Master thesis
Computing Science
Digital Security

Radboud University

Proposal for a Pyramid scheme

Author:
Auke Zeilstra
s4751701

First supervisors/assessors:
prof. dr. Andreas Hülsing
a.t.huelsing@tue.nl

prof. dr. Peter Schwabe
p.schwabe@cs.ru.nl

dr. Bas Westerbaan
bas@westerbaan.name

Second assessor:
dr. Simona Samardjiska
simonas@cs.ru.nl

April 1, 2022

Abstract

Hash-based signatures are a conservative choice for a post-quantum digital signature scheme. In recent years, the stateless hash-based signature framework SPHINCS$^+$ has streamlined several aspects of hash-based signatures [BHK$^+$19]. The improvements from SPHINCS$^+$ have not been applied to stateful schemes in a manner that ensures compatibility with SPHINCS$^+$. In this work, we give a proposal for a stateful hash-based signature scheme named Pyramid. In the proposal, we give a description of Pyramid, along with an addressing scheme and hash-function instantiations. We include a concrete way of achieving forward security.

We accompany the proposal with four implementations, which range from naive to sophisticated, and regular to forward-secure. The sophisticated implementations use the Simple algorithm/reference code [KT21a], originally written for XMSS. We also attempt to restructure the signing routine into manageable subroutines.

Last, we aim to ease the formal verification process of future Pyramid implementations by including an implementation written in Jasmin. This may also serve as an experiment, in which we establish the applicability of the Jasmin language to the type of algorithm that we implement.

Contents

1 Introduction **4**
- 1.1 This work . . . 5

2 Preliminaries **6**
- 2.1 Historic background . . . 6
- 2.2 Stateful HBS: XMSS . . . 7
- 2.3 WOTS example . . . 8
- 2.4 Hypertree overview . . . 9

3 Pyramid description **11**
- 3.1 Preliminaries . . . 11
- 3.2 Tweakable hash functions . . . 12
- 3.3 Tweak legend . . . 13
- 3.4 Pyramid building blocks . . . 14
  - 3.4.1 WOTS-TW . . . 14
  - 3.4.2 WOTS-TW compression . . . 16
  - 3.4.3 The hypertree . . . 18
- 3.5 Pyramid . . . 19
  - 3.5.1 Pyramid key generation . . . 19
  - 3.5.2 Pyramid signature . . . 20
  - 3.5.3 Pyramid verification . . . 20
- 3.6 Pyramid proof status . . . 21
  - 3.6.1 Preliminaries . . . 21
  - 3.6.2 Partial SPHINCS$^+$ proof summary . . . 21
  - 3.6.3 Pyramid proof implication . . . 22

4 Pyramid instantiations **23**
- 4.1 SHAKE256 instantiations . . . 23
- 4.2 SHA256 instantiations . . . 24
- 4.3 Haraka instantiations . . . 26
- 4.4 Discussion . . . 27
  - 4.4.1 Function family independence . . . 28
  - 4.4.2 Address lengths . . . 28

4.4.3 Forward-secure instantiations 28
4.4.4 OptRand 28
4.4.5 XOF WOTS-TW secret generation 29
4.4.6 Non-repudiation 29
4.5 Addressing scheme 29
4.6 Key format 30

5 Pyramid implementations 31
5.1 Common 31
5.2 External algorithm & format 33
5.3 Stepping stone implementations 34
5.3.1 NFS-Naive 34
5.3.2 FS-StackRestore 35
5.4 NFS-Simple & background 37
5.4.1 BDS & HRB 37
5.4.2 Simple 38
5.5 FS-Simple 39
5.5.1 Forward-secure BDS 39
5.5.2 Forward-secure MMT 40

6 Implementations in Jasmin 44
6.1 General design choices 45
6.2 Directory structure 46
6.3 NFS-Naive 47
6.3.1 SHAKE256/FIPS 202 47
6.3.2 Treehash 47
6.3.3 WOTS 48
6.3.4 Root computation 49
6.3.5 Parameters & compilation 50
6.4 FS-Simple 50
6.4.1 Structures 50
6.5 Performance evaluation 51
6.5.1 System information 52
6.5.2 Results 52

7 Related Work 54

8 Conclusions and Future Work 55

A Figures 60
A.1 Pyramid format 60
A.2 Implementation key & state structures 61
A.3 MMT & BDS scheduling examples 62
A.3.1 MMT 62

A.3.2 BDS 63
A.4 Sign signature skeleton 64

B Software rescources 65
B.1 Directory structure 65
B.2 Jasmin build 66

Chapter 1

Introduction

Digital signature schemes (DSS) are used to achieve integrity and non-repudiation in the public-key setting. Integrity is achieved by verifying the authenticity of a message when given a signature for the message and an authentic public key. Examples of traditional digital signature schemes include RSA, DSA and ECDSA. Traditional digital signature schemes generally make at least two assumptions. First, a collision-resistant hash function is assumed, which compresses a variable-length message to a fixed-length digest. Second, a computational problem is assumed to be hard; examples include the integer factorization problem and DLP. In 1994, Shor published algorithms that compute discrete logarithms and factor integers, on a quantum computer, efficiently [Sho94]. This breaks the intractability assumptions that most traditional schemes make. Alternative assumptions for digital signature schemes in a post-quantum setting are found in certain categories of NP-hard problems, examples of which include SVP, BDD, and MQ. Other examples, and schemes based on these problems, can be found in [oST21].

An alternative is hash-based signature schemes (HBS). Hash-based signature schemes can base their security argument solely on the existence of a one-way function, which is a minimal assumption for a secure digital signature scheme. Hash-based signature schemes do have to take into account the quadratic speedup achieved by Grover's algorithm [Gro96] over classical search algorithms. However, note that Grover's algorithm does not parallelize well [Flu17].

Hash-based signatures are thought to be relatively well-understood compared to other post-quantum DSS alternatives, which has led to a recommendation for stateful hash-based signature schemes by NIST [oST20a]. The NIST-approved schemes include XMSS and LMS. Furthermore, SPHINCS$^+$ is a stateless hash-based signature framework, which is listed as a third-round alternative candidate algorithm for a DSS in the NIST PQC standardization process [oST21].

SPHINCS$^+$ introduces the notion of *tweakable hash functions*. SPHINCS$^+$ is based in part on XMSS, but includes further revisions that make it incompatible with previous stateful schemes. For this reason, we prepare a first proposal for Pyramid: a stateful hash-based signature scheme that is compatible with SPHINCS$^+$. We include the first attempt at reference implementations.

Implementations for (stateful) hash-based signature schemes can be complex and the process of creating one is error-prone. This is a broader issue in cryptography, which has led to frameworks for the formal verification of cryptographic schemes and their implementations [Mei21a]. EasyCrypt [BDG+13] is a toolset that can be used for the construction and verification of game-based cryptographic proofs. The Jasmin [ABB+17] framework offers a language and a compiler for creating high-assurance and high-speed cryptographic software. The Jasmin compiler is verified for functional correctness: a safe Jasmin source program can be compiled into a safe- and functionally equivalent assembly program [ABB+20a]. Furthermore, an embedding of the Jasmin language into EasyCrypt is defined.

We will provide the first attempt for a reference implementation of Pyramid in Jasmin, to take the first step towards formal verification of future versions of Pyramid. Due to the relatively high-level language structures that are used in stateful HBS implementations, this also provides an insight into the applicability of Jasmin in this setting.

1.1 This work

In Chapter 2, we indicate advances that led to the current state of affairs for stateful HBS. In Chapter 3, we provide a description of how we envision Pyramid in this proposal. Chapter 4 lists instantiations that attempt to achieve the properties that we desire in Chapter 3. Chapters 5 and 6 describe our approach to first implementations for Pyramid in C and Jasmin. Chapter 7 briefly relates Pyramid to LMS. Finally, Chapter 8 summarizes our results.

Chapter 2

Preliminaries

We provide a textual background on hash-based signatures; further details may be found in the cited works. A more complete/in-depth history can be found in [Rij19]. Because Pyramid is based on several schemes that we reiterate partially in Chapter 3, we omit a formal description here.

2.1 Historic background

The core concept of Hash-based signatures dates back to 1979, with the introduction of a One-Time Signature scheme (OTS) by Lamport [Lam79]. The security of the scheme solely relies on the existence of a one-way function. The number of bits that one can sign with a keypair is limited to half the number of digests that are part of the public key. To prove the authenticity of a message, one reveals preimages of digests in the public key. Revealed secrets are unique to the message, but their intersection for a different message may be non-empty. Signing two different messages with the same secret key reveals signatures for additional messages, thereby making the scheme "one-time".

Merkle improves the signature size by only revealing preimages for 1-bits, an improvement that is made possible by also signing a negated checksum [Mer89]. The paper accompanies this approach with a technique that allows decreasing signature- and public key size, even more, at the cost of additional signing- and verification time. This technique, accredited to Winternitz, is referred to as "WOTS". WOTS authenticates groups of message bits using just one secret/public element per group of bits, but multiple invocations of a one-way function F. Grouping more bits together requires the signer/verifier to compute longer hash chains, making this a time/space trade-off.

In [Mer89], Merkle uses binary hashing trees to authenticate a finite number of OTS instances, at the cost of having to include an authentication path, along with the OTS index, in signatures. The hash function that one

uses to combine nodes in the tree is commonly denoted H. In combination with WOTS, the public key can be reduced to just the tree root. A common procedure for generating (sub)roots is called “Treehash”. We include Treehash in Algorithm 1. One *gripe* of the scheme is that it must keep track of the OTS instances that have been used, by storing the OTS index in the secret key. This is incompatible with traditional DSS APIs and vulnerable to misuse [oST20a].

2.2 Stateful HBS: XMSS

In recent years, the hash-based signature scheme XMSS [BDH11] is introduced. XMSS and its successors use a binary hashing tree, with compressed WOTS^+ instances as their leaves. WOTS^+ does not need to rely on a collision-resistant hash function for its security argument, unlike WOTS [Rij19]. WOTS^+ is collision-resilient: it is not vulnerable to collision attacks against the used hash function. One has to attack WOTS-TW by attacking either second preimage resistance, one-wayness, or undetectability of the used function family [Hül13b].

XMSS^{MT} [HRB13] uses XMSS trees to sign the roots of other XMSS trees, similar to a (binary) authentication tree. OTS instances that are authenticated by trees at the bottom are used to sign messages. XMSS^{MT} allows one to attain a high number of WOTS keypairs, without having to generate all of the leaf nodes at once to obtain a tree root.

Whereas verification is one standard routine, signature generation for XMSS and XMSS^{MT} can freely choose *tree-traversal* algorithms to generate authentication paths. Amongst other things, tree-traversal algorithms retain- and manage tree nodes to speed up computation of the authentication path. We will sketch a background on two such algorithms, BDS [BDS08] and Simple [KT21a], in Chapter 5.

Finally, XMSS-T [HRS16] recognizes that an attacker can gather a large amount of images from the aforementioned XMSS scheme instance(s) to mount a multi-target attack. To mitigate such attacks, XMSS^{MT} uses a user-specific function key and an invocation-specific nonce to make hash function input unique.

At the time of writing, XMSSs “final form” is found in RFC 8391 [HBG+18]. RFC 8391 provides a great amount of detail on practical details, such as hash function- and addressing scheme instantiations. RFC 8391 was written *after* the proposal for the stateless SPHINCS. Inconveniently, advancements from SPHINCSs successor, SPHINCS^+, are relevant to stateful signature schemes, but postdate RFC 8391. The advancements, combined with the fact that the other NIST-approved stateful HBS scheme LMS is incompatible with the constructions in the SPHINCS^+ framework, are the main motivation for the proposal for Pyramid.

2.3 WOTS example

In Section 3.4.1, we reiterate the description of WOTS-TW, which is originally found in [HK21]. To prevent duplicate descriptions, we omit a description of WOTS. Instead of a description, we include a WOTS example in Figure 2.1. Readers that are familiar with WOTS may prefer to skip this section.

Figure 2.1 depicts a WOTS instance for $w = 4$; we clarify WOTS parameters further in Section 3.4.1. From here, we use binary representation for the example by default. In the example, the WOTS instance signs a message $m = (00, 10, 11)_w$. Figure 2.1 highlights the hash chain values after m_i applications of the hash function. For example, $m_0 = 00$, meaning that we include sk_0 in the WOTS signature. The values sk_i are commonly generated from a shorter seed (using a PRG) to reduce the storage space required for the secret key.

Without the inclusion of a signature over an inverted checksum, trivial forgeries appear. Without the checksum, the only value that we can authenticate for a message group m_i is $m_i = 00$.

The inverted checksum in the example is given by $c = \sum_{i=0}^{2} \neg m_i = 4_{10} = (01, 00)_w$. The number of chains for the checksum must be able to represent the number $3_{10} \cdot 11 = 9_{10}$ in base-w; the example requires two chains. Perhaps the simplest forgery attempt is one in which an adversary forwards one of the hash chains in an original signature. Thereby, the adversary hopes to authenticate the original message with an increased m_i. The checksum prevents such an attack: some $c_i' < c_i$ for the checksum value c_i' in the forgery, demanding the adversary to include an unrevealed secret.

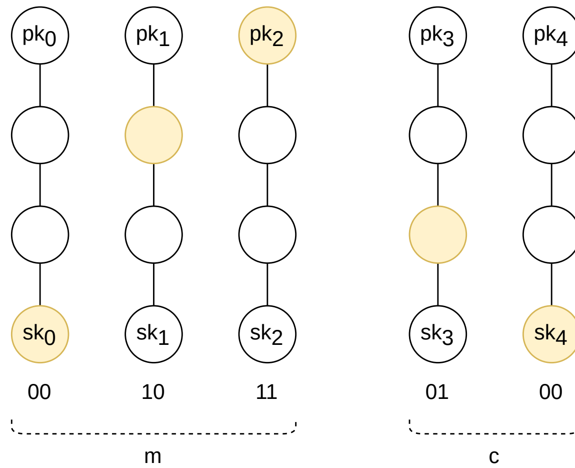

Figure 2.1: A WOTS example, inspired by Figure 3.2 in [Rij19].

2.4 Hypertree overview

Figure 2.2 depicts the certification tree construction from XMSS$^{\text{MT}}$, also referred to as a hypertree. The figure depicts a "full tree height" of $h = 6$, which we achieve using $d = 2$ layers of subtrees. We clarify these parameters further in Section 3.4.3. In the construction that is depicted in Figure 2.2, the OTS instances are used to sign messages at the bottom layer. At higher layers, the OTS instances sign the roots of trees on lower layers. The public keys of the OTS instances serve as the data blocks in the Merkle tree. A compressed public key serves as a *leaf node*. A non-leaf node is referred to as an *inner nodes*, or simply a *node*.

The highlighted nodes in Figure 2.2 depict authentication path nodes. In each tree layer, the corresponding authentication path nodes serve as a Merkle proof. The Merkle proof demonstrates the membership of an OTS public key in the tree on the corresponding layer. In the example, we sign a message m using an OTS instance. The OTS instance has a public key that serves as a data block in the Merkle tree. We prove membership for the public key in the bottom tree by including the nodes that are highlighted in the bottom tree in a signature.

In a singular Merkle tree, or the top tree in the hypertree construction, we then verify membership of a public key in the tree by "recomputing" the root node, using the authentication path. We check the resulting root node against the root node that we acquire from a trusted source.

On the lower layers of the hypertree, we do not directly compare the supposed subtree root to an authenticated subtree root in the public key. This would require a large number of roots in the public key. Instead, the signer includes an OTS signature for the subtree root. The verifier uses the OTS signature to recover a public key, for which one again checks the Merkle proof in the (next) subtree. We repeat this process until the inclusion of the root in the public key becomes feasible. In practice, and Figure 2.2, we only include the absolute hypertree root (labelled "pk") in the public key.

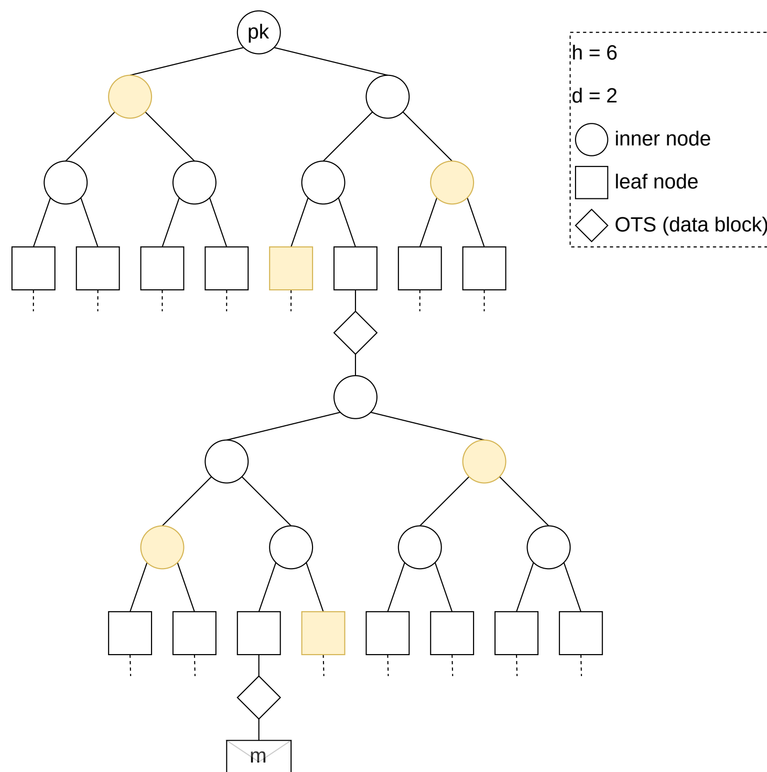

Figure 2.2: The hypertree construction, inspired by Figure 1 in [BHK+19].

Chapter 3

Pyramid description

In this chapter, we provide a description of Pyramid. We describe the building blocks for Pyramid in Section 3.4. These descriptions effectively serve as a mixed restatement of the descriptions of [BDH11], [HRB13], [HRS16], and [HK21]. Section 3.3 does not describe a building block per se, but gives a first indication of the role of context in Pyramid. Then, we describe Pyramid in Section 3.5 based on the building blocks.

3.1 Preliminaries

The Pyramid description includes an optional way of achieving forward security. A forward-secure DSS evolves its secret key over time, using a one-way function to update the secret key. If an adversary compromises the secret key, then our degree of trust in the authenticity of previously signed messages remains unchanged, as long as these were signed with a prior secret key [BM99]. The public key remains fixed.

Forward security of Pyramid is a result from generating WOTS secret values using a forward-secure generator (FSG). A forward-secure (pseudorandom bit) generator is a stateful generator. Informally, a pseudorandom generator (PRG) efficiently transforms a uniform string into a longer pseudorandom string. A PRG output must not be efficiently distinguishable from a truly random string of equal length. We reiterate a slightly modified definition of a PRG, from [Rij19], in Definition 3.1.1.

A stateful generator allows one to generate a finite number of pseudorandom strings. Unlike a PRG, a stateful generator is a stateful object. Given a state/seed, requesting output grants a pseudorandom string along with the state for a successive request. Finally, an FSG ensures that a compromised state does not allow an attacker to efficiently distinguish previous outputs from truly random strings.

Definition 3.1.1 (Pseudorandom generator (PRG))**.** Let G be an efficient algorithm implementing the function $G : \{0,1\}^n \to \{0,1\}^l$, for l polynomial

in n. G is a pseudorandom generator if the following conditions hold:

1. For every n it holds that $l > n$.
2. For any probabilistic polynomial-time algorithm $\mathcal{A}$, given $U_0 = G(x)$ for $x \xleftarrow{\$} \{0,1\}^n$, $U_1 \xleftarrow{\$} \{0,1\}^l$, and $b \xleftarrow{\$} \{0,1\}$, the success probability of running $\mathcal{A}(U_b)$ to find b differs negligibly from random guessing.

The FSG- and PRG constructions that we use in Pyramid can be found in [HRB13] [Hül13a]. These use a construction from [BY03], titled "A [PRG] construction based on PRFs". The paper also includes a way of constructing a (forward-secure) stateful generator from a standard PRG. We reiterate a slightly modified definition of a length-preserving pseudorandom function (PRF) from [Rij19] in Definition 3.1.2:

Definition 3.1.2 (Pseudorandom function (PRF))**.**
Let $F : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a family of efficient, length-preserving functions. We say F is a pseudorandom function if, for any probabilistic polynomial-time algorithm $\mathcal{A}$ and $k \xleftarrow{\$} \{0,1\}^n$, the success probability of running $\mathcal{A}(F_k)$ to distinguish between F_k and a truly random function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ differs negligibly from random guessing.

3.2 Tweakable hash functions

One of the main goals for Pyramid is to achieve verification routine compatibility with SPHINCS$^+$, with minimal additional scheme-specific code. In part, Pyramid achieves this goal by including the abstraction of tweakable hash functions from SPHINCS$^+$. Generally, to be able to reuse most of the proof of security of SPHINCS$^+$ for Pyramid, we should embrace the same abstractions when possible.

Section 2 references several versions of XMSS. SPHINCS$^+$ recognizes that node generation, in WOTS and the (hyper)tree construction, is the primary variation between the XMSS versions. Still, each scheme includes a separate security analysis. Contrarily, the description for SPHINCS$^+$ leaves the way in which it generates nodes open. The specification leaves node generation up to tweakable hash functions. Enforcing a certain way of computing nodes requires the specification of a tweakable hash function construction. Other parts of the SPHINCS$^+$ description can remain unaltered. Still, one must make sure that the construction for tweakable hash functions achieves the properties that the SPHINCS$^+$ specification requires from tweakable hash functions. These property requirements arise from the proof of security of SPHINCS$^+$.

SPHINCS$^+$ includes two constructions of tweakable hash functions, for which it includes instantiations. In summary, tweakable hash functions allow

for separation between analysis of node computation strategies and the high-level SPHINCS$^+$ construction.

We repeat the definition of a tweakable hash function from SPHINCS$^+$ [HK21] in Definition 3.2.1. In the definition, note that in practice $n \mid m$ holds: a tweakable hash function maps one or more nodes to a single one. The public parameter and the tweak arguments allow SPHINCS$^+$ to make hash-function calls independent between different keypairs and hypertree locations. The generic constructions of tweakable hash functions remain unchanged in Pyramid, these can be found in the SPHINCS$^+$ description [BHK$^+$19].

Definition 3.2.1 (Tweakable hash function)**.** Let $n, m \in \mathbb{N}$. $\mathcal{P}$ is the public parameter space and $\mathcal{T}$ is the tweak space. A tweakable hash function is an efficient function: $\mathrm{Th} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^m \to \{0,1\}^n$. It maps an m-bit message to an n-bit value, given a public parameter (function key) $\mathrm{P} \in \mathcal{P}$ and a tweak (nonce) $\mathrm{T} \in \mathcal{T}$.

We denote $\mathrm{Th}(\mathrm{P}, \mathrm{T}, M)$ as $\mathrm{Th}_{\mathrm{P},\mathrm{T}}(M)$. We use $\mathrm{F}, \mathrm{H}, \mathrm{Th}_\lambda$ for Th with input length $m = n, 2n, ln$, respectively. n is the security parameter for Pyramid and l is the number of nodes that we include in a WOTS signature. We clarify the parameters further in Section 3.4.1.

3.3 Tweak legend

The following sections describe the foundation of Pyramid. Most concepts use a tuple $\mathcal{C}$ that contains context information; examples include a prefix ADRS and a public parameter Seed. The prefix ADRS in $\mathcal{C}$ allows the overarching structures to ensure that tweaks in its components differ. For example, we grant a Merkle tree in Pyramid a dedicated address space, which is unique in the entirety of the Pyramid *structure*. We define the Pyramid construction/structure in Section 3.5. All node computations in this Merkle tree use a unique address within this dedicated address space. Finally, the public parameter (Seed) and hypertree root (Root) diversify hash-function input between Pyramid instances.

Every section in Table 3.1 describes a construction. The column ADRS shows the type of address space that we dedicate to these constructions. To acquire a complete address, hereafter referred to as a tweak T, a construction appends the fields that are listed in the T column to its given prefix. We omit function symbol prefixes from here in ADRS; these follow from context. The combination "tree.keypair" is synonymous with "index" in Section 3.4.2.

Note that usage of the legend is optional for understanding the Pyramid building blocks. The main purpose of Table 3.1 is to provide an indicator of the way in which we manage unique tweaks in Pyramid. Practical details of the addressing scheme are found in Section 4.5.

Section	ADRS (prefix)	T
WOTS-TW (3.4.1)	layer.tree.F.keypair	chain.hash
WOTS-TW pk compression (3.4.2)	layer.tree.Th_λ	keypair
Merkle tree (3.4.3)	layer.tree.H	height.index
Message hash (3.4.2)	$\mathrm{H}_{\mathrm{msg}}$	tree.keypair
Message hash randomization (3.4.2)	PRF	tree.keypair
WOTS-TW sk expansion (3.4.1)	layer.tree.$\mathrm{PRF}_{\mathrm{kg}}$	keypair
Forward-secure $\mathcal{S}$ generation (3.4.2)	layer.tree.FSG.keypair	direction

Table 3.1: Relation between ADRS and T for every building block.

3.4 Pyramid building blocks

3.4.1 WOTS-TW

Pyramid uses a variation of the WOTS scheme as its OTS, which is referred to as WOTS-TW. WOTS-TW is defined for SPHINCS$^+$ in [HK21]. WOTS-TW specifies its chaining function in terms of tweakable hash functions. This is unlike other WOTS versions.

For a WOTS-TW instance, this means that it must acquire a piece of context information $\mathcal{C} = (\mathrm{Seed}, \mathrm{ADRS})$. This allows the WOTS-TW instance to achieve uniqueness between its Th calls, both between Pyramid instances and within the Pyramid structure. The toy example that is shown in Figure 3.1 may be of help. In Figure 3.1, we essentially "zoom out" from the WOTS-TW instance on the left, thereby uncovering the structure built around this WOTS-TW instance.

Our description of WOTS-TW is very close to that of [HK21]. We will apply the same notation, with the addition of $\mathrm{T}_{i,j} \coloneqq \mathrm{T}[i, j]$: a completed tweak for hash j in chain i, according to 3.3. We use this notation because we abstain from explicitly defining the chaining function, which is primarily convenient in a proof that we do not give.

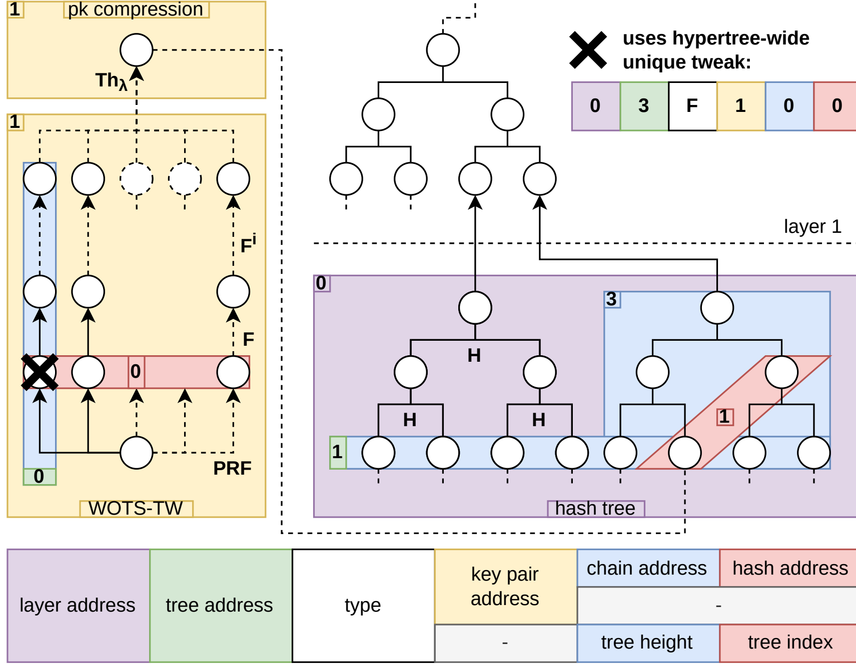

Figure 3.1: Toy example that shows the tweak for the node marked "×".

Parameters

WOTS-TW has the following parameters:

1. $n \in \mathbb{N}$ Security parameter;
2. $w \in \mathbb{N}, w > 1$ Winternitz parameter;
3. $m \in \mathbb{N}$ Message length; $m = n$ for Pyramid.

n represents the length of a secret key/public key/signature element in bits. In practice, the m-bit "message" is an n-bit message digest of a variable-length message. We further define:

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

l_1 n-bit values represent the message limbs that we sign, and l_2 n-bit values represent the accompanying checksum limbs in the signature. We sign the message and its checksum in identical fashion. We require a tweakable hash function $\mathrm{F} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$. We require one pseudorandom function:

$$\begin{cases} \mathrm{PRF_{kg}} : \{0,1\}^n \times \mathcal{T} \to \{0,1\}^n & \text{Regular Pyramid} \\ \mathrm{PRF_{kg}} : \mathcal{P} \times \{0,1\}^n \times \mathcal{T} \to \{0,1\}^n & \text{Forward-secure Pyramid} \end{cases}$$

Key generation: $((\mathrm{SK}, \mathrm{PK}) \leftarrow \mathrm{WOTS.kg}(\mathcal{C}; \mathcal{S}))$

Given the pseudorandom secret seed $\mathcal{S}$ and Pyramid context information $\mathcal{C} = (\mathrm{Seed}, \mathrm{ADRS})$, we generate the WOTS-TW secret key $\mathrm{sk} = (\mathrm{sk}_1, \ldots, \mathrm{sk}_l)$ following:

$$\mathrm{sk}_i = \begin{cases} \mathrm{PRF}_{\mathrm{kg}}(\mathcal{S}, \mathrm{T}[i, 0]) & \text{Regular Pyramid} \\ \mathrm{PRF}_{\mathrm{kg}}(\mathrm{Seed}, \mathcal{S}, \mathrm{T}[i, 0]) & \text{Forward-secure Pyramid} \end{cases} \quad \text{for } 1 \leq i \leq l$$

We compute the WOTS-TW verification key $\mathrm{pk} = (\mathrm{pk}_1, \ldots, \mathrm{pk}_l)$ as:

$$\mathrm{pk}_i = (\mathrm{F}_{\mathrm{Seed}, \mathrm{T}[i, w-2]} \circ \cdots \circ \mathrm{F}_{\mathrm{Seed}, \mathrm{T}[i, 0]})(\mathrm{sk}_i), 1 \leq i \leq l$$

We return $\mathrm{SK} = (\mathcal{S}, \mathcal{C}), \mathrm{PK} = (\mathrm{pk}, \mathcal{C})$. Unlike WOTS$^+$/WOTS-T, WOTS-TW *enforces* sk compression by returning $\mathcal{S}$ instead of sk.

Signing: $(\sigma \leftarrow \mathrm{WOTS.sign}(M, \mathrm{SK})$

Given an m-bit message M and a secret key SK, we compute base-w representations for the message M and the negated checksum C as follows:

$$M = (M_{l_1}, \ldots, M_1)_w, \quad C = \sum_{i=1}^{l_1} (w - 1 - M_i), \quad C = (C_{l_2}, \ldots, C_1)_w$$

The concatenation $B = (b_l \ldots b_1)_w = (M \| C)$ consists of l base-w values; b_i corresponds to the number of applications of F on sk_i while computing the signature. We compute the signature $\sigma = (\sigma_1, \ldots, \sigma_l)$ as:

$$\sigma_i = \mathrm{F}_{\mathrm{Seed}, \mathrm{T}[i, b_i - 1]} \circ \cdots \circ \mathrm{F}_{\mathrm{Seed}, \mathrm{T}[i, 0]}(\mathrm{sk}_i), 1 \leq i \leq l$$

Verification: $(\mathrm{pk}' \leftarrow \mathrm{WOTS.vf}(M, \sigma, \mathrm{PK}))$

Given an m-bit message M, a signature σ, and a public key PK, we compute B as shown in Section 3.4.1. We compute $\mathrm{pk}' = (\mathrm{pk}'_1, \ldots, \mathrm{pk}'_l)$ as follows:

$$\mathrm{pk}'_i = (\mathrm{F}_{\mathrm{Seed}, \mathrm{T}[i, w-2]} \circ \cdots \circ \mathrm{F}_{\mathrm{Seed}, \mathrm{T}[i, b_i]})(\sigma_i), 1 \leq i \leq l$$

If WOTS-TW were to be used as a standalone OTS, one would return 1 when pk' equals pk, and 0 otherwise.

3.4.2 WOTS-TW compression

WOTS$^+$ was commonly used alongside some form of secret key-, message- and verification-key compression when part of another construction. This is also the case in Pyramid, with secret key compression being the "default" of WOTS-TW. We build upon parameters from Section 3.4.1 and follow the tweak legend from Section 3.3. To avoid confusion between states and seeds, we will refer to an FSG *state* as a reference.

Forward-secure $\mathcal{S}$ generation

We sample a starting FSG reference $\mathcal{R}_0 \xleftarrow{\$} \{0,1\}^n$. We use Pyramid context information $\mathcal{C} = (\text{Seed}, \text{ADRS})$. We require a pseudorandom function $\text{PRF} : \mathcal{P} \times \{0,1\}^n \times \mathcal{T} \rightarrow \{0,1\}^n$. For forward-secure Pyramid, the value of a WOTS-TW key pair seed $\mathcal{S}_i$ from Section 3.4.1 follows from:

$$\text{FSG}(\text{Seed}, \mathcal{R}_i, \mathcal{C}) = (\mathcal{R}_{i+1} \parallel \mathcal{S}_i) = (\text{PRF}_{\text{Seed},\mathcal{R}_i}(\text{T}[0]) \parallel \text{PRF}_{\text{Seed},\mathcal{R}_i}(\text{T}[1]))$$

Message compression

We perform the initial message compression in line with the message compression construction for XMSS$^{\text{MT}}$ [HBG+18]. We require one randomization element $r \in \{0,1\}^n$ and Pyramid context information $\mathcal{C} = (\text{Root}, \text{ADRS})$. Finally, we require a cryptographic hash function $\text{H}_{\text{msg}} : \{0,1\}^n \times \mathcal{P}' \times \mathcal{T} \times \{0,1\}^* \rightarrow \{0,1\}^n$. Compression MD of an arbitrary-length message M that we compress for signing is given by

$$MD = \text{H}_{\text{msg}}(r, \text{Root}, \text{T}, M),$$

where r randomizes the hash evaluation, while Root and T makes the hash function call user- and position-dependent, respectively [HBG+18]. In Pyramid, r is a pseudorandom value that we generate as follows. We require a pseudorandom function $\text{PRF} : \{0,1\}^n \times \mathcal{T} \rightarrow \{0,1\}^n$, along with Pyramid context information $\mathcal{C} = \text{ADRS}$ and a secret value $S \in \{0,1\}^n$. We generate $r \in \{0,1\}^n$ following: $r = \text{PRF}_S(\text{T})$.

Verification key compression

We require a WOTS-TW verification key $\text{pk} = (\text{pk}_1, \ldots, \text{pk}_l)$ and Pyramid context information $\mathcal{C} = (\text{Seed}, \text{ADRS})$. Given a tweakable hash function $\text{Th}_\lambda : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{ln} \rightarrow \{0,1\}^n$, we compress the verification key to an n-bit value N, later referred to as a leaf *node*, conform [BHK+19]:

$$N = \text{Th}_\lambda(\text{Seed}, \text{T}, \text{pk}_1 \| \ldots \| \text{pk}_l)$$

3.4.3 The hypertree

Parameters

The two hypertree parameters are as follows:

1. $h \in \mathbb{N}$ Tree height;
2. $d \in \mathbb{N}$, for $d|n$ Number of intermediate layers.

Like SPHINCS$^+$, we require a tweakable hash function $\mathrm{H} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{2n} \rightarrow \{0,1\}^n$. We will start with a description of a standalone tree, similar to what is shown in [Mer89] [DOTV08]. Note that computation of the tree root and the *authentication path* (defined in Section 3.4.3) may take advantage of optimized algorithms such as BDS [BDS08]. The descriptions that we give below aim to describe the Pyramid hypertree structure, without specifying computation strategies. The first three sections assume one tree, i.e. $d = 1$, for clarity.

Pyramid tree

A Pyramid tree of height $h \in \mathbb{N}$ is a binary hash tree with $h+1$ levels. The root node is located at level h and the leaf nodes at level 0. The jth node on level i is denoted by $N_{i,j}$ for $0 \leq j < 2^{h-i}, 0 \leq i \leq h$. We compute node $N_{i,j}$ for $0 < i \leq h$ as:

$$N_{i,j} = \mathrm{H}(\mathrm{Seed}, \mathrm{T}[i,j], N_{i-1,2j} \| N_{i-1,2j+1})$$

Note that we choose $\mathrm{T}[i,j]$, aligning with the node $N_{i,j}$ we compute.

Pyramid tree root (Root $\leftarrow$ GenRoot($\mathcal{C}, \mathcal{R}$))

We are given a secret $\mathcal{R} \in \{0,1\}^n$, which is either an FSG reference or a regular secret. We also take Pyramid context information $\mathcal{C} = (\mathrm{Seed}, \mathrm{ADRS})$. We will denote $\mathcal{C}[i] := (\mathrm{Seed}, \mathrm{ADRS}[i])$ as the context for the ith WOTS-TW instance in this tree. If this is a forward-secure Pyramid instance, we compute $\mathcal{S}_i$ for $0 \leq i < 2^h$ conform Section 3.4.2 for $\mathcal{R}_0 = \mathcal{R}$. We then compute:

$$(\mathrm{SK_i}, \mathrm{PK_i}) \leftarrow \begin{cases} \mathrm{WOTS.kg}(\mathcal{C}[i], \mathcal{R}) & \text{Regular Pyramid} \\ \mathrm{WOTS.kg}(\mathcal{C}[i], \mathcal{S}_i) & \text{Forward-secure Pyramid} \end{cases} \quad \text{for } 0 \leq i < 2^h$$

We then compress PK_i into $N_{0,i}$ according to Section 3.4.2, in context $\mathcal{C}[i]$. Finally, we compute Root $= N_{h,0}$ following Section 3.4.3.

Pyramid authentication path

Assume a set of base nodes $N_{0,i}$ for $0 \leq i < 2^h$ and the $2^h - 1$ combinations following Section 3.4.3. To compute the authentication path Auth_k for node $N_{0,k}$, we return the nodes $(N_{0,j_0}, \ldots, N_{h-1,j_i})$ for $j_i = \lfloor \frac{k}{2^i} \rfloor \oplus 1,\ 0 \leq i < h$.

Now, given a root node $N_{h-1,0}$ and a leaf node $N_{0,k}$, a Merkle proof Auth_k demonstrates that $N_{0,k}$ is part of the root's tree. A concrete computation strategy for the authentication path is Treehash, found in Algorithm 1.

The hypertree

We use the "hypertree" construction from [HRB13]. The goal is to generate a tree that allows for 2^h signatures, but the cost of generating this tree is computationally expensive. Instead, we define d layers of trees of height h/d. A tree on the bottom layer $d = 0$ contains $2^{h/d}$ WOTS-TW instances, used to sign arbitrary-length messages chosen by the user. We use the following notation:

$$\text{idx} = (b_d \dots b_1)_{2^{h/d}},\ \text{idx}_i = (b_{i+1} \dots b_1)_{2^{h/d}},\ \text{idx}'_i = (b_d \dots b_{i+2})_{2^{h/d}}$$

At hypertree layer $i = 0$, we use the b_1th keypair in the idx'_0th tree to sign the idxth message. At hypertree layer $0 < i < d$, we use the b_{i+1}th keypair in the idx'_ith tree to sign the root of the idx'_{i-1}th tree on layer $i - 1$.

3.5 Pyramid

Finally, we describe Pyramid using the previously introduced building blocks. We use the notation that we established in Section 3.4.3 for idx.

3.5.1 Pyramid key generation $((\text{SK}, \text{PK}) \leftarrow \text{kg}(1^n))$

Pyramid key generation is similar to the key generation of [HRS16]. We start by sampling a public parameter $\text{Seed} \xleftarrow{\$} \{0,1\}^n$. Next, we sample a secret value $\mathcal{R}$, or a secret reference $\mathcal{R}_i$ for every layer $0, \dots, d-1$:

$$\mathcal{R} = \begin{cases} \mathcal{R}_* \xleftarrow{\$} \{0,1\}^n & \text{Regular Pyramid} \\ (\mathcal{R}_{0,0}, \dots, \mathcal{R}_{d-1,0}), \mathcal{R}_{i,0} \xleftarrow{\$} \{0,1\}^n \text{ for } 0 \le i < d & \text{FS Pyramid} \end{cases}$$

We stress that using multiple seeds is a consequence of the forward-secure property; a regular Pyramid instance with $d > 1$ samples *one* n-bit string.

Let ADRS be an empty address space. We denote $\text{ADRS}[i, j]$ for the jth address space at the ith hypertree layer, and $\mathcal{C}[i, j] := (\text{Seed}, \text{ADRS}[i, j])$. Next, we perform:

$$\text{Root} \leftarrow \begin{cases} \text{GenRoot}(\mathcal{C}[d-1, 0], \mathcal{R}_{d-1,0}) & \text{Regular Pyramid} \\ \text{GenRoot}(\mathcal{C}[d-1, 0], \mathcal{R}_*) & \text{Forward-secure Pyramid} \end{cases}$$

We may now assemble the Pyramid public key: $\text{PK} = (\text{Root}, \text{Seed})$. For the secret key, we sample an additional secret value $\text{S}_{\text{PRF}} \xleftarrow{\$} \{0,1\}^n$. We may now assemble the Pyramid secret key: $\text{SK} = (\text{idx} = 0, \mathcal{R}, \text{S}_{\text{PRF}}, \text{PK})$.

3.5.2 Pyramid signature $((\Sigma, \mathrm{SK}') \leftarrow \mathrm{sign}(M, \mathrm{SK}))$

We are given a message $M \in \{0,1\}^*$ and secret key $\mathrm{SK} = (\mathrm{idx}, \mathcal{R}, \mathrm{S}_{\mathrm{PRF}}, \mathrm{PK})$. Let ADRS be an empty address space. We denote $\mathrm{ADRS}[i,j,k]$ for the kth keypair address space, located in the jth tree on hypertree layer i; let $\mathcal{C}[i,j,k] \coloneqq (\mathrm{Seed}, \mathrm{ADRS}[i,j,k])$. Let $MD_{\mathrm{idx}} = R_0$ be the compressed message, generated conform Section 3.4.2, in context $\mathcal{C}[0, \mathrm{idx}'_0, b_1]$. We can now start the iterative process of signing and computing authentication paths. We start in context $\mathcal{C}[i, \mathrm{idx}'_i, b_{i+1}]$. We compute:

$$(\mathrm{SK}, \mathrm{PK}) \leftarrow \begin{cases} \mathrm{WOTS.kg}(\mathcal{C}[i, \mathrm{idx}'_i, b_{i+1}], \mathcal{R}_*) & \text{Regular Pyramid} \\ \mathrm{WOTS.kg}(\mathcal{C}[i, \mathrm{idx}'_i, b_{i+1}], \mathcal{S}_{0,\mathrm{idx}}) & \text{Forward-secure Pyramid} \end{cases}$$

We compute $\mathcal{S}_{0,\mathrm{idx}}$ according to Section 3.4.2. We create a WOTS-TW signature $\sigma_i \leftarrow \mathrm{WOTS.sign}(R_i, \mathrm{SK}_i)$ and obtain a leaf node by compressing it per Section 3.4.2. We then compute the authentication path Auth_i of the leaf node in accordance with 3.4.3. Finally, we generate the root of the tree $\mathrm{Root}_i = R_{i+1}$, which forms the message that we sign on the $(i+1)$th hypertree layer. We iterate this process for hypertree layers $0 \leq i < d$, following Section 3.4.3. We generate the Pyramid signature $\Sigma = (\mathrm{idx}, r, \sigma_0, \mathrm{Auth}_0, \ldots, \sigma_{d-1}, \mathrm{Auth}_{d-1})$; r is the message randomization element that we use for the initial message hash MD_{idx}. Finally, we prepare SK for the next signature. For a forward-secure Pyramid instance, we advance the FSG references. Note that $\mathrm{idx}_{-1} = 0$. Starting from hypertree layer $i = 0$, as long as $\mathrm{idx}_{i-1} = 2^{ih/d} - 1$ holds and $i < d$, we obtain a new seed $\mathcal{R}_{i,\mathrm{idx}'_{i-1}+1}$ for layer i using FSG. Intuitively, this is akin to an addition $\mathrm{idx} + 1$; we update seeds up to- and inluding the b_i where the carry lands. We now update $\mathrm{idx} = \mathrm{idx} + 1$ and return (Σ, SK').

3.5.3 Pyramid verification $(b \leftarrow \mathrm{vf}(M', \Sigma, \mathrm{PK}))$

We are given a message $M' \in \{0,1\}^*$, a public key $\mathrm{PK} = (\mathrm{Root}, \mathrm{Seed})$, and a signature Σ. Using r from Σ, we compute a compressed message MD'_{idx} and interpret idx in base-$2^{h/d}$ words. We can now start the iterative process of regenerating verification keys and calculating/verifying tree roots. We perform WOTS.vf for the b_{i+1}th node in the idx'_ith tree on hypertree layer i. We perform WOTS-TW verification either on a compressed message MD'_{idx} $(i = 0)$ or a tree root Root'_{i-1} $(i > 0)$. This grants a WOTS-TW public key pk'_i, which we then compress. We use the Merkle proof Auth_i and the compressed public key to combine until we obtain the root Root'_i. We iterate this process for hypertree layers $0 \leq i < d$. Finally, we return 1 if $\mathrm{PK.Root} \stackrel{?}{=} \mathrm{Root}'_{d-1}$ and 0 otherwise.

3.6 Pyramid proof status

In this document, we do not give a formal proof for the security of the Pyramid signature scheme. Instead, we will briefly discuss the state of affairs for such a proof. Section 3.6.1 includes preliminaries for the summary in Section 3.6.2. Finally, we describe the implications of the structure of the proof of security for SPHINCS$^+$, for Pyramid, in Section 3.6.3.

3.6.1 Preliminaries

We repeat the standard DSS security notion called existential unforgeability under adaptive chosen message attacks (EU-CMA). We use the definition of the EU-CMA experiment that is shown in [Hül13a]. For Experiment 3.6.1 (and 3.6.2), we consider a digital signature scheme $\mathrm{Dss} = (\mathrm{Kg}, \mathrm{Sign}, \mathrm{Vf})$, an adversary $\mathcal{A}$, and a security parameter n.

Experiment 3.6.1 ($\mathrm{Exp}^{\mathrm{EU-CMA}}_{\mathrm{Dss}(1^n)}(\mathcal{A})$)**.**
$(\mathrm{sk}, \mathrm{pk}) \leftarrow \mathrm{Kg}(1^n)$.
$(M', \sigma') \leftarrow \mathcal{A}^{\mathrm{Sign}(\mathrm{sk},\cdot)}(\mathrm{pk})$.
Let $\{(M_i, \sigma_i)\}_{i=1}^q$ *be the query-answer pairs of* $\mathrm{Sign}(\mathrm{sk}, \cdot)$.
Return 1 iff $\mathrm{Vf}(\mathrm{pk}, M', \sigma') \stackrel{?}{=} 1$ *and* $M' \notin \{M_i\}_{i=1}^q$.

A signature scheme is said to be EU-CMA-secure if any adversary $\mathcal{A}$, running in time polynomial in the security parameter n, has negligible success probability.

In [HK21], a related DSS security notion is considered, called EU-naCMA. "na" stands for "non-adaptive", referring to the way in which the adversary $\mathcal{A}$ is forced to make its queries. Instead of being allowed to perform adaptive queries like in Experiment 3.6.1, the adversary $\mathcal{A}$ starts by specifying all messages that it wants to query. Afterwards, $\mathcal{A}$ is granted the respective signatures for the queries, along with pk. The EU-naCMA experiment from [HK21] is shown in Experiment 3.6.2.

Experiment 3.6.2 ($\mathrm{Exp}^{\mathrm{EU-naCMA}}_{\mathrm{Dss}(1^n)}(\mathcal{A})$)**.**
$(\mathrm{sk}, \mathrm{pk}) \leftarrow \mathrm{Kg}(1^n)$
$\{M_1, \ldots, M_q\} \leftarrow \mathcal{A}()$.
Compute $\{(M_i, \sigma_i)\}_{i=1}^q$ *using* $\mathrm{Sign}(\mathrm{sk}, \cdot)$.
$(M', \sigma') \leftarrow \mathcal{A}(\{(M_i, \sigma_i)\}_{i=1}^q, \mathrm{pk})$
Return 1 iff $\mathrm{Vf}(\mathrm{pk}, M', \sigma') \stackrel{?}{=} 1$ *and* $M' \notin \{M_i\}_{i=1}^q$.

3.6.2 Partial SPHINCS$^+$ proof summary

First, we will try to give some insight into the difference between WOTS and WOTS-TW. WOTS-TWs EU-naCMA insecurity is bounded by the insecurity of PRF and Th for a number of properties.

The properties for tweakable hash functions share a prefix "SM", meaning "single function, multi-target". In short, "multi-target" limits the adversary $\mathcal{A}$'s success event in these properties to instances in which $\mathcal{A}$ only queried with distinct tweaks T. The prefix "single-function" limits $\mathcal{A}$'s success event in these properties to instances in which $\mathcal{A}$ achieves success for a specific public parameter Seed. Regardless of a Th instantiation achieving these properties, we need to enforce the assumptions in these properties, in WOTS-TW, to "apply" for them.

In the case SPHINCS$^+$ (and conceivably that of Pyramid), this line of reasoning is extended to (1) multiple WOTS-TW instances and (2) the structure that efficiently connects those WOTS-TW instances. For Pyramid, we would like to preserve the approach of SPHINCS$^+$: enforce unique tweaks, *even* between functions that are of a different purpose within the hypertree structure. This paves the way for a bound solely in terms of Th, instead of F, H, and Th_λ seperately.

The WOTS-TW proof is extended to multiple instances. One uses tweak (prefix) uniqueness between all WOTS-TW instances to make a similar argument to that of single instance WOTS-TW. Finally, one uses the WOTS-TW proof in the proof of security for SPHINCS$^+$, resulting in a modular proof.

3.6.3 Pyramid proof implication

As stated in [HK21], the strategy of proving EU-naCMA for an OTS like WOTS-TW, instead of a property like EU-CMA, is not trivial to reproduce in the case of stateful signature schemes. Intuitively, this is because the EU-naCMA model does not encompass both usage scenarios of WOTS-TW in stateful schemes. The OTS at the bottom layer in schemes like Pyramid sign the output of (randomized) message hash function calls, which may be influenced by an adversary. Therefore, an adversary should be able to base his message query on the public key at the bottom layer. If the reduction is not able to base its public key on the message query like in EU-naCMA, then it needs to *guess* the positions of signature elements in the chain, to plant the same challenges that it does currently. This *guessing* is an important consideration for WOTS$^+$ [Hül13b], which introduces randomization elements to insert (PRE) challenges. This mode "enables the tight security proof without requiring the used hash function family to be collision-resistant".

Chapter 4

Pyramid instantiations

In this Chapter, we will propose instantiations for the functions that we require in Section 3.4. The instantiations are largely based on two considerations. We first consider constructions that were used previously in stateful schemes, which are included in RFC 8391 [HBG+18]. The second consideration is alignment with the verification procedure of SPHINCS$^+$. By keeping the verification subroutines of Pyramid similar to those of SPHINCS$^+$, verification interoperability may be preserved without a great amount of additional code. The functions that are relevant for the verification routine are $\mathrm{H}_{\mathrm{msg}}$ and Th_*. Alignment between the two is manifested in both the choices of hash functions and the input constructions that are used for these functions. Furthermore, we adapt to the inclusion of PK.Seed in PRF by SPHINCS$^+$.

For a sponge-based construction F, we will denote $F(M, d)$ as the application of F on M to obtain an output of d bits. Details on the Haraka sponge instantiations are found in the SPHINCS$^+$ draft [ABB+20b].

In previous schemes, the forward-secure pseudorandom generator received a byte $b \in \{0, 1\}$ as input. In Pyramid, we incorporate this byte into the ADRS input, called the *direction*. We denote Seed for a temporal secret value (not in SK), generated by FSG via direction $= 0$. We denote $\mathrm{SK.Seed}_i$ for a temporary secret reference (in SK at some stage), generated by FSG via direction $= 1$. We generate R using PRF. n and m are the Pyramid security parameter and the message digest length, in bytes.

4.1 SHAKE256 instantiations

For Pyramid-SHAKE256, we arrange ADRS at byte-position n. We shorten "SHAKE256" to "SHAKE" purely for formatting purposes. For regular

Pyramid-SHAKE256 we define:

$$\begin{aligned}
\mathrm{H_{msg}}(\mathrm{PK.Root}, \mathrm{ADRS}, R, M) &= \mathrm{SHAKE}(R \| \mathrm{ADRS} \| \mathrm{PK.Root} \| M, 8m) \\
\mathrm{PRF}(\mathrm{SK.PRF}, \mathrm{PK.Seed}, \mathrm{ADRS}) &= \mathrm{SHAKE}(\mathrm{SK.PRF} \| \mathrm{ADRS} \| \mathrm{PK.Seed}, 8n) \\
\mathrm{PRF_{kg}}(\mathrm{SK.Seed}, \mathrm{PK.Seed}, \mathrm{ADRS}) &= \mathrm{SHAKE}(\mathrm{SK.Seed} \| \mathrm{ADRS} \| \mathrm{PK.Seed}, 8n)
\end{aligned}$$

For forward-secure Pyramid-SHAKE256 we define:

$$\begin{aligned}
\mathrm{PRF_{kg}}(\mathrm{Seed}, \mathrm{ADRS}) &= \mathrm{SHAKE}(\mathrm{Seed} \| \mathrm{ADRS}, 8n) \\
\mathrm{FSG}(\mathrm{SK.Seed}_i, \mathrm{PK.Seed}, \mathrm{ADRS}) &= \mathrm{SHAKE}(\mathrm{SK.Seed}_i \| \mathrm{ADRS} \| \mathrm{PK.Seed}, 8n)
\end{aligned}$$

The tweakable hash functions are defined *exactly* like those of SPHINCS$^+$. The robust instantiations are as follows:

$$\begin{aligned}
\mathrm{F}(\mathrm{PK.Seed}, \mathrm{ADRS}, M) &= \mathrm{SHAKE256}(\mathrm{PK.Seed} \| \mathrm{ADRS} \| M^{\oplus}, 8n) \\
\mathrm{H}(\mathrm{PK.Seed}, \mathrm{ADRS}, M_1 \| M_2) &= \mathrm{SHAKE256}(\mathrm{PK.Seed} \| \mathrm{ADRS} \| M_1^{\oplus} \| M_2^{\oplus}, 8n) \\
\mathrm{Th}_{\lambda}(\mathrm{PK.Seed}, \mathrm{ADRS}, M) &= \mathrm{SHAKE256}(\mathrm{PK.Seed} \| \mathrm{ADRS} \| M^{\oplus}, 8n)
\end{aligned}$$

The simple instantiations are as follows:

$$\begin{aligned}
\mathrm{F}(\mathrm{PK.Seed}, \mathrm{ADRS}, M) &= \mathrm{SHAKE256}(\mathrm{PK.Seed} \| \mathrm{ADRS} \| M, 8n) \\
\mathrm{H}(\mathrm{PK.Seed}, \mathrm{ADRS}, M_1 \| M_2) &= \mathrm{SHAKE256}(\mathrm{PK.Seed} \| \mathrm{ADRS} \| M_1 \| M_2, 8n) \\
\mathrm{Th}_{\lambda}(\mathrm{PK.Seed}, \mathrm{ADRS}, M) &= \mathrm{SHAKE256}(\mathrm{PK.Seed} \| \mathrm{ADRS} \| M, 8n)
\end{aligned}$$

We generate the bitmasks for the robust instantiations of tweakable hash functions following SPHINCS$^+$:

$$M^{\oplus} = M \oplus \mathrm{SHAKE256}(\mathrm{PK.Seed} \| \mathrm{ADRS}, l) \text{ for } M \in \{0,1\}^l$$

4.2 SHA256 instantiations

First, we will set $\mathrm{S_{PK.Seed}} \coloneqq \mathrm{PK.Seed} \| \mathrm{toByte}(0, 64-n)$. We always take the first $8n$ bits of SHA256 output and discard the rest. For Pyramid-SHA256, we arrange $\mathrm{ADRS}^{\mathcal{C}}$ at byte-position 64. For regular Pyramid-SHA256 we define:

$$\begin{aligned}
&\mathrm{H_{msg}}(\mathrm{PK.Root}, \mathrm{ADRS}, R, M) \\
&\qquad = \mathrm{SHA256}(R \| \mathrm{PK.Root} \| \mathrm{toByte}(0, 64 - 2n) \| \mathrm{ADRS}^{\mathcal{C}} \| M)
\end{aligned}$$

$$\begin{aligned}
\mathrm{PRF}(\mathrm{SK.PRF}, \mathrm{PK.Seed}, \mathrm{ADRS}) &= \mathrm{SHA256}(\mathrm{S_{PK.Seed}} \| \mathrm{ADRS}^{\mathcal{C}} \| \mathrm{SK.PRF}) \\
\mathrm{PRF_{kg}}(\mathrm{SK.Seed}, \mathrm{PK.Seed}, \mathrm{ADRS}) &= \mathrm{SHA256}(\mathrm{S_{PK.Seed}} \| \mathrm{ADRS}^{\mathcal{C}} \| \mathrm{SK.Seed})
\end{aligned}$$

For forward-secure Pyramid-SHA256 we define:

$$\mathrm{PRF_{kg}}(\mathrm{Seed}, \mathrm{ADRS}) = \mathrm{SHA256}(\mathrm{Seed}\|\mathrm{ADRS}^{\mathcal{C}})$$
$$\mathrm{FSG}(\mathrm{SK.Seed}_i, \mathrm{PK.Seed}, \mathrm{ADRS}) = \mathrm{SHA256}(\mathrm{S_{PK.Seed}}\|\mathrm{ADRS}^{\mathcal{C}}\|\mathrm{SK.Seed}_i)$$

The tweakable hash functions are defined *exactly* like those of SPHINCS$^+$. The robust instantiations are as follows:

$$\mathrm{F}(\mathrm{S_{PK.Seed}}, \mathrm{ADRS}^{\mathcal{C}}, M) = \mathrm{SHA256}(\mathrm{S_{PK.Seed}}\|\mathrm{ADRS}^{\mathcal{C}}\|M^{\oplus})$$
$$\mathrm{H}(\mathrm{S_{PK.Seed}}, \mathrm{ADRS}^{\mathcal{C}}, M_1\|M_2) = \mathrm{SHA256}(\mathrm{S_{PK.Seed}}\|\mathrm{ADRS}^{\mathcal{C}}\|M_1^{\oplus}\|M_2^{\oplus})$$
$$\mathrm{Th}_\lambda(\mathrm{S_{PK.Seed}}, \mathrm{ADRS}^{\mathcal{C}}, M) = \mathrm{SHA256}(\mathrm{S_{PK.Seed}}\|\mathrm{ADRS}^{\mathcal{C}}\|M^{\oplus})$$

The simple instantiations are as follows:

$$\mathrm{F}(\mathrm{S_{PK.Seed}}, \mathrm{ADRS}^{\mathcal{C}}, M) = \mathrm{SHA256}(\mathrm{S_{PK.Seed}}\|\mathrm{ADRS}^{\mathcal{C}}\|M)$$
$$\mathrm{H}(\mathrm{S_{PK.Seed}}, \mathrm{ADRS}^{\mathcal{C}}, M_1\|M_2) = \mathrm{SHA256}(\mathrm{S_{PK.Seed}}\|\mathrm{ADRS}^{\mathcal{C}}\|M_1\|M_2)$$
$$\mathrm{Th}_\lambda(\mathrm{S_{PK.Seed}}, \mathrm{ADRS}^{\mathcal{C}}, M) = \mathrm{SHA256}(\mathrm{S_{PK.Seed}}\|\mathrm{ADRS}^{\mathcal{C}}\|M)$$

We generate the bitmasks for the robust instantiations of tweakable hash functions following SPHINCS$^+$:

$$M^{\oplus} = M \oplus \text{MGF1-SHA256}(\mathrm{PK.Seed}\|\mathrm{ADRS}^{\mathcal{C}}, l) \text{ for } M \in \{0,1\}^l$$

The SHA256 instantiations are the result of a number of considerations.

Intermediate state We use the intermediate SHA256 state $\mathrm{S_{PK.Seed}}$. SPHINCS$^+$ also uses this optimization. Instead of adding an n-bit string to functions that require PK.Seed, the function starts from a state that only needs to be computed once, without having to process PK.Seed thereafter.

We currently start from $\mathrm{S_{PK.Seed}}$ in functions that require PK.Seed. However, a future Pyramid proposal could consider doing so for every SHA256 instantiation, as it does not hurt security and could reduce code size. We have applied this simplification in Chapter 5.

PRF construction Pyramid defines the PRF instantiations PRF and $\mathrm{PRF_{kg}}$. Inputs are of a fixed length and the functions are keyed on the chaining value, allowing for a relatively simple PRF construction compared to HMAC-SHA256. RFC 8391 also employs this PRF construction. More detail can be found in [Hül13a]. Note that constructions that use two keys, like PRF, do not apply for this proof to preserve speed. This is because we do not pad and compress the secondary key, i.e. SK.Seed, before compressing the input.

We explicitly note that we do not enforce the padding for SHA256 in the instantiations of the other constructions (RFC 8391).

PRG construction The PRG construction that we use to achieve forward-secure Pyramid can be found in [HRB13] [Hül13a]. However, like PRF, the current instantiation does not perform the padding step from the aforementioned PRF constructions for fixed-length MD hash functions for the secondary key. FSG is not proven to be a (forward-secure) PRG when using SHA256. The benefit of the construction is that we only process a single SHA256 input block for every FSG call.

A perk of the compressed SHA256 address in SPHINCS$^+$ is that it solves a previous concern for the PRG construction in stateful schemes. An example of this is the PRG construction that is implemented in [HKS20]. The PRG construction uses a byte $b \in \{0,1\}$ to diversify inputs for the generation of OTS seeds and FSG references. That is, the byte b would sometimes require an entire additional block to be processed if one includes ADRS in its entirety. The solution would be to include only address parts that enforce uniqueness: the tree layer/index, the keypair index, and a value to ensure the independence of the different function families.

4.3 Haraka instantiations

Haraka is a dedicated short-input hash function. SPHINCS$^+$ includes Haraka to give a "demonstration" of the possible speedup that such a function may provide. This section proposes instantiations that closely resemble those of SPHINCS$^+$. We always take the first $8n$ bits of Haraka256/Haraka512 output and discard the rest.

For Pyramid-Haraka, we arrange ADRS at byte-position 0 for HarakaS, Haraka256, and Haraka512. For regular Pyramid-Haraka we define:

$$\begin{aligned}
\mathrm{H}_{\mathrm{msg}}(\mathrm{PK.Root}, \mathrm{ADRS}, R, M) &= \mathrm{HarakaS}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| R \| \mathrm{PK.Root} \| M, 8m) \\
\mathrm{PRF}(\mathrm{SK.PRF}, \mathrm{PK.Seed}, \mathrm{ADRS}) &= \mathrm{Haraka512}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| \mathrm{SK.PRF}, 8n) \\
\mathrm{PRF}_{\mathrm{kg}}(\mathrm{SK.Seed}, \mathrm{PK.Seed}, \mathrm{ADRS}) &= \mathrm{Haraka512}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| \mathrm{SK.Seed}, 8n)
\end{aligned}$$

For forward-secure Pyramid-Haraka we define:

$$\begin{aligned}
\mathrm{PRF}_{\mathrm{kg}}(\mathrm{Seed}, \mathrm{ADRS}) &= \mathrm{Haraka256}_{\mathrm{Seed}}(\mathrm{ADRS}) \\
\mathrm{FSG}(\mathrm{SK.Seed}_i, \mathrm{PK.Seed}, \mathrm{ADRS}) &= \mathrm{Haraka512}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| \mathrm{SK.Seed}_i, 8n)
\end{aligned}$$

The tweakable hash functions are defined *exactly* like those of SPHINCS$^+$. The robust instantiations are as follows:

$$\begin{aligned}
\mathrm{F}(\mathrm{PK.Seed}, \mathrm{ADRS}, M) &= \mathrm{Haraka512}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| M^{\oplus}) \\
\mathrm{H}(\mathrm{PK.Seed}, \mathrm{ADRS}, M_1 \| M_2) &= \mathrm{HarakaS}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| M_1^{\oplus} \| M_2^{\oplus}, 8n) \\
\mathrm{Th}_{\lambda}(\mathrm{PK.Seed}, \mathrm{ADRS}, M) &= \mathrm{HarakaS}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| M^{\oplus}, 8n)
\end{aligned}$$

The simple instantiations are as follows:

$$\begin{aligned}
\mathrm{F}(\mathrm{PK.Seed}, \mathrm{ADRS}, M) &= \mathrm{Haraka512}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| M) \\
\mathrm{H}(\mathrm{PK.Seed}, \mathrm{ADRS}, M_1 \| M_2) &= \mathrm{HarakaS}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| M_1 \| M_2, 8n) \\
\mathrm{Th}_\lambda(\mathrm{PK.Seed}, \mathrm{ADRS}, M) &= \mathrm{HarakaS}_{\mathrm{PK.Seed}}(\mathrm{ADRS} \| M, 8n)
\end{aligned}$$

We generate the bitmasks for the robust instantiations of tweakable hash functions following SPHINCS$^+$:

$$\begin{aligned}
\text{For F: } M^\oplus &= M \oplus \mathrm{Haraka256}_{\mathrm{PK.Seed}}(\mathrm{ADRS}) \text{ for } M \in \{0,1\}^{8n} \\
\text{else: } M^\oplus &= M \oplus \mathrm{HarakaS}_{\mathrm{PK.Seed}}(\mathrm{ADRS}, l) \text{ for } M \in \{0,1\}^{8l}
\end{aligned}$$

PRG construction An initial position of concern for our Haraka instantiations was the FSG. The problem with ADRS was that, for $n = 32$, the one-byte value $b \in \{0, 1\}$ is pushed to a third input block of the sponge. The FSG construction uses a byte $b \in \{0, 1\}$ to diversify inputs for the generation of OTS seeds and FSG references. The one-byte value prohibits usage of $\mathrm{Haraka512}_{\mathrm{PK.Seed}}$. We provide a solution by including b in the addressing scheme. Compressing the address like SHA256 is not an option, because this requires changes from SPHINCS$^+$.

Round constants We copy the construction of SPHINCS$^+$ for inclusