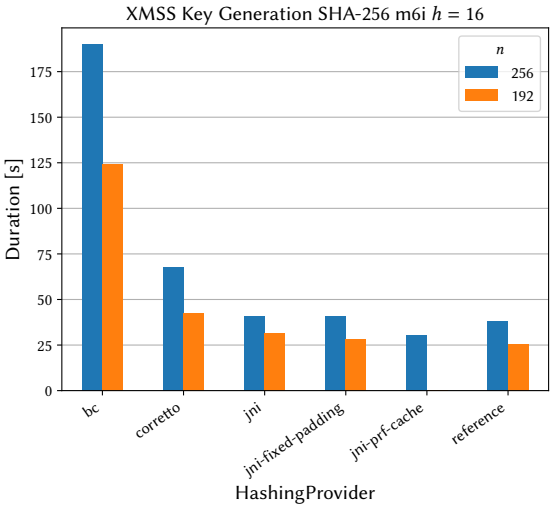
(a) XMSS Hashing on `m5zn`



(b) XMSS Hashing on `m6i`



(c) XMSS Key Generation on `m5zn` with $h = 16$



(d) XMSS Key Generation on `m6i` with $h = 16$

Figure 7.10.: Benchmark results for XMSS hashing and key generation with SHA-256 and `jni-hash`

brings a slight improvement over jni bringing the overall reduction to 54.3% and 80.7%, respectively.

We reiterate that for all shown optimization levels except jni-prf-cache, we expect F and PRF to behave identically as they operate on the same input size. The optimization level jni-prf-cache improves upon jni-fixed-padding by significantly reducing the cost of PRF. This is due to the PRF caching optimization and comparable to the observations for PRF caching in bc-optimized presented in Section 7.2.1. On m5zn, jni-fixed-padding performs roughly on par with the reference implementation while it performs slightly worse on m6i. Compared to PRF with bc, the combination of JNI and the fixed padding optimization results in an improvement of 74.1% and 89.5%, respectively.

The results for the key generation once more directly reflect the behavior of F for jni and jni-fixed-padding. Overall, the performance of these optimization levels is comparable to the reference implementation. For jni-prf-cache, we find a significant improvement over jni and jni-fixed-padding. This means that XMSS key generation for $n = 256$ is also faster than the reference implementation. Compared to the reference implementation, jni-prf-cache reduces the required time for the key generation by 28.8% on m5zn and 20.4% on m6i. Relative to bc, this is 66.2% and 84.1%.

The smaller savings on m6i compared to the XMSS reference implementation may be because hashing is considerably faster, yet the cost introduced by the use of JNI is comparable on both platforms.

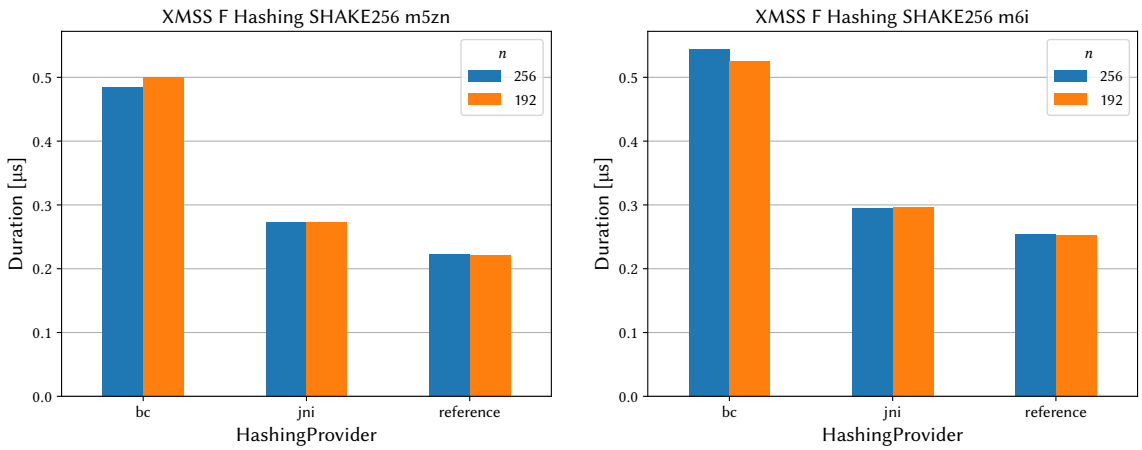
In conclusion, we find that the PRF caching optimization in combination with a fixed padding or a custom padding implementation is more than able to compensate for the cost of the JNI function calls. As a result, we can achieve better performance from a Java implementation than the XMSS reference implementation.

However, we note that it would be easy to integrate the PRF caching optimization into the reference implementation. For $n = 24$, it would be possible to achieve better performance by using a custom padding implementation as described in Section 6.2.4.2.

For LMS, we observe similar results. For example, jni-fixed-padding reduces the key generation time on m6i by up to 78.4% (see Figure A.10). We expect to see worse results for LMS than for XMSS because the PRF caching optimization can not be used with LMS. We do not present benchmark results for SPHINCS$^+$ with SHA-2 via JNI as we did not implement this variant.

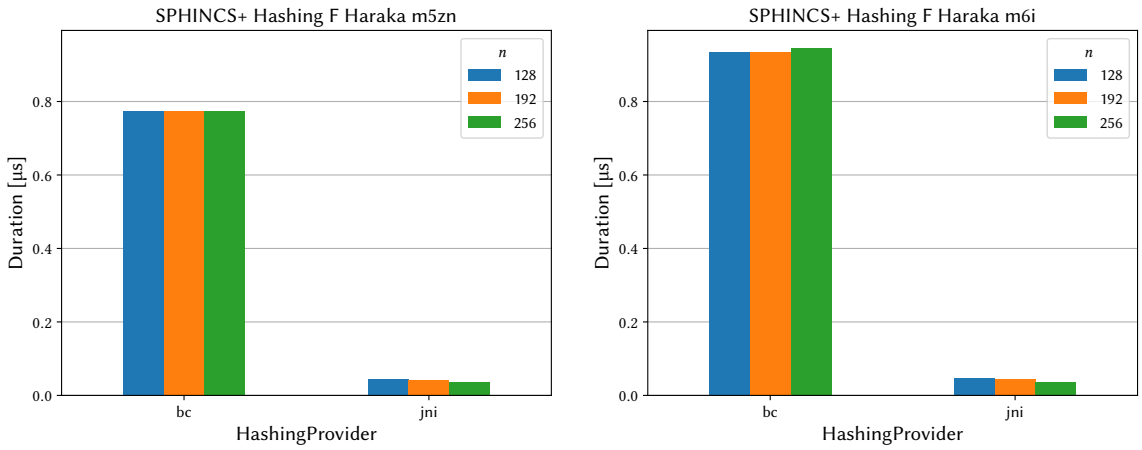
SHAKE256 Figure 7.11 shows the performance of XMSS hashing with SHAKE256. The optimization level jni improves the hashing cost by up to 45.4% on m5zn and 45.9% on m6i relative to bc. However, it still is noticeably slower than the reference implementation. Unlike SHA-256, we are not aware of any further optimizations that could be applied to jni.

XMSS key generation with SHAKE256 behaves similar to hashing and we measure a reduction of up to 43.1% on m5zn (see Figures A.3 and A.4). We refrain from presenting the benchmark results at this point and refer to Appendix A.



(a) `m5zn` (b) `m6i`

Figure 7.11.: Benchmark results for XMSS hashing with `SHAKE256` and `jni`



(a) `m5zn` (b) `m6i`

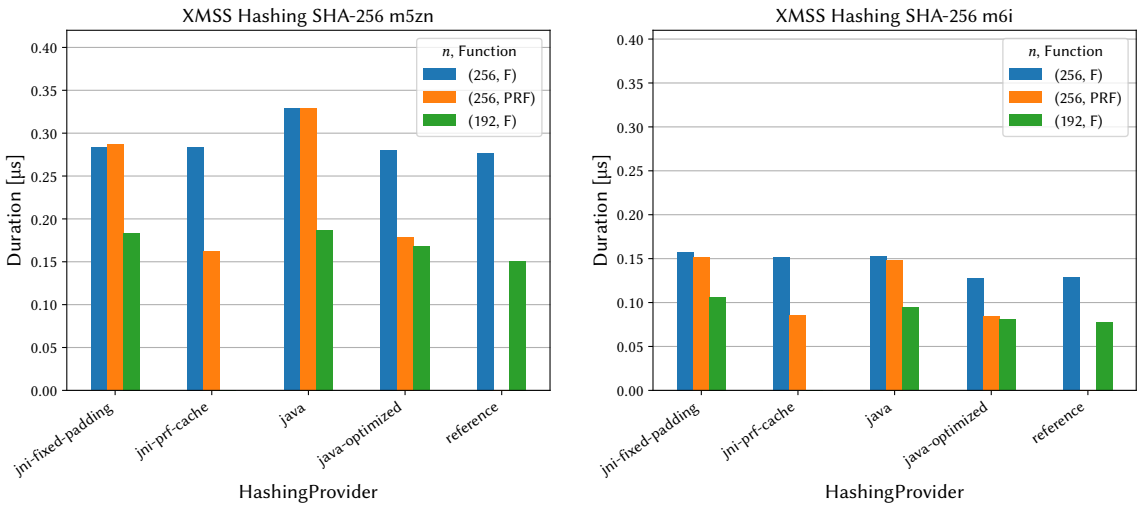
Figure 7.12.: Benchmark results for SPHINCS$^+$ hashing with Haraka and `jni`

Haraka Figure 7.12 compares the performance of the Winternitz chaining function F in SPHINCS$^+$ for all specified values of n with Haraka as the underlying hash function. Using `jni` provides a reduction of up to 96.1% compared to `bc`. This is mainly due to the use of the AES-NI as described in Section 4.1.2.

As each invocation of F requires one evaluation of the permutation π_{512}, we expect the runtime of Haraka to be independent of n. This matches the benchmark results for `bc`. However, the runtime of Haraka with `jni` decreases as n increases. We conjecture that this might be caused by the use of one Netty `ByteBuf` for transferring both input to and output from JNI. For $n = 256$, the input fills the entire buffer. For smaller values of n, this is not the case, and it is necessary to explicitly clear the rest of the buffer because it may still contain output data from the last invocation. This might lead to worse performance for $n < 256$.

For key generation, we observe an improvement of up to 95.3% compared to `bc`. We again do not present further benchmark results at this point and refer to Appendix A.

7.2.4. Java



(a) `m5zn` (b) `m6i`

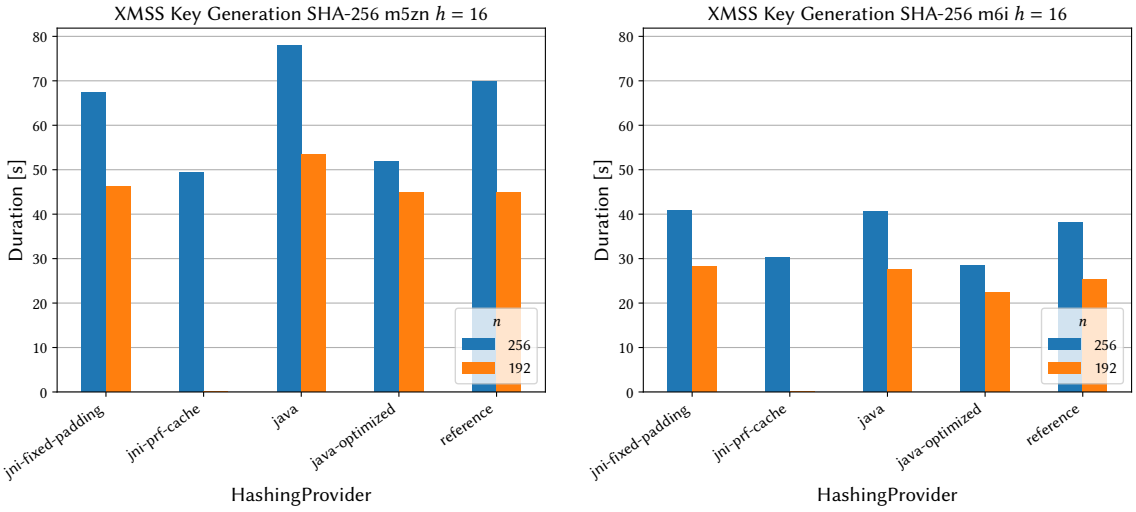
Figure 7.13.: Benchmark results for XMSS hashing with SHA-256 for `java` and `java-optimized`

SHA-256 Figure 7.13 depicts the benchmark results for hashing with the optimization levels `java` and `java-optimized`. We find that, on `m6i`, the level `java` slightly outperforms `jni-fixed-padding`. On `m5zn`, `java` is slower than `jni-fixed-padding`.

The optimization level `java-optimized` significantly reduces the time required to calculate a hash compared to `java`. For F, the only difference is the way the implementation is invoked: `java` uses the JCA/JCE interface, while `java-optimized` directly invokes the

underlying implementation. This shows that, for short-input hashing, the JCA/JCE adds a considerable cost. Due to this optimization, `java-optimized` performs roughly on par with `reference` on both platforms. Relative to BC as shown in Figures 7.10a and 7.10b, the required time for one evaluation of F is reduced by up to 54.8% on `m5zn` and 84.4% on `m6i`.

For $n = 256$, we find that the PRF caching optimization significantly reduces the cost of hashing with PRF. This is consistent with our observations for `bc-optimized` and `jni-prf-cache` shown in Figures 7.4, 7.10a and 7.10b. Overall, this brings the total improvement over `bc` for PRF to 71.5% and 89.7%, respectively.



(a) `m5zn` (b) `m6i`

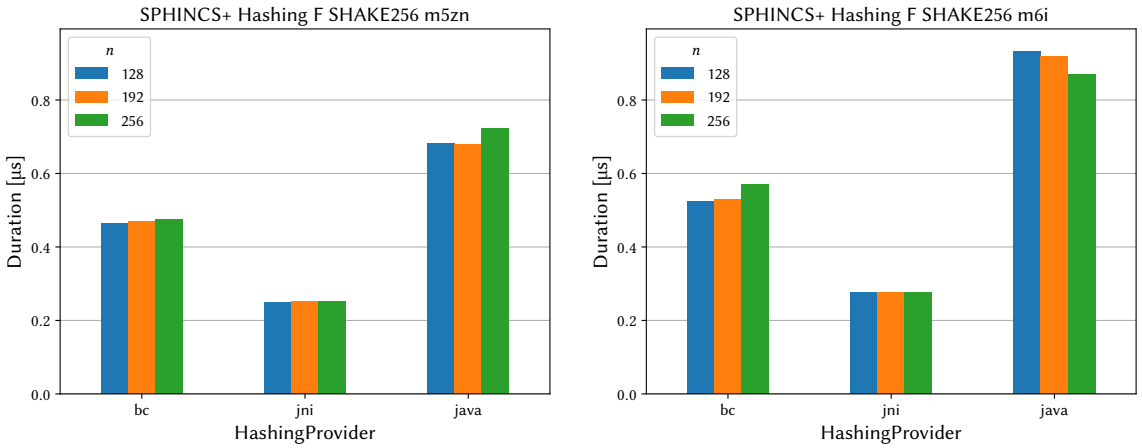
Figure 7.14.: Benchmark results for XMSS key generation with `SHA-256` for `java` and `java-optimized`

The benchmark results for the XMSS key generation are visualized in Figure 7.14. Key generation with `java` on `m6i` is comparable to `jni-fixed-padding` while it is slower on `m5zn`. One possible reason for this is that the SHA-2 implementation in OpenSSL may be better optimized than the implementation of the intrinsic. This would not affect `m6i` because the SHA-NI are used on this platform for both implementations.

We find that even for $n = 192$ without PRF caching, `java-optimized` is faster than `jni-fixed-padding` on both platforms. It performs on par with the reference implementation on `m5zn` and is better than the reference on `m6i`. Relative to `bc` as shown in Figure 7.5, the key generation cost is reduced by 52.1% on `m5zn` and 81.9% on `m6i`.

For $n = 256$, `java-optimized` considerably outperforms the reference implementation. While it is slightly faster than `jni-prf-cache` on `m6i`, it is insignificantly slower on `m5zn`. Compared to `bc` as shown in Figure 7.5 (and the reference implementation), we achieve an improvement of 64.5% (25.8%) on `m5zn` and 85.0% (25.1%) on `m6i`.

Overall, we find that the JNI and JIT intrinsics used in the `java` levels perform similarly with all possible optimizations. For the majority of parameter sets and platforms, the approach using intrinsics is faster, but only slightly.



(a) `m5zn` (b) `m6i`

Figure 7.15.: Benchmark results for SPHINCS⁺ hashing with SHAKE256 for the optimization level `java`

SHAKE256 We implement SPHINCS⁺ as the only signature scheme with SHAKE256 for the optimization level `java`. Figure 7.15, therefore, shows the hashing results for SPHINCS⁺. Unexpectedly, the optimization level `java` is up to 3.4 times slower than `jni`. Furthermore, it is up to about 80% slower than `bc`.

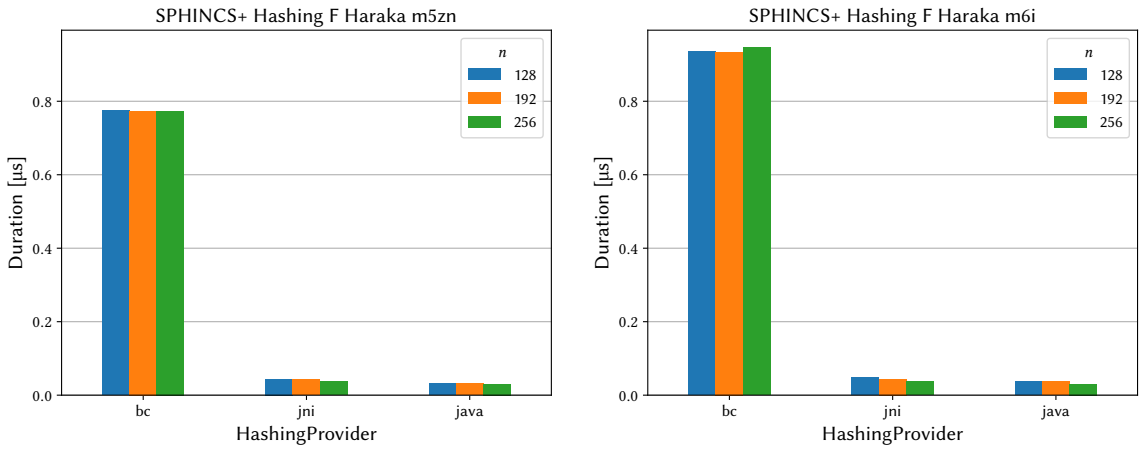
The main reason is that even though the SHA-3 state update function in the `SUN` provider is an intrinsic candidate, there is no implementation of this intrinsic for the x86 platform. In OpenJDK 18 [72], there is only an intrinsic implementation for the aarch64 platform. Therefore, we can not expect `java` to reach the level of performance of a native implementation for SHAKE256.

The reason why `java` performs worse than `bc` may be a combination of a less efficient Java implementation and the overhead introduced by the JCA/JCE interface.

Without an intrinsic, we do not consider further implementation and investigation for SHAKE256 with `java` to be worthwhile. We consider integrating a high-performance SHAKE256 implementation as an intrinsic to be too complex and, therefore, out of scope for this thesis.

Haraka Figure 7.16 shows that the Haraka intrinsic used in `java` performs even better than `jni`. Overall, this is an improvement by up to 96.8% compared to `bc`. For the SPHINCS⁺ key generation, we observe a reduction by up to 95.7%. The benchmarks for the SPHINCS⁺ operations are available in Appendix A.3.

This is mainly due to the design of Haraka. It is designed to be efficiently implementable using the AES round function and a permutation that can be implemented using a few



(a) `m5zn` (b) `m6i`

Figure 7.16.: Benchmark results for SPHINCS⁺ hashing with Haraka for the optimization level `java`

vector instructions. In Java, both the AES round function and the permutation must be manually implemented which leads to an inefficient implementation. Additionally, Haraka with AES-NI is significantly faster than the other hash functions, including SHA-2 with SHA-NI. Therefore, the relative impact of the JNI performance is larger and this leads to a larger relative improvement between `jni` and `java`.

On `m6i`, we observe that $n = 256$ is faster than smaller inputs. Our remarks for `jni` in Section 7.2.3.3 also apply here because our implementation uses one buffer for both input and output.

7.2.5. Summary

Overall, we find that techniques that use platform-specific native implementations via JNI or as an intrinsic significantly boost performance. The results are comparable to native implementations.

General Implementation Guidance Comparing JNI and intrinsics as a method to use native instructions, intrinsics generally have a lower overhead and are therefore faster. This may not be the case for SHA-256 on `m5zn` as displayed in Figure 7.13a. We expect that this is not caused by how the underlying native implementation is called, but instead by different native implementations.

Introducing new intrinsics requires changes to the Java runtime which is often not feasible. However, many large software vendors already maintain a fork of OpenJDK for their purposes. In this case, introducing new intrinsics may be viable.

An additional advantage of using intrinsics is that it is necessary to provide a working Java implementation as well. This means that the resulting code can also be run on other

architectures, even if only with reduced performance. This is generally not the case for JNI implementations if the required native binary is not available for the current platform.

On the other hand, JNI implementations are more flexible as this approach is easier to develop and maintain. Furthermore, the resulting code can be executed on any Java runtime.

Overall, we recommend considering intrinsics for the best performance. Otherwise, JNI provides a more flexible and simpler alternative at a slightly increased cost.

Optimizations for BouncyCastle We also want to discuss which of the presented optimizations can reasonably be integrated into BouncyCastle. The crypto library aims to be compatible across many Java versions. Requiring a specific JDK with custom intrinsics would therefore not be feasible. Additionally, the BouncyCastle library may be used as part of other applications which may have their own requirements to the Java runtime.

From a technical point of view, little speaks against integrating a native hash implementation over JNI. However, to the best of the author's knowledge, BouncyCastle is currently a Java-only library without native dependencies. To retain the platform independence, it would be necessary to include a Java implementation as a fallback for when the library is used on another platform for which no dedicated library was built. However, introducing native dependencies into BouncyCastle would considerably increase the complexity of the project and therefore the maintenance effort. Furthermore, BouncyCastle explicitly aims to provide Java implementations of cryptographic algorithms [68].

If integrating native hashing via JNI is not an option, the underlying hash implementation should be made configurable. In this case, a developer integrating the library into an application can provide a hash implementation suitable for the application scenario: a JNI implementation, one using the `SUN` JCA/JCE provider, or the default BouncyCastle implementation if performance is not critical.

Additionally and independent of the underlying hash implementations, the PRF caching should be integrated into BouncyCastle. The effort required is presumably quite small and yet it considerably accelerates XMSS with $n = 256$.

For LMS, a better tree traversal algorithm should be applied as BouncyCastle does not use an explicit one yet. The best choice at the moment is the BDS algorithm [10] which is already used by BouncyCastle for XMSS.

7.3. Parallelization

We run benchmarks for parallelized key generation on the EC2 instances `m5zn.2xlarge` and `m6i.2xlarge`. With SMT disabled, they provide four CPU cores. For each combination of parameter set and optimization level, the sequential and parallel key generation benchmarks are executed in direct succession to minimize the impact of possible long-term performance changes of the benchmark machines.

7.3.1. Results

Figure 7.17 shows the speedup for XMSS and LMS key generation, that is, the run time of the parallel execution divided by the sequential execution. Note that for perfectly parallelizable tasks the maximum speedup theoretically possible is the number of processors used.

The traversal of a large Merkle tree is well parallelizable as the cost of the sequential merging of the subtasks is negligible compared to the cost of a subtask as long as the number of subtasks is reasonably small. In practice, we might expect a somewhat smaller speedup value when the parallel execution used all available processors. This is due to other tasks running on the machine like background tasks of the operating systems. If the parallel benchmark uses all available processors, these tasks might interrupt the benchmark and therefore influence its performance.

Concretely, when running a parallelized Merkle tree generation with $h \geq 15$ on four processors, we expect a speedup close to 4, but not exceeding it.

Figure 7.17 shows our benchmark results. We observe speedup values in the range of 3.5 to 4.22. The majority of observed speedups is around 3.9 which matches our expectations. Yet, we observe some outliers, especially speedups exceeding 4. This represents a superlinear speedup, the sum of the computation time spent by the processors is lower for the parallel execution than for the sequential speedup. This requires further investigation.

Overall, we can reduce the required time for an XMSS key generation on `m6i` by 96.4% compared to the single-threaded `bc` implementation when using the parallelized implementation with `java-optimized` on four cores.

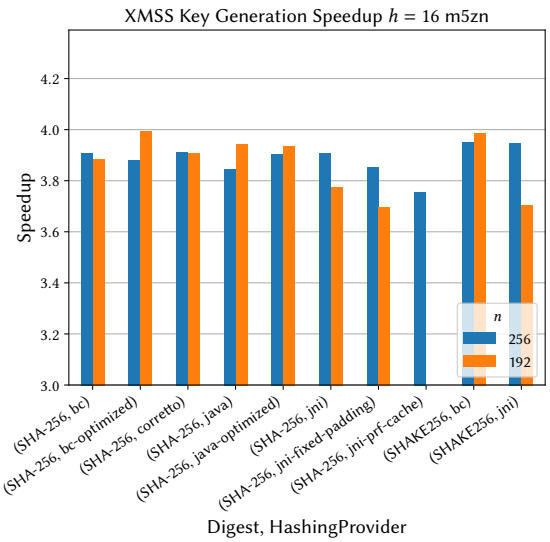
7.3.2. Superlinear Speedups

Superlinear speedups are not per se impossible. Consider a perfectly parallelizable task whose working set does not fit into the processor cache and therefore requires a large number of accesses to the memory. Splitting the task leads to smaller working sets that might fit entirely into the processor cache. In such a situation, a superlinear speedup may be achievable.

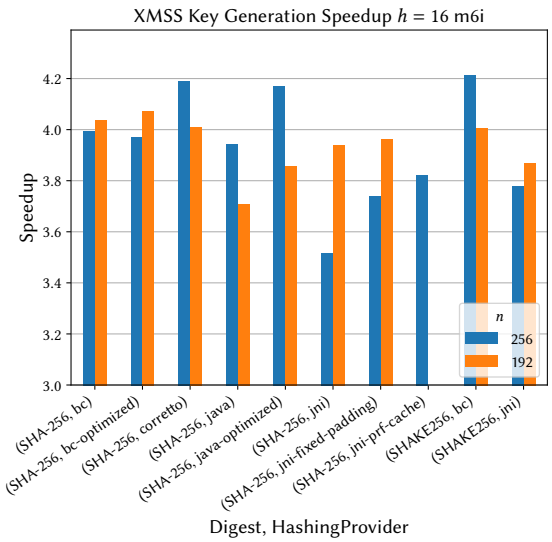
However, this is not the case for the key generation in Merkle trees. For both implementations, the number of subtasks does not significantly influence the size of the working set.

We were able to reproduce this behavior in an isolated setting independent of the HBS implementations. Listing 1 shows the code for a benchmark that computes 10,000,000 hash values with the `SHA-256` implementation provided by BouncyCastle. This is done either on only one thread or four threads in parallel. Note that the tasks on the threads are independent and do not change when the number of threads changes.

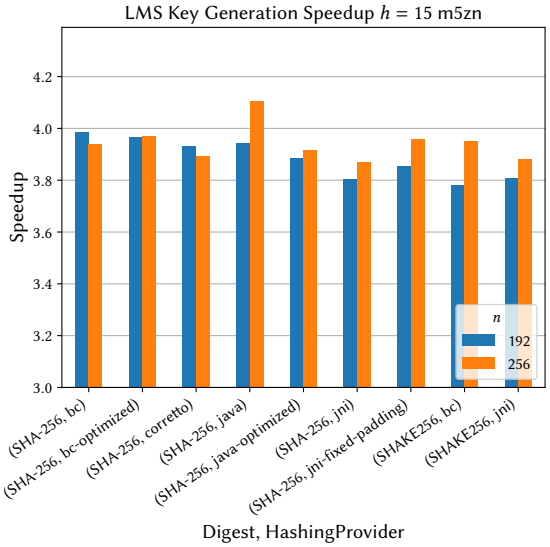
We expect that executing this code on four threads takes slightly longer than on one thread due to the overhead of creating threads and the full utilization of all CPU cores.



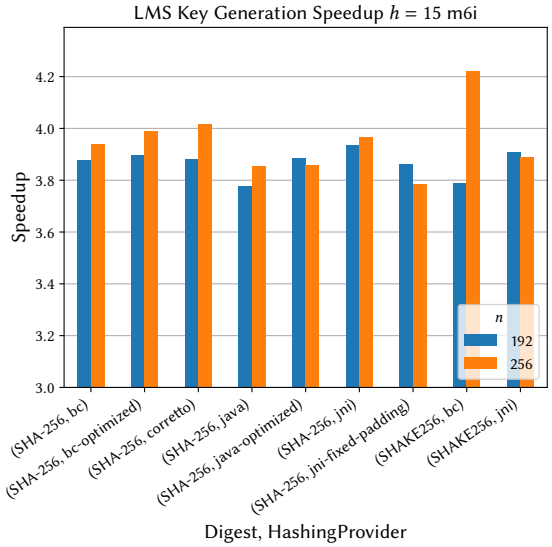
(a) XMSS on `m5zn` with $h = 16$



(b) XMSS on `m6i` with $h = 16$



(c) LMS on `m5zn` with $h = 15$



(d) LMS on `m6i` with $h = 15$

Figure 7.17.: Speedup for parallel key generation on 4 cores

```java
@Param({"1", "4"})
int threads;

@Param({"10000000"})
int iterations;

@Benchmark
public int testParallelHash() {
        return IntStream.range(0, threads)
        .parallel()
        .map((i) -> {
                int sum = 0;
                SHA256Digest digest = new SHA256Digest();
                byte[] data = new byte[64];
                byte[] out = new byte[32];
                for(int j = 0; j < iterations; j++){
                        data[2] = (byte) j;
                        digest.update(data, 0, data.length);
                        digest.doFinal(out, 0);
                        sum += out[0];
                }
                return sum;
        }).sum();
}
```

Listing 1: Benchmark for the parallel execution of BouncyCastle's `SHA256Digest`

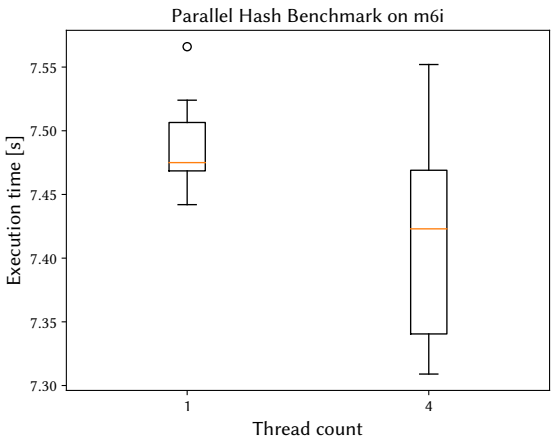


Figure 7.18.: Box plot of the individual executions of the parallel hash benchmark

This benchmark is executed on `m6i.2xlarge` which provides four CPU cores with SMT disabled. Figure 7.18 displays the distribution time of the benchmark. Each execution consists of one invocation of the benchmark method in Listing 1. Overall, we observe that the execution times of the benchmark with four threads show more variation than for only one thread. However, the majority of executions with four threads are faster than the fastest single-threaded execution. Over 15 iterations, the executions with four threads are also faster on average.

This does not align with our expectations. Yet, it is similar to our observations for parallelized HBS key generation. We investigated multiple hypotheses for the reasons for this behavior including the following:

Code Caching If the same code and constants are used by multiple threads concurrently, it may be held in higher levels of the CPU cache. If only one thread is used, the code in the cache is accessed less frequently and might be evicted more often, causing more accesses to memory or a higher cache level which cause a slower execution. This could especially affect the benchmark above because it requires the round constants for SHA-2 which are relatively large.

We were hoping to confirm this by reducing the code footprint of the task in the benchmark.

Dynamic Performance Allocation Our benchmarks run on a virtual machine on AWS. We have no influence on and little information about how the host's resources are allocated to the virtual machines. One possibility is that the hypervisor generally allocates more resources to machines that utilize all provided virtual CPUs. For example, the hypervisor could ensure that a cores' multi-threads are not used by another VM or that such a VM is interrupted less frequently leading to fewer related cache flushes.

We hoped to confirm this by only executing the hash benchmark on one thread while keeping the other cores busy with another simple task.

Unfortunately, we were unable to finally confirm or repudiate any of our hypotheses. Nevertheless, we are able to reproduce this behavior in an isolated setting and are therefore confident that this is caused by the execution environment used for the benchmarks and not by the parallelized implementation itself.

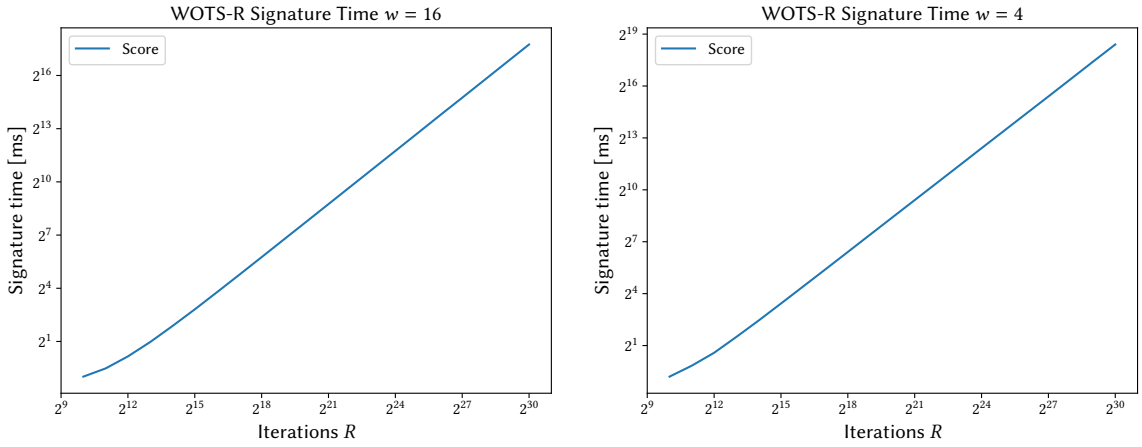
7.4. Verification-Optimized Signatures

This section presents the results of the experimental validation of the verification-optimized WOTS variants WOTS-BR and WOTS+C. We first discuss WOTS-BR, then WOTS+C and finish by comparing both schemes.

7.4.1. WOTS-BR

For this section, we only focus on the WOTS-R optimization as the addition of WOTS-B only causes an improvement by a constant offset. Section 7.4.3 compares both scheme variants.

The benchmarks for WOTS-R are executed on the `m6i` instances with the `java-optimized` optimization level. Two choices of the Winternitz parameter are evaluated, $w = 16$ (default in XMSS) and $w = 4$. We do not provide any comparison to other optimization levels and instances and focus on the impact of the parameter R on the scheme.



(a) $w = 16$ (b) $w = 4$

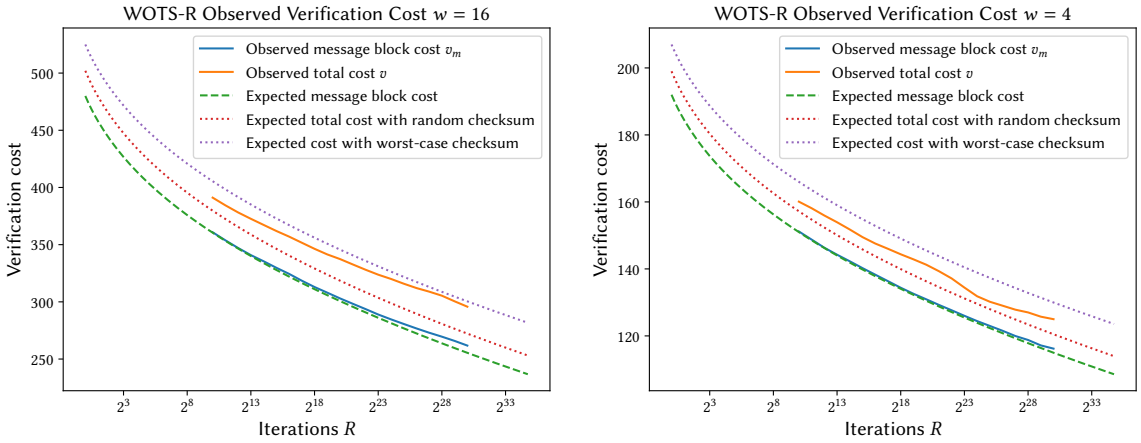
Figure 7.19.: WOTS-R signature time on `m6i` with `java-optimized`

Runtime Figure 7.19 shows that there is a clear linear dependence between the number of iterations R and the signature generation time. The curve flattens a little for small values of R. This is because the WOTS-R signing process consists of two main operations: finding the best nonce and generating the WOTS signature. While the cost of the first is linear in R, the cost of the latter is constant.

Verification Cost This section discusses the observed verification costs for the generated signatures and compares them to the approximations of the expected value presented in Section 6.4.1.1.

Figure 7.20 shows our experimentally achieved verification costs compared to the expected values presented in the theoretical evaluation. We recall that the observed value is always the average over a number of messages (usually 1024).

For both choices of w, the observed verification cost for the message blocks v_m is very close to the expected value. The total verification cost v lies between the expected value assuming random checksums and the expected value assuming the worst-case checksum. For $w = 16$, the observed values start roughly in the middle between both for $R = 2^{10}$ and



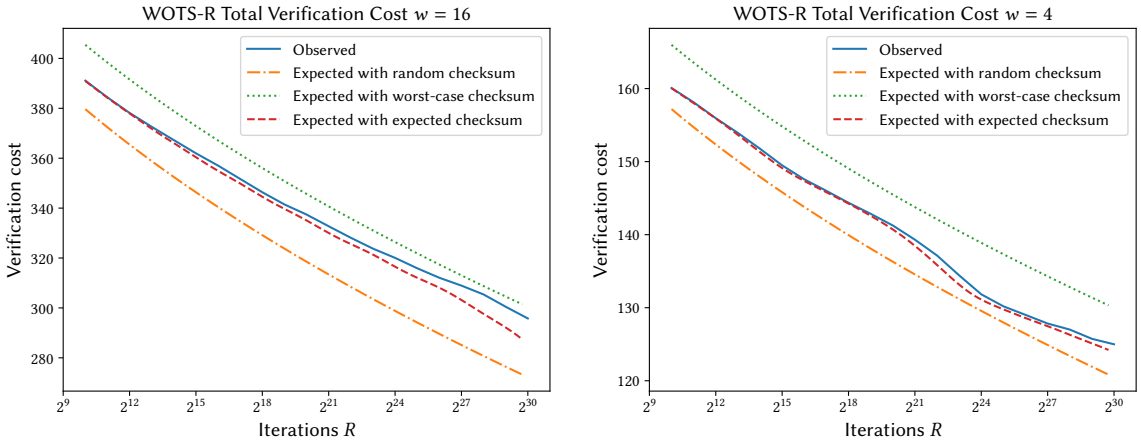
(a) $w = 16$ (b) $w = 4$

Figure 7.20.: WOTS-R observed verification cost

then converge to the expected value for the worst-case checksum. However, a slight bend is recognizable at about $R = 2^{28}$.

For $w = 4$, the observed total verification cost behaves differently. It starts roughly in the middle, approaches the expected value for a random checksum at around $R = 2^{24}$, and returns to the middle.

We note that our experimental results closely resemble those presented by Bos et al. [8]. All features described here are recognizable in their results as well.



(a) $w = 16$ (b) $w = 4$

Figure 7.21.: WOTS-R total verification cost

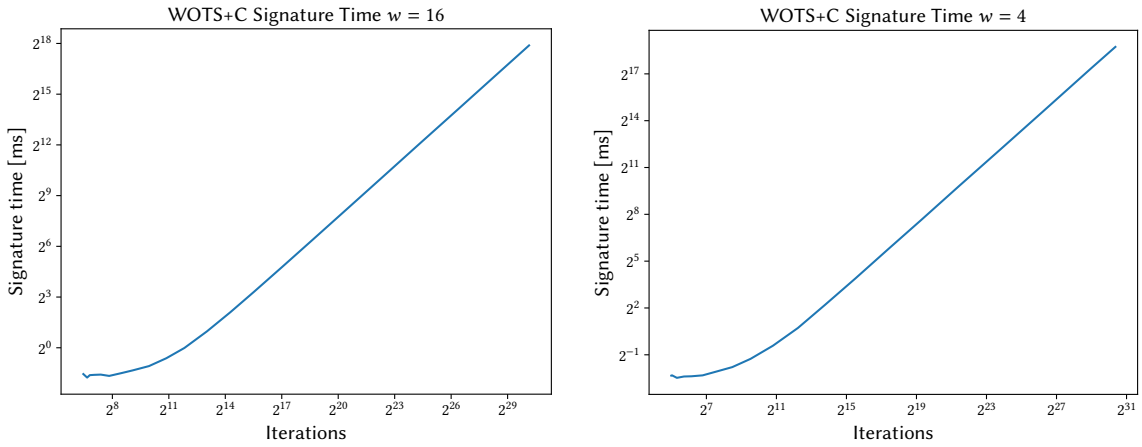
Figure 7.21 compares the observed total verification cost v to the approximations presented in Section 6.4.1.1. We observe for $w = 4$, the expected value with the expected checksum very closely approximates the observed behavior. The same holds for $w = 16$, but only for

small values of R. For $R \geq 2^{26}$, the approximation significantly diverges from the observed results.

One reason for this may be that for large R, we only evaluate a smaller number of messages. Therefore, the observed signature costs may be less representative. However, the experimental results presented by Bos et al. [8] are very similar to ours for large values of R.

7.4.2. WOTS+C

As for WOTS-R, we run all benchmarks on `m6i` instances with the optimization level `java-optimized`. We set $z = 0$ and evaluate both $w = 16$ and $w = 4$.



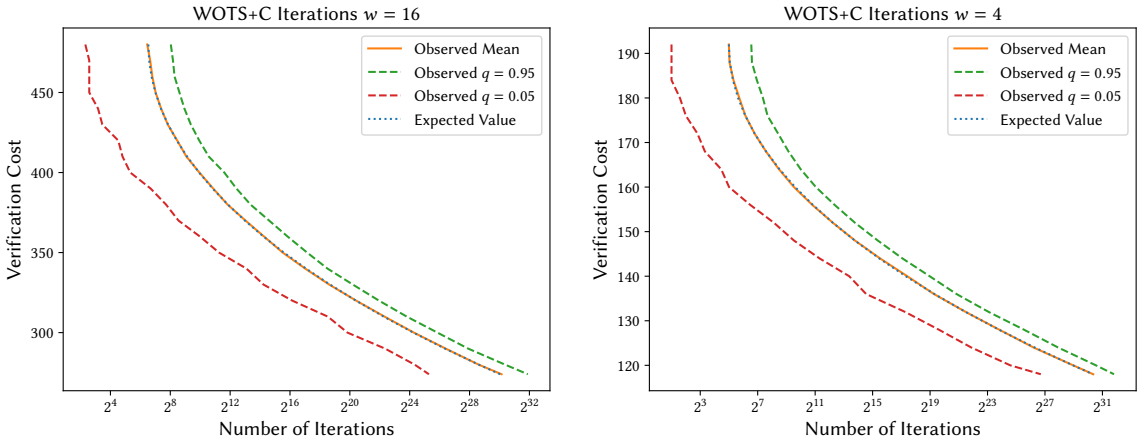
(a) $w = 16$ (b) $w = 4$

Figure 7.22.: WOTS+C signature time on `m6i` with `java-optimized`

Runtime Figure 7.22 shows the total signature runtime by the number of iterations made. For larger iteration counts, we observe a clear linear dependency as one would expect. Like with WOTS-R, we see the impact of the WOTS signature after a suitable nonce has been found for small iteration counts. This effect looks more distinct in Figure 7.22 than in the benchmark results for WOTS-R in Figure 7.19. This is, however, only because the benchmark results for WOTS+C include measurements for smaller iteration counts while we only evaluated $R \geq 2^{10}$ for WOTS-R.

Verification Cost Figure 7.23 shows the observed number of iterations on average (and its quantiles) for a given verification cost which is configured by the parameter S. Additionally, it contains the expected behavior determined in the theoretical evaluation.

For both choices of w, the experimental results match the expected value remarkably closely. This validates the theoretical evaluation.



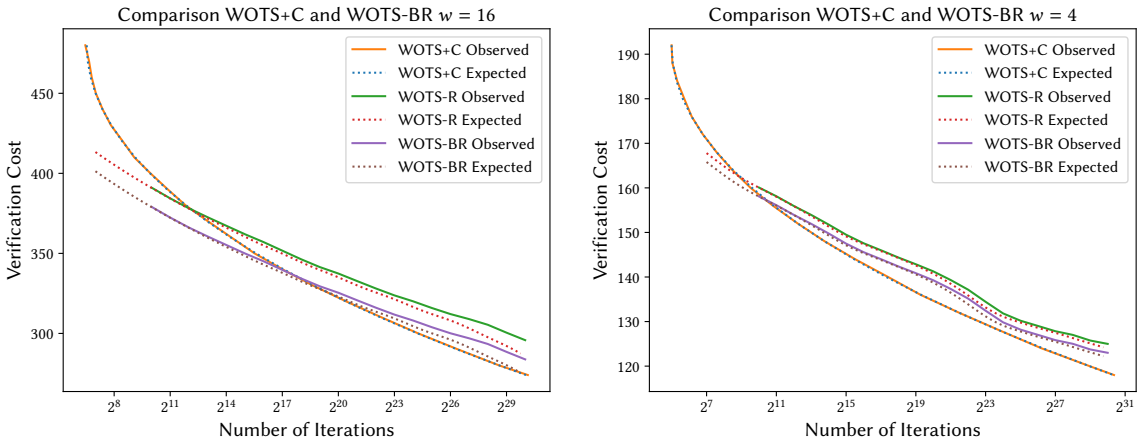
(a) $w = 16$ (b) $w = 4$

Figure 7.23.: WOTS+C observed iterations for given verification costs

7.4.3. Comparison

Iterations To confirm that the cost of iterations of WOTS+C and WOTS-R are indeed comparable, we chose a message and set $w = 16$ and $S = 650$. For these parameters, we benchmark WOTS+C and find that the signing process takes 615 ms and requires 2,986,488 iterations. Hence, we set R to this value and benchmark WOTS-R. Here, the signing process takes 613 ms. We consider the difference between both measurements to be insignificant.

Due to the selection of parameters, both schemes make the same number of iterations and require the same amount of time for a signature. Therefore, we argue that the amount of time required for one iteration has to be equal for both schemes.



(a) $w = 16$ (b) $w = 4$

Figure 7.24.: Comparison of WOTS+C, WOTS-R and WOTS-BR

Experimental comparison In Figure 7.24, we compare the experimental results for the variants WOTS+C, WOTS-B, and WOTS-BR. As we note in the previous sections, the results for WOTS+C closely match the expected values, while the results for WOTS-R and WOTS-BR tend to be worse than the expected value. This is especially the case for $w = 16$ with large R.

Overall, this confirms our remarks in the theoretical comparison in Section 6.4.1.3. For $w = 4$ and a reasonably large number of iterations, the verification cost for WOTS+C is on average smaller than WOTS-R and WOTS-BR while also reducing the size of the signature. For $w = 16$, we observe that the observed results for WOTS-R and WOTS-BR are worse than the expected value, especially so for larger iteration counts. Therefore, we recommend the use of WOTS+C also for this choice of parameters. For iteration counts significantly larger than 2^{30}, a further investigation is required as we expect WOTS-BR to perform better (see Section 6.4.1.3). However, we consider such iteration counts to be infeasible for most application scenarios.

Hence, we find that for common parameter choices, WOTS+C performs better than or on par with WOTS-BR. Therefore, we recommend the use of WOTS+C in these scenarios.

8. Conclusion

This chapter gives a short summary of the results of this thesis and provides an outlook on possible future work.

8.1. Summary

Implementation Optimizations We find that using native hash implementations via JNI or as an intrinsic significantly improves the performance of HBS implementations in Java. Intrinsics are generally the more efficient way to integrate native implementations but JNI with the proper data transfer technique is usually only marginally slower and more flexible.

Using the PRF caching optimization, we achieve a faster XMSS key generation with our Java implementation than with the XMSS reference implementation. Overall, we achieve an improvement of up to 85.0% for SHA-2 with SHA-NI. For Haraka in SPHINCS$^+$, the acceleration achieved through the use of AES-NI is even larger. We observe a reduction of the SPHINCS$^+$ key generation time by up to 95.7%.

Additionally, we observe that parallelization also improves the performance of HBS operations significantly. The key generation for a Merkle tree is very well parallelizable. We confirm this experimentally. However, our experimental results prove to be inconsistent. Still, we are confident that this is not due to the parallelized task or our implementation, but an artifact of the benchmark environment.

Verification-Optimized Signatures We improve the models for WOTS+C and WOTS-BR presented in the literature and provide a theoretical comparison. It shows that, for small Winternitz parameters w and a reasonable number of iterations, we can expect WOTS+C to yield signatures with a lower verification cost than WOTS-BR while also reducing the signature size.

We experimentally validate the models and note that the observed behavior of WOTS+C closely matches the expected behavior, while WOTS-BR performs worse for some parameters in our experiments than in the theoretical analysis. Therefore, we recommend WOTS+C even for larger Winternitz parameters. However, this depends on the concrete choice of parameters and requires careful consideration when choosing a scheme for non-standard parameters.

8.2. Future Work

This section presents additional optimization strategies for HBS that were not covered in this thesis but could be investigated in future studies.

Multi-Message Hashing Several papers presented in Chapter 4 use multi-message hashing to accelerate hashing or specifically HBS operations. We expect that the HBS implementations could be further improved by using a multi-message hashing implementation that utilizes some form of hardware-level parallelism. Such implementations could then be invoked from Java via JNI or as an intrinsic.

Java Vector API The Java Vector API [65] allows Java code to utilize the vector capabilities of the CPU in a platform-independent way. At the time of writing, it is still in the incubator and not finally released. This API could be used to implement multi-message hashing or vectorized message scheduling in SHA-2 using Java code only. We do not expect that such an implementation will perform better than a native implementation using SHA-NI or AES-NI. However, a vectorized hash implementation in Java could perform better than current Java-only implementations. The benefit of this approach is that the resulting implementation does not rely on native libraries or a modified JDK and is therefore fully platform-independent.

Java Foreign Function & Memory API The upcoming Java Foreign Function & Memory API [13] provides an alternative way to invoke native functions from Java and to access native memory from Java. Similar to the Vector API, this API was not finally released at the time of writing. Future work could investigate and evaluate integrating native hash implementations via this API instead of JNI. The new API explicitly aims to deliver performance comparable to or better than JNI. Additionally, passing data to native methods is handled differently than using JNI. Overall, the use of this new API could provide a more performant way to integrate native hash implementations.

Parallelized Signing and Verification A parallelized HBS implementation should also make use of parallelized signing and verification operations. Sun et al. [67] describe how WOTS signatures can be computed in parallel. Further research could also evaluate the parallelization of the BDS algorithm for Merkle tree traversal.

Encodings for Verification-Optimized Signatures In addition to our comparison of WOTS-BR and WOTS+C, further work may investigate the use of different encoding functions for verification-optimized signatures. The most prominent examples are constant-sum encodings [16, 37, 56] and encodings using non-adjacent forms [64, 81]. These proposals generally have many parameters. Future research could determine if and how they can be tuned for verification-optimized signatures.

Bibliography

[1] Amazon Web Services, Inc. *Amazon EC2 Instance Types*. [Online; accessed 2023-02-15]. Feb. 2023. URL: https://aws.amazon.com/ec2/instance-types/.

[2] Amazon.com Inc. *Amazon Corretto Crypto Provider*. GitHub Repository. Feb. 2023. URL: https://github.com/corretto/amazon-corretto-crypto-provider.

[3] Jean-Philippe Aumasson et al. *SPHINCS$^+$ - Submssion to the NIST post-quantum project, v3.1*. June 2022. URL: https://sphincs.org/data/sphincs+-r3.1-specification.pdf.

[4] Daniel J. Bernstein et al. "SPHINCS: practical stateless hash-based signatures". In: *IACR Cryptol. ePrint Arch.* (2014), p. 795. URL: http://eprint.iacr.org/2014/795.

[5] Daniel J. Bernstein et al. "The SPHINCS$^+$ Signature Framework". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro et al. ACM, 2019, pp. 2129–2146. DOI: 10.1145/3319535.3363229.

[6] Guido Bertoni et al. *The Keccak Reference*. Jan. 2011. URL: https://keccak.team/files/Keccak-reference-3.0.pdf.

[7] Dan Boneh and Victor Shoup. *A graduate course in applied cryptography*. Jan. 2020. URL: https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_5.pdf.

[8] Joppe W. Bos et al. "Rapidly Verifiable XMSS Signatures". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.1 (2021), pp. 137–168. DOI: 10.46586/tches.v2021.i1.137-168.

[9] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions". In: *Post-Quantum Cryptography*. Ed. by Bo-Yin Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 117–129. DOI: 10.1007/978-3-642-25405-5_8.

[10] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. "Hash-based Digital Signature Schemes". In: *Post-Quantum Cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 35–93. DOI: 10.1007/978-3-540-88702-7_3.

[11] Johannes Buchmann et al. "On the Security of the Winternitz One-Time Signature Scheme". In: *Progress in Cryptology – AFRICACRYPT 2011*. Ed. by Abderrahmane Nitaj and David Pointcheval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 363–378. DOI: 10.1007/978-3-642-21969-6_23.

[12] Jason Chia, Ji-Jian Chin, and Sook-Chin Yip. "Digital signature schemes with strong existential unforgeability". In: *F1000Research* 10 (Sept. 2021), p. 931. DOI: 10.12688/f1000research.72910.1.

[13] Maurizio Cimadamore. *JEP 434: Foreign Function & Memory API (Second Preview)*. [Online; accessed 2023-03-21]. Mar. 2023. URL: https://openjdk.org/jeps/434.

[14] David Cooper et al. *Recommendation for Stateful Hash-Based Signature Schemes*. en. Oct. 2020. DOI: 10.6028/NIST.SP.800-208.

[15] Carlos Coronado. "On the security and the efficiency of the Merkle signature scheme". In: *IACR Cryptol. ePrint Arch.* (2005), p. 192. URL: http://eprint.iacr.org/2005/192.

[16] Jason Paul Cruz, Yoshio Yatani, and Yuichi Kaji. "Constant-sum fingerprinting for Winternitz one-time signature". In: *2016 International Symposium on Information Theory and Its Applications, ISITA 2016, Monterey, CA, USA, October 30 - November 2, 2016*. IEEE, 2016, pp. 703–707. URL: https://ieeexplore.ieee.org/document/7840516/.

[17] Quynh Dang. *Secure Hash Standard*. en. Aug. 2015. DOI: 10.6028/NIST.FIPS.180-4.

[18] Quynh Dang. *The Keyed-Hash Message Authentication Code (HMAC)*. en. July 2008. DOI: 10.6028/NIST.FIPS.198-1.

[19] Morris Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. en. Aug. 2015. DOI: 10.6028/NIST.FIPS.202.

[20] Morris Dworkin et al. *Advanced Encryption Standard (AES)*. en. Nov. 2001. DOI: 10.6028/NIST.FIPS.197. URL: https://csrc.nist.gov/publications/detail/fips/197/final.

[21] Armando Faz-Hernández, Weikeng Chen, and Ana Karina D. S. de Oliveira. *FLO-SHANI-AESNI*. GitHub Repository. Apr. 2021. URL: https://github.com/armfazh/FLO-shani-aesni.

[22] Armando Faz-Hernández, Julio César López-Hernández, and Ana Karina D. S. de Oliveira. "SoK: A Performance Evaluation of Cryptographic Instruction Sets on Modern Architectures". In: *Proceedings of the 5th ACM on ASIA Public-Key Cryptography Workshop, APKC@AsiaCCS, Incheon, Republic of Korea, June 4, 2018*. Ed. by Keita Emura, Jae Hong Seo, and Yohei Watanabe. ACM, 2018, pp. 9–18. DOI: 10.1145/3197507.3197511.

[23] Scott R. Fluhrer. "Further Analysis of a Proposed Hash-Based Signature Standard". In: *IACR Cryptol. ePrint Arch.* (2017), p. 553. URL: http://eprint.iacr.org/2017/553.

[24] Brian Goetz. *JEP 348: Compiler Intrinsics for Java SE APIs*. [Online; accessed 2023-02-15]. Feb. 2023. URL: https://openjdk.org/jeps/348.

[25] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511721656. URL: https://www.wisdom.weizmann.ac.il/~oded/foc-vol2.html.

[26] Vinodh Gopal et al. *Processing Multiple Buffers in Parallel to Increase Performance on Intel® Architecture Processors*. Tech. rep. Intel Corporation, July 2010. URL: https://web.archive.org/web/20220901053149/https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-multi-buffer-paper.pdf.

[27] Shay Gueron. "Speeding Up SHA-1, SHA-256 and SHA-512 on the 2nd Generation Intel® Core™ Processors". In: *Ninth International Conference on Information Technology: New Generations, ITNG 2012, Las Vegas, Nevada, USA, 16-18 April, 2012*. Ed. by Shahram Latifi. IEEE Computer Society, 2012, pp. 824–826. DOI: 10.1109/ITNG.2012.62.

[28] Shay Gueron and Vlad Krasnov. "Parallelizing message schedules to accelerate the computations of hash functions". In: *J. Cryptogr. Eng.* 2.4 (2012), pp. 241–253. DOI: 10.1007/s13389-012-0037-z.

[29] Shay Gueron and Vlad Krasnov. "Simultaneous Hashing of Multiple Messages". In: *J. Information Security* 3.4 (2012), pp. 319–325. DOI: 10.4236/jis.2012.34039.

[30] Jim Guilford, Kirk Yap, and Vinodh Gopal. *Fast SHA-256 Implementations on Intel ® Architecture Processors*. Tech. rep. Intel Corporation, May 2012. URL: https://web.archive.org/web/20220901094842/https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf.

[31] Sean Gulley et al. *Intel SHA Extensions: New Instructions Supporting the the Secure Hash Algorithm on Intel Architecture Processors*. Tech. rep. Intel Corporation, July 2013. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sha-extensions-white-paper.pdf.

[32] Thomas Hanson et al. "Optimization for SPHINCS+ using Intel Secure Hash Algorithm Extensions". In: *IACR Cryptol. ePrint Arch.* (2022), p. 1726. URL: https://eprint.iacr.org/2022/1726.

[33] Lena Heimberger. *JavaSphincsPlus*. GitLab Repository. Feb. 2021. URL: https://extgit.iaik.tugraz.at/krypto/javasphincsplus.

[34] Andreas Hülsing. "W-OTS+ - Shorter Signatures for Hash-Based Signature Schemes". In: *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*. Ed. by Amr M. Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien. Vol. 7918. Lecture Notes in Computer Science. Springer, 2013, pp. 173–188. DOI: 10.1007/978-3-642-38553-7_10.

[35] Andreas Hülsing, Joost Rijneveld, and Fang Song. "Mitigating Multi-target Attacks in Hash-Based Signatures". In: *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*. Ed. by Chen-Mou Cheng et al. Vol. 9614. Lecture Notes in Computer Science. Springer, 2016, pp. 387–416. DOI: 10.1007/978-3-662-49384-7_15.

[36] Andreas Hülsing et al. *XMSS: eXtended Merkle Signature Scheme*. RFC 8391. May 2018. DOI: 10.17487/RFC8391. URL: https://www.rfc-editor.org/info/rfc8391.

[37] Yuichi Kaji, Jason Paul Cruz, and Yoshio Yatani. "Hash-Based Signature with Constant-Sum Fingerprinting and Partial Construction of Hash Chains". In: *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications, ICETE 2018 - Volume 2: SECRYPT, Porto, Portugal, July 26-28, 2018*. Ed. by Pierangela Samarati and Mohammad S. Obaidat. SciTePress, 2018, pp. 463–470. DOI: 10.5220/0006828204630470.

[38] Burt Kaliski and Jessica Staddon. *PKCS #1: RSA Cryptography Specifications Version 2.0*. RFC 2437. Oct. 1998. DOI: 10.17487/RFC2437. URL: https://www.rfc-editor.org/info/rfc2437.

[39] Jonathan Katz. "Analysis of a Proposed Hash-Based Signature Standard". In: *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016, Proceedings*. Ed. by Lidong Chen, David A. McGrew, and Chris J. Mitchell. Vol. 10074. Lecture Notes in Computer Science. Springer, 2016, pp. 261–273. DOI: 10.1007/978-3-319-49100-4_12.

[40] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. DOI: 10.1201/b17668.

[41] Kockmeyer. *A schematic that shows the SHA-2 algorithm*. [Online; accessed 2023-03-09]. Mar. 2007. URL: https://commons.wikimedia.org/wiki/File:SHA-2.svg.

[42] Stefan Kölbl. *Haraka v2*. GitHub Repository. Sept. 2017. URL: https://github.com/kste/haraka.

[43] Stefan Kölbl. *Putting Wings on SPHINCS*. GitHub Repository. Nov. 2017. URL: https://github.com/kste/sphincs.

[44] Stefan Kölbl. "Putting Wings on SPHINCS". In: *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*. Ed. by Tanja Lange and Rainer Steinwandt. Vol. 10786. Lecture Notes in Computer Science. Springer, 2018, pp. 205–226. DOI: 10.1007/978-3-319-79063-3_10.

[45] Stefan Kölbl et al. "Haraka v2 - Efficient Short-Input Hashing for Post-Quantum Applications". In: *IACR Trans. Symmetric Cryptol.* 2016.2 (Feb. 2017), pp. 1–29. DOI: 10.13154/tosc.v2016.i2.1-29.

[46] Mikhail A. Kudinov et al. "SPHINCS+C: Compressing SPHINCS+ With (Almost) No Cost". In: *IACR Cryptol. ePrint Arch.* (2022), p. 778. URL: https://eprint.iacr.org/2022/778.

[47] Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. Tech. rep. CSL-98. This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. Oct. 1979. URL: https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/.

[48] Frank T. Leighton and Silvio Micali. *Large provably fast and secure digital signature schemes based on secure hash functions*. U.S. Patent 5,432,852. US Patent 5,432,852. July 1995. URL: https://image-ppubs.uspto.gov/dirsearch-public/print/downloadPdf/5432852.

[49] David McGrew, Michael Curcio, and Scott Fluhrer. *Leighton-Micali Hash-Based Signatures*. RFC 8554. Apr. 2019. DOI: 10.17487/RFC8554. URL: https://www.rfc-editor.org/info/rfc8554.

[50] Ralph C. Merkle. "A Certified Digital Signature". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by Gilles Brassard. New York, NY: Springer New York, 1990, pp. 218–238. DOI: 10.1007/0-387-34805-0_21.

[51] Dustin Moody. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. 2022. DOI: 10.6028/nist.ir.8413.

[52] Chris Newland. *VM Intrinsics Explorer - HotSpot Intrinsics for OpenJDK18*. [Online; accessed 2023-02-15]. Feb. 2023. URL: https://chriswhocodes.com/hotspot_intrinsics_openjdk18.html.

[53] Scott Oaks. *Java Performance - The Definitive Guide: Getting the Most Out of Your Code*. O'Reilly, 2014. URL: http://shop.oreilly.com/product/0636920028499.do.

[54] Oracle. *Java Native Interface Specification*. [Online; accessed 2023-02-15]. 2022. URL: https://docs.oracle.com/en/java/javase/18/docs/specs/jni/.

[55] Lucas Pandolfo Perin. "Message encoding algorithms for Winternitz signatures". PhD thesis. Universidade Federal de Santa Catarina, 2021. URL: https://repositorio.ufsc.br/handle/123456789/231005.

[56] Lucas Pandolfo Perin et al. "Improved constant-sum encodings for hash-based signatures". In: *J. Cryptogr. Eng.* 11.4 (2021), pp. 329–351. DOI: 10.1007/s13389-021-00264-9.

[57] Lucas Pandolfo Perin et al. "Tuning the Winternitz hash-based digital signature scheme". In: *2018 IEEE Symposium on Computers and Communications, ISCC 2018, Natal, Brazil, June 25-28, 2018*. IEEE, 2018, pp. 537–542. DOI: 10.1109/ISCC.2018.8538642.

[58] Andy Polyakov. *CRYPTOGAMS*. GitHub Repository. Feb. 2023. URL: https://github.com/dot-asm/cryptogams.

[59] Tim Rausch. *Optimizing Hash-Based Signatures in Java*. GitHub Repository. Mar. 2023. URL: https://github.com/SAP-samples/optimizing-hash-based-signatures-java.

[60] Leonid Reyzin and Natan Reyzin. "Better than BiBa: Short One-time Signatures with Fast Signing and Verifying". In: *IACR Cryptol. ePrint Arch.* (2002), p. 14. URL: http://eprint.iacr.org/2002/014.

[61] Joost Rijneveld. *SPHINCS-256*. GitHub Repository. May 2018. URL: https://github.com/sphincs/sphincs-256.

[62] Joost Rijneveld et al. *SPHINCS+*. GitHub Repository. Jan. 2023. URL: https://github.com/sphincs/sphincsplus.

[63] Joost Rijneveld et al. *XMSS reference code*. GitHub Repository. Mar. 2021. URL: https://github.com/XMSS/xmss-reference.

[64] Dongyoung Roh, Sangim Jung, and Daesung Kwon. "Winternitz Signature Scheme Using Nonadjacent Forms". In: *Secur. Commun. Networks* 2018 (2018), 1452457:1–1452457:12. DOI: 10.1155/2018/1452457.

[65] Paul Sandoz. *JEP 426: Vector API (Fourth Incubator)*. [Online; accessed 2023-02-15]. July 2022. URL: https://openjdk.org/jeps/426.

[66] Rainer Steinwandt and Viktória I. Villányi. "A one-time signature using run-length encoding". In: *Inf. Process. Lett.* 108.4 (2008), pp. 179–185. DOI: 10.1016/j.ipl.2008.05.004.

[67] Shuzhou Sun, Rui Zhang, and Hui Ma. "Efficient Parallelism of Post-Quantum Signature Scheme SPHINCS". In: *IEEE Trans. Parallel Distributed Syst.* 31.11 (2020), pp. 2542–2555. DOI: 10.1109/TPDS.2020.2995562.

[68] The Legion of the Bouncy Castle. *The Bouncy Castle Crypto Package For Java*. GitHub Repository. Feb. 2023. URL: https://github.com/bcgit/bc-java.

[69] The Legion of the Bouncy Castle. *The Legion of the Bouncy Castle*. [Online; accessed 2023-02-14]. URL: https://www.bouncycastle.org/.

[70] The Netty Project. *Netty Project*. GitHub Repository. Feb. 2023. URL: https://github.com/netty/netty.

[71] The OpenJDK Project. *Java Microbenchmark Harness*. GitHub Repository. Jan. 2023. URL: https://github.com/openjdk/jmh.

[72] The OpenJDK Project. *JDK*. GitHub Repository. Feb. 2023. URL: https://github.com/openjdk/jdk.

[73] The OpenSSL Project, Eric A. Young, and Tim J. Hudson. *OpenSSL*. GitHub Repository. Feb. 2023. URL: https://github.com/openssl/openssl.

[74] Wen Wang et al. "XMSS and Embedded Systems". In: *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*. Ed. by Kenneth G. Paterson and Douglas Stebila. Vol. 11959. Lecture Notes in Computer Science. Springer, 2019, pp. 523–550. DOI: 10.1007/978-3-030-38471-5_21.

[75] Wen Wang et al. *XMSS and Embedded Systems - XMSS Hardware Accelerators for RISC-V*. Code Archive. May 2019. URL: https://caslab.csl.yale.edu/code/xmsshwswriscv/.

[76] Wikipedia contributors. *Ice Lake (microprocessor) — Wikipedia, The Free Encyclopedia*. [Online; accessed 2023-02-14]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Ice_Lake_(microprocessor)&oldid=1132219048.

[77] Wikipedia contributors. *Intel SHA extensions — Wikipedia, The Free Encyclopedia*. [Online; accessed 2023-02-14]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Intel_SHA_extensions&oldid=1123723151.

[78] Wikipedia contributors. *Rocket Lake — Wikipedia, The Free Encyclopedia*. [Online; accessed 2023-02-14]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Rocket_Lake&oldid=1139076937.

[79] XKCP Contributors. *eXtended Keccak Code Package*. GitHub Repository. Dec. 2022. URL: https://github.com/XKCP/XKCP.

[80] Kaiyi Zhang, Hongrui Cui, and Yu Yu. "SPHINCS-α: A Compact Stateless Hash-Based Signature Scheme". In: *IACR Cryptol. ePrint Arch.* (2022), p. 59. URL: https://eprint.iacr.org/2022/059.

[81] Amos Zheng and Marcos A. Simplicio Jr. "z-OTS: a one-time hash-based digital signaturescheme with fast verification". In: *IACR Cryptol. ePrint Arch.* (2021), p. 1506. URL: https://eprint.iacr.org/2021/1506.

A. Benchmark Results

A.1. XMSS

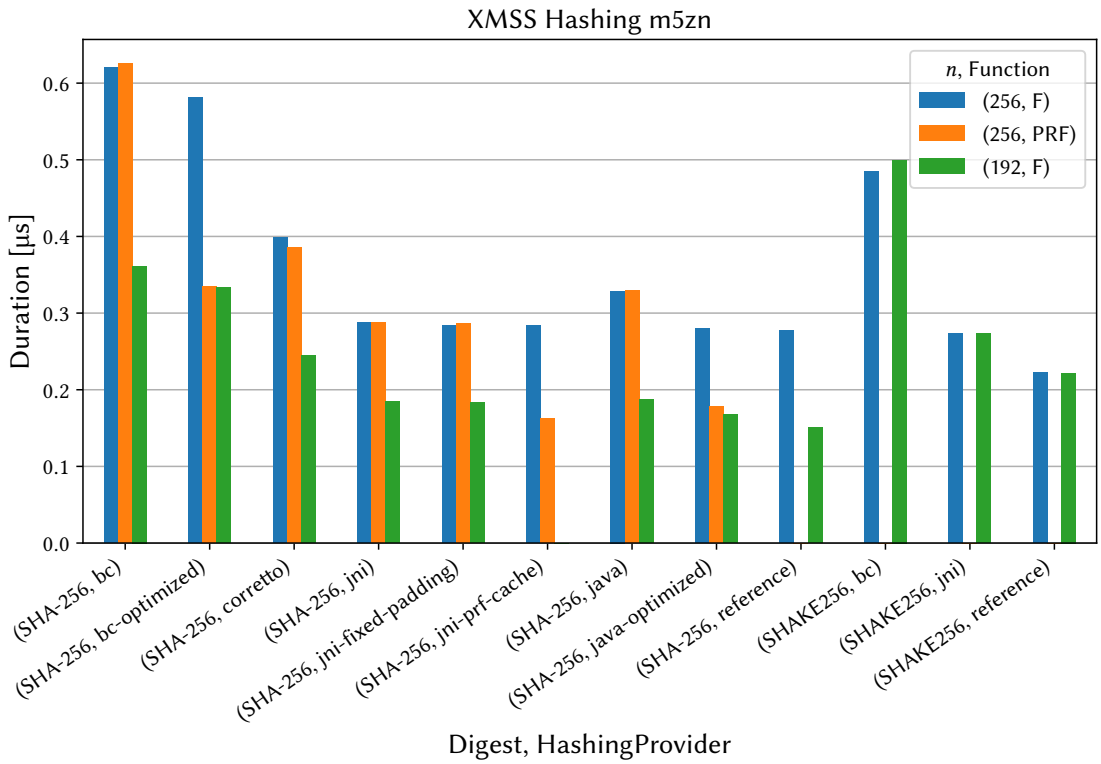


Figure A.1.: XMSS hashing on `m5zn`

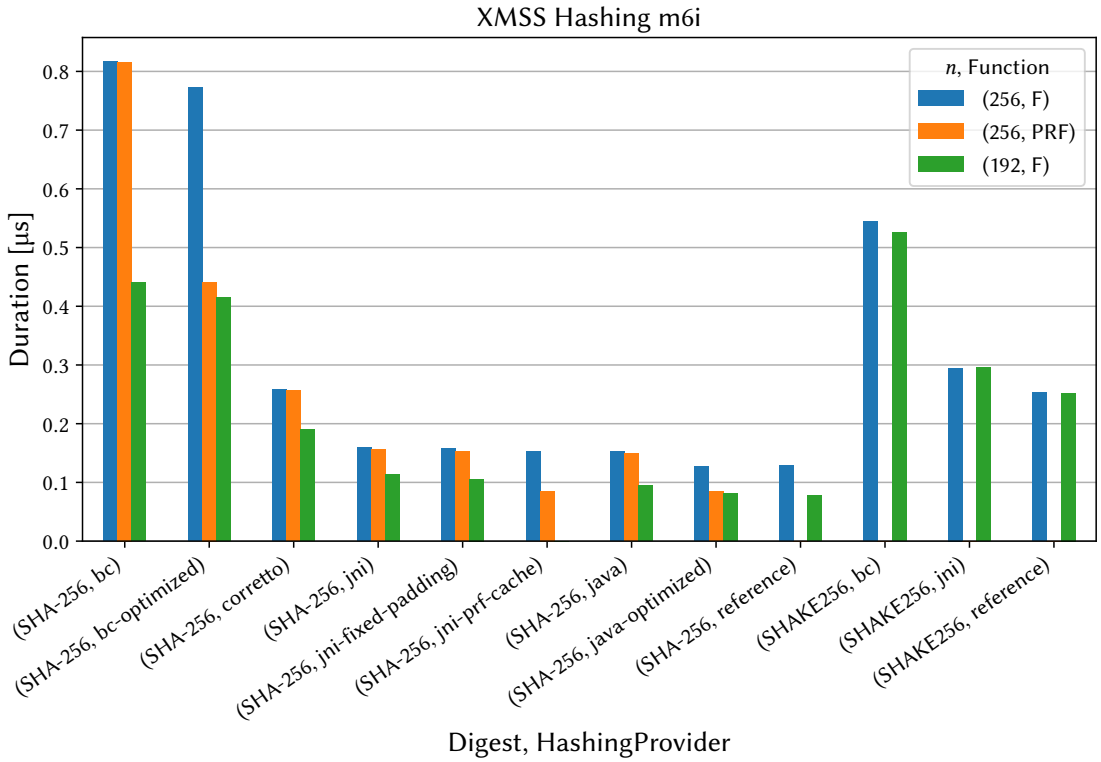


Figure A.2.: XMSS hashing on `m6i`

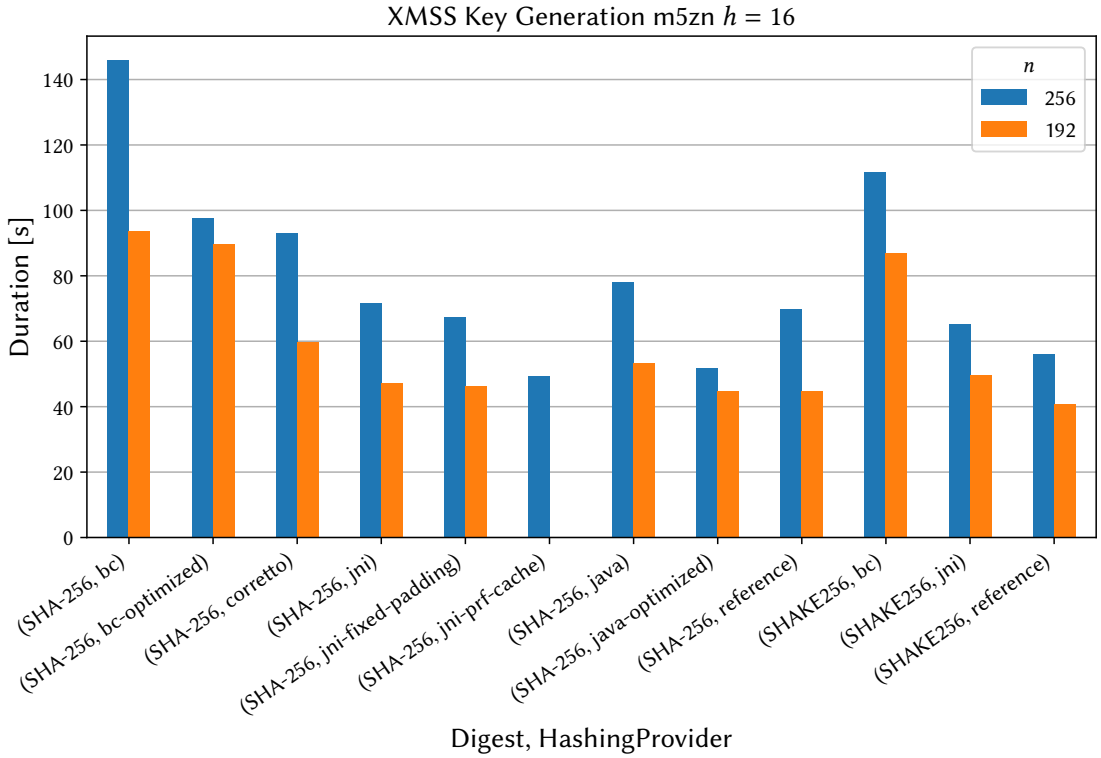


Figure A.3.: XMSS key generation on `m5zn`

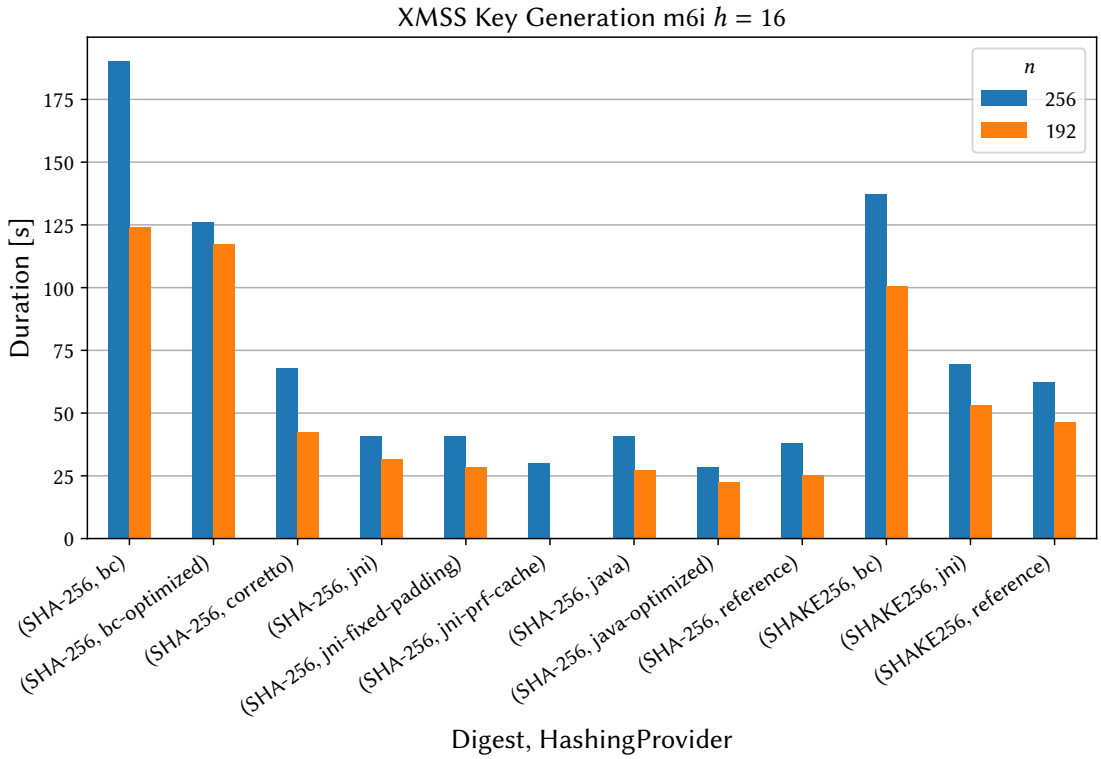


Figure A.4.: XMSS key generation on `m6i`

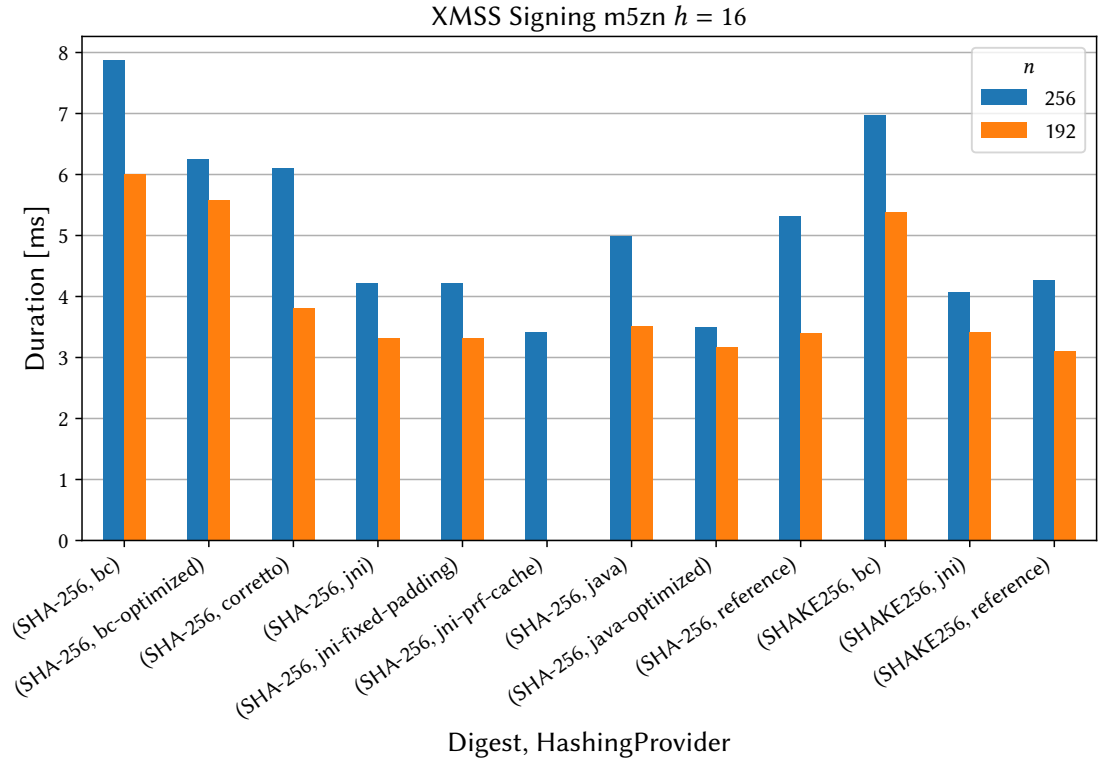


Figure A.5.: XMSS signing on `m5zn`

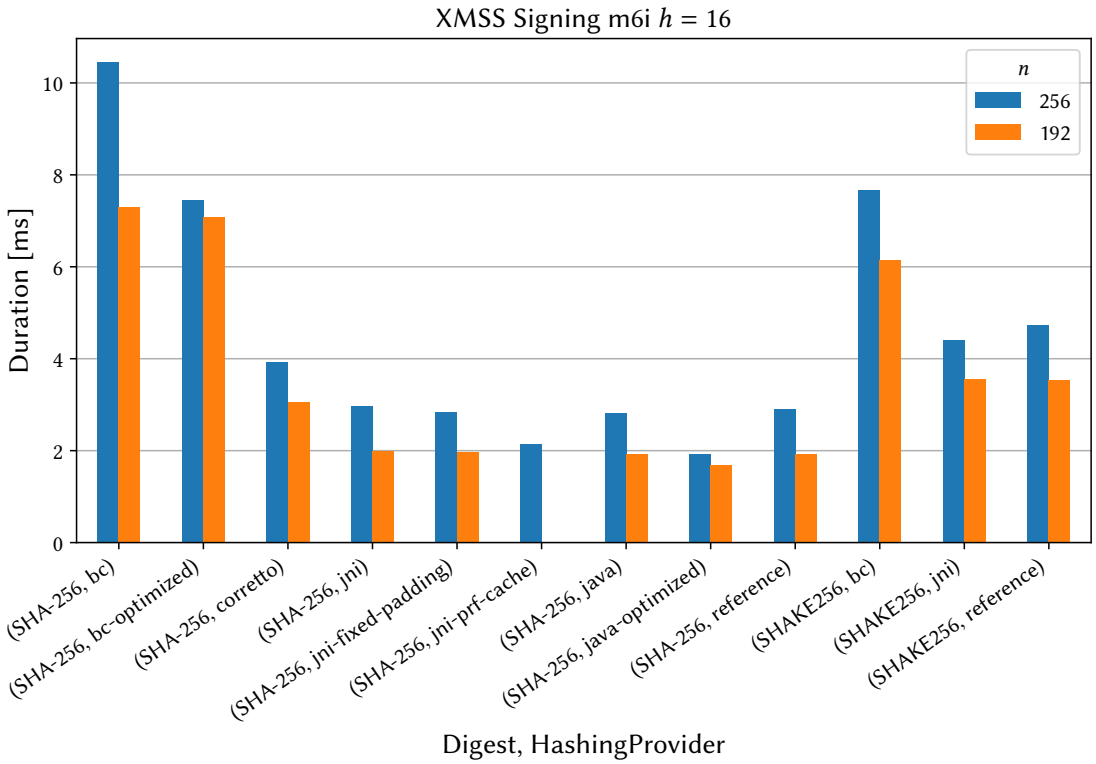


Figure A.6.: XMSS signing on `m6i`

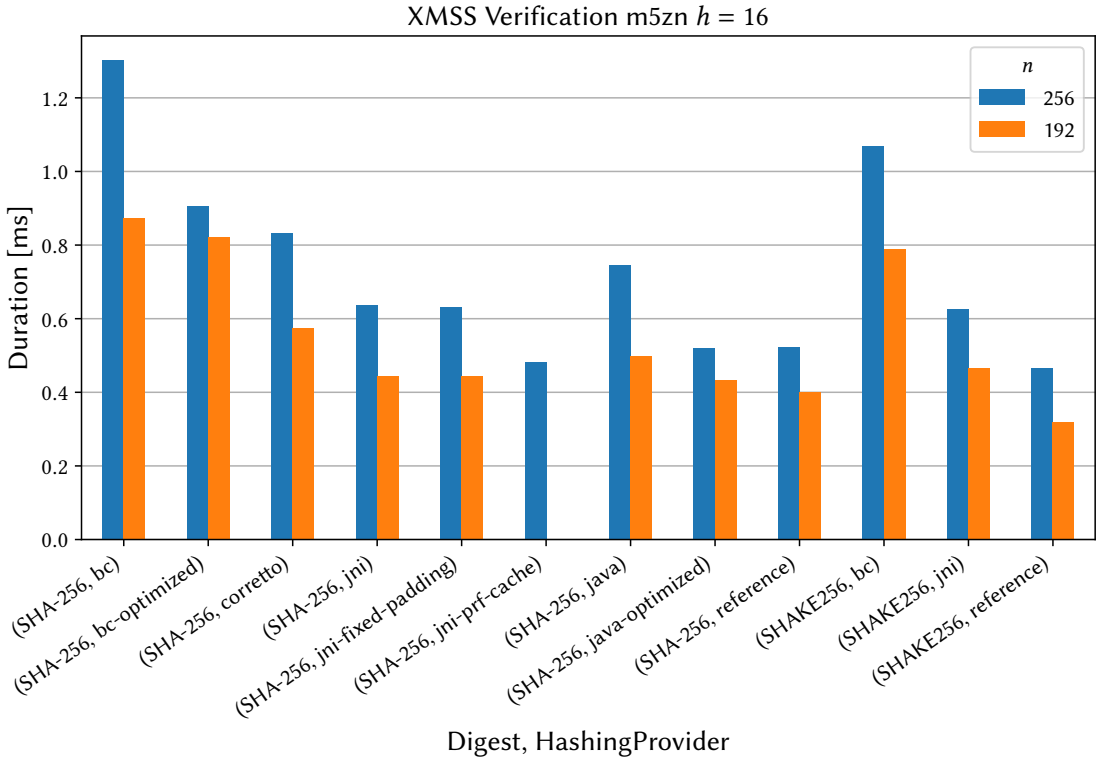


Figure A.7.: XMSS verification on `m5zn`

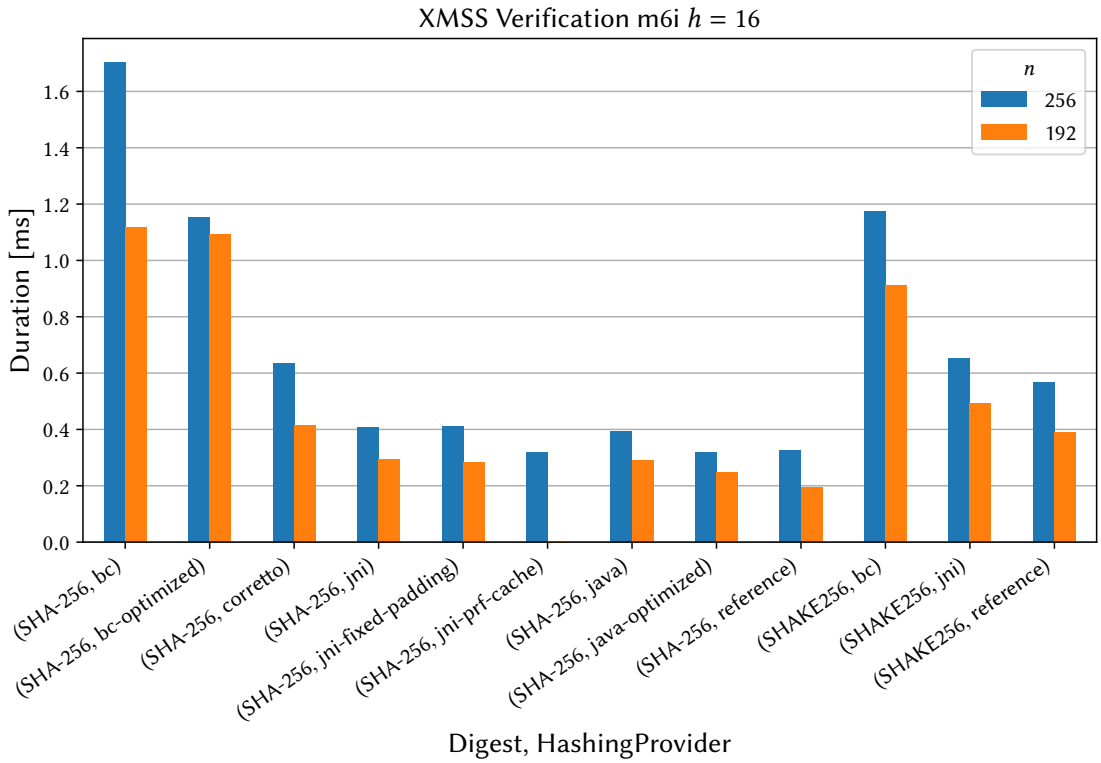


Figure A.8.: XMSS verification on `m6i`

A.2. LMS

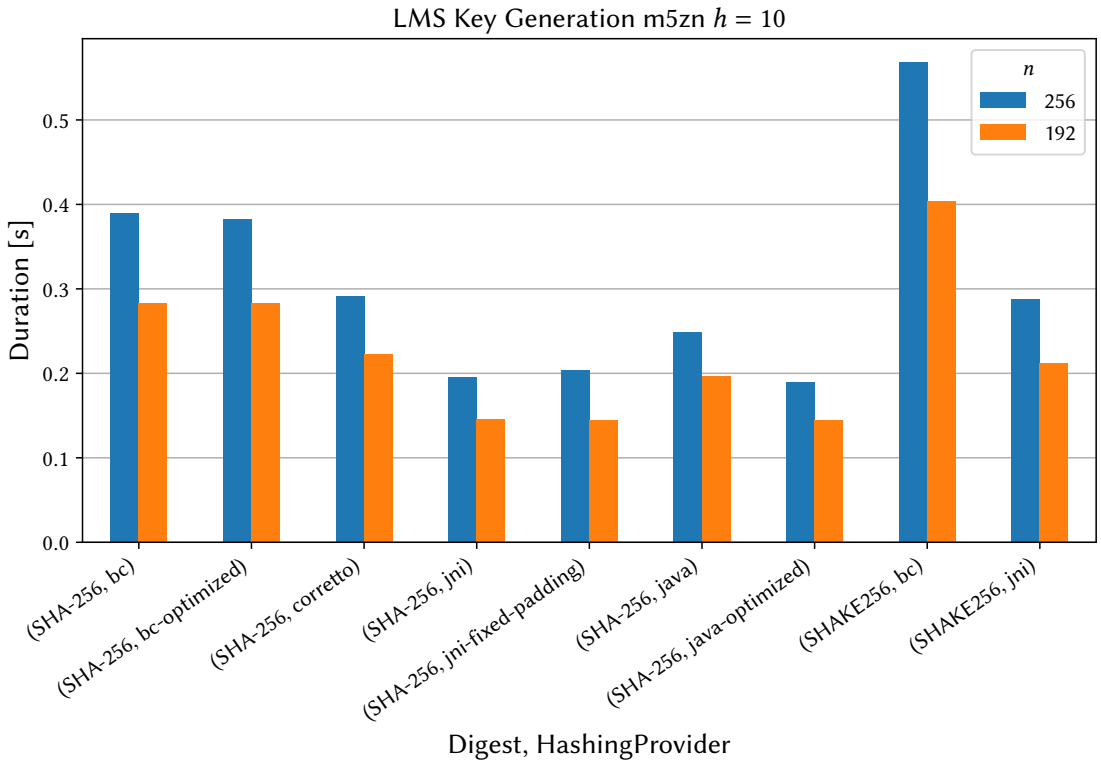


Figure A.9.: LMS key generation on `m5zn`

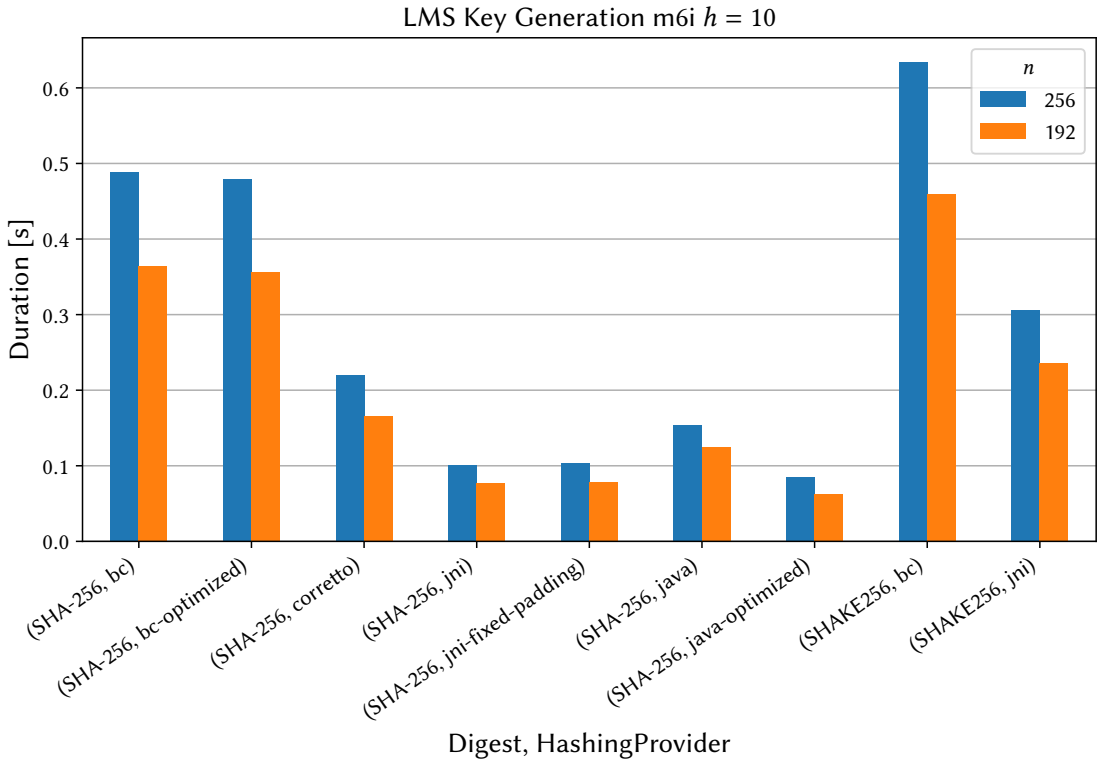


Figure A.10.: LMS key generation on `m6i`

A.3. SPHINCS+

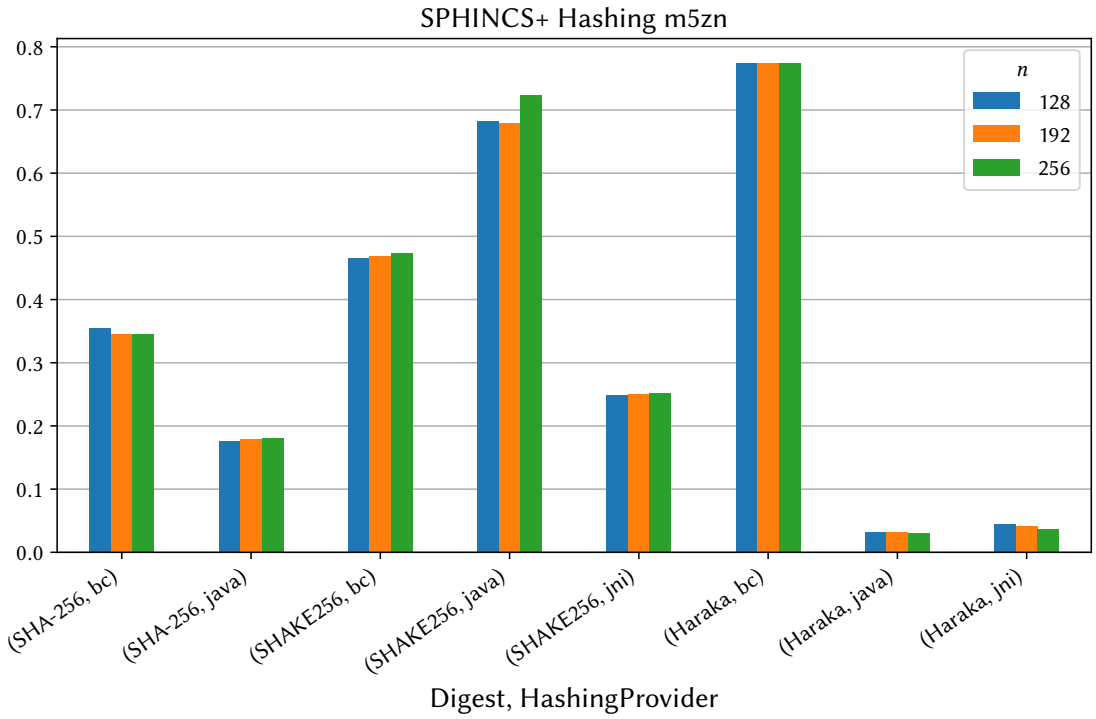


Figure A.11.: SPHINCS+ hashing on `m5zn`

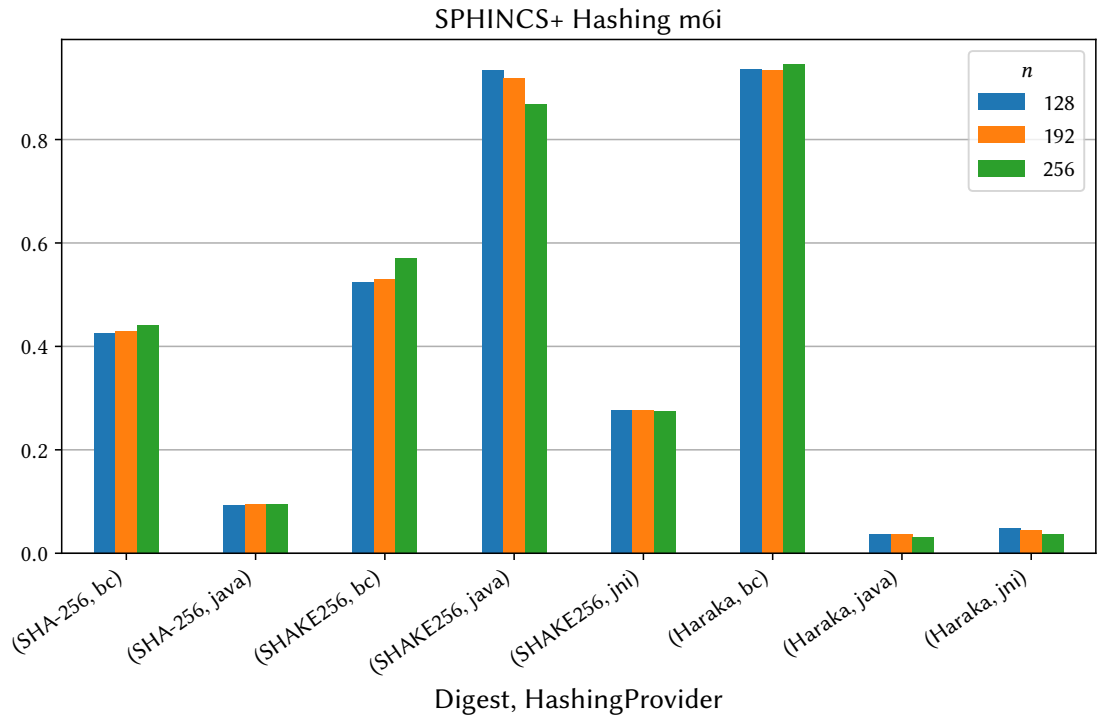


Figure A.12.: SPHINCS⁺ hashing on `m6i`

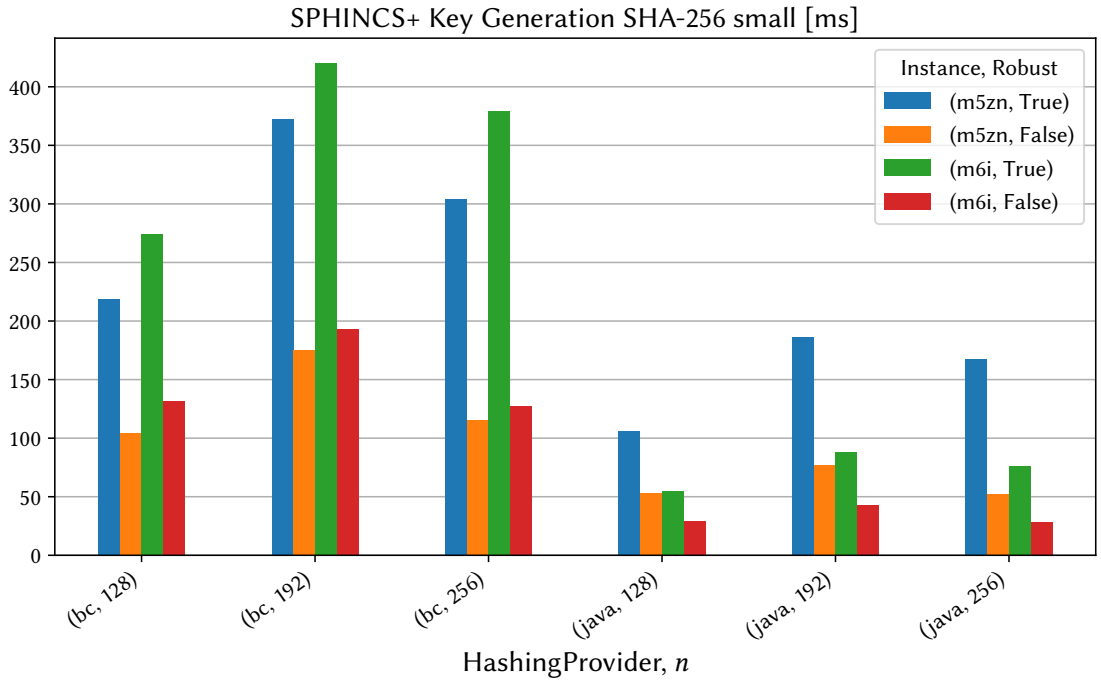


Figure A.13.: SPHINCS⁺ key generation with SHA-256

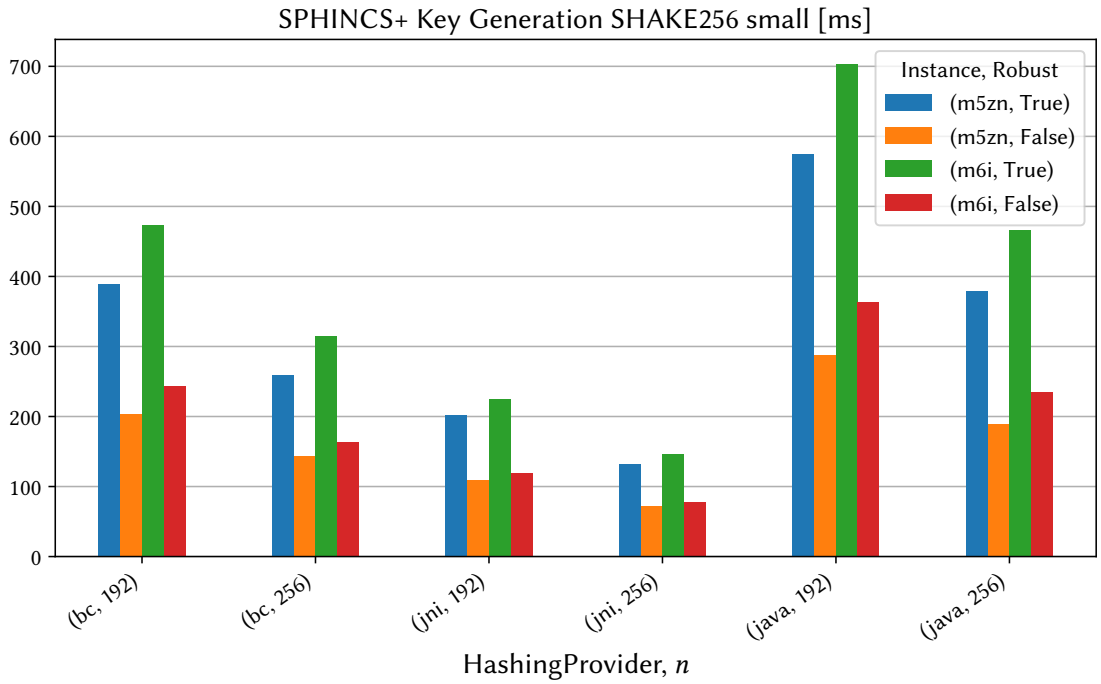


Figure A.14.: SPHINCS+ key generation with SHAKE256

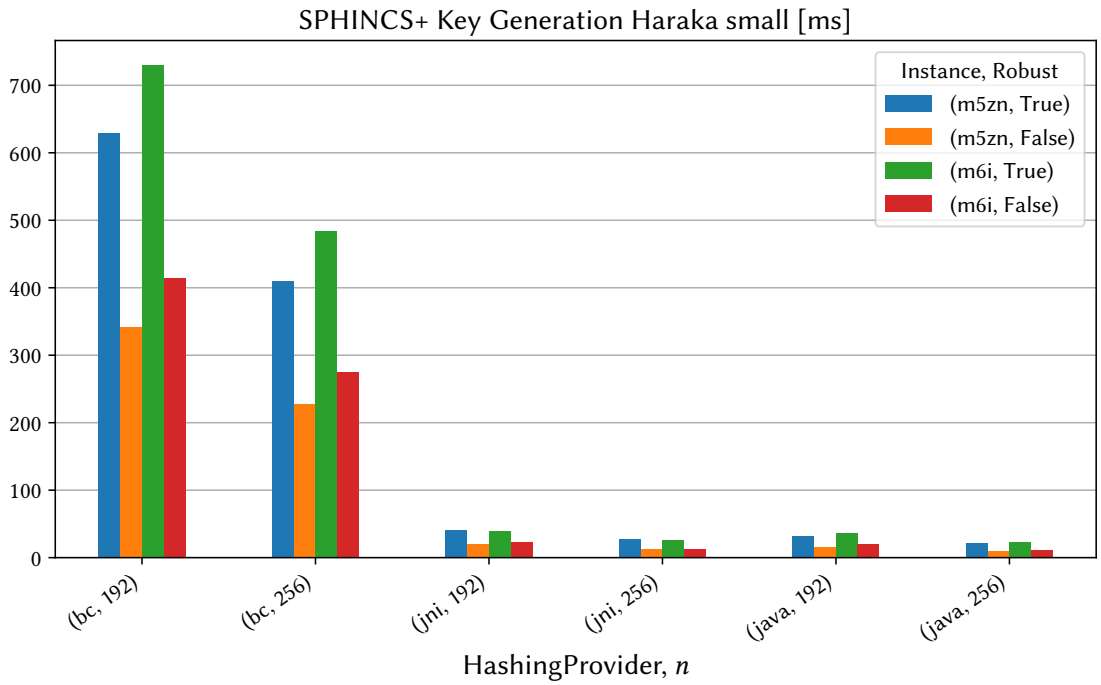


Figure A.15.: SPHINCS+ key generation with Haraka

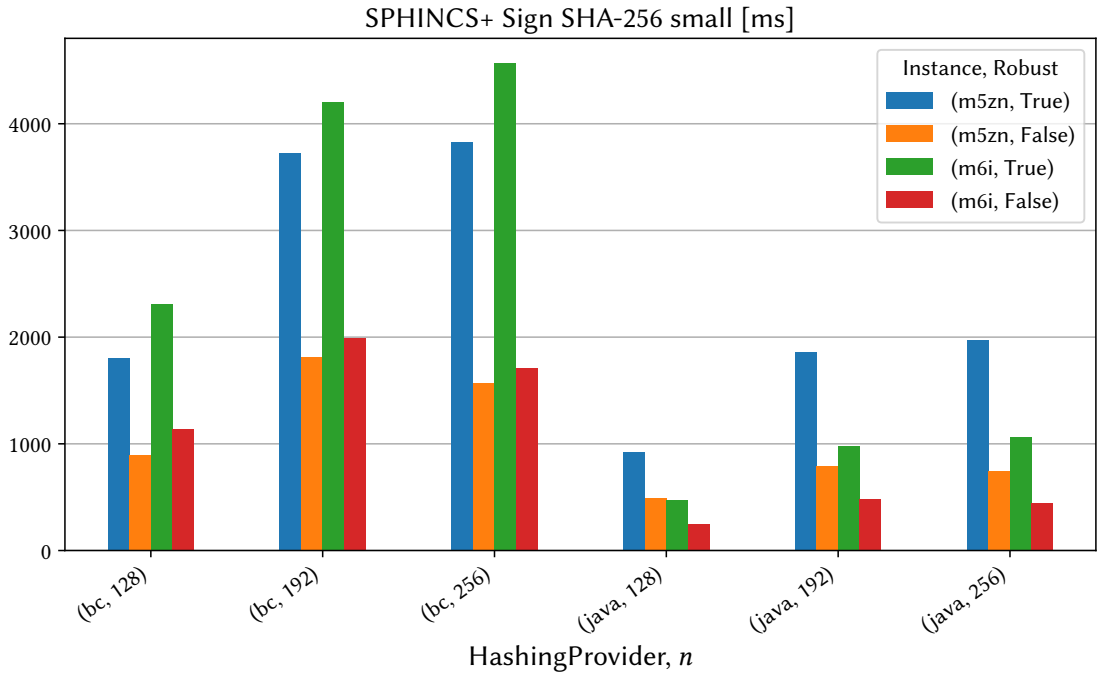


Figure A.16.: SPHINCS$^+$ signing with SHA-256

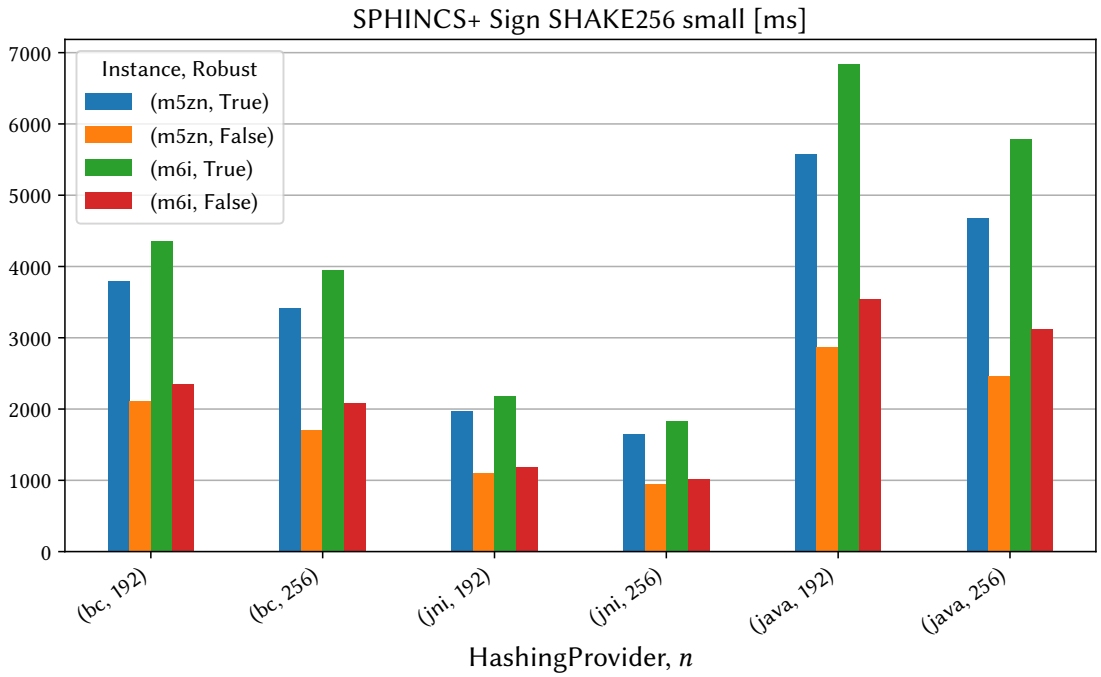


Figure A.17.: SPHINCS$^+$ signing with SHAKE256

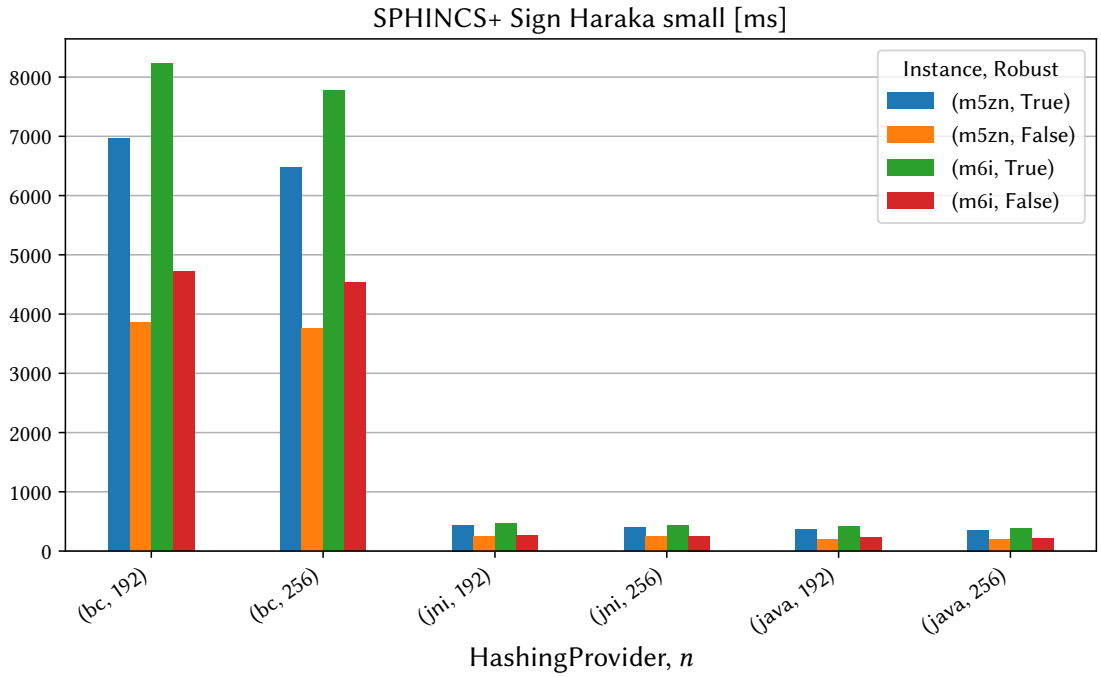


Figure A.18.: SPHINCS+ signing with Haraka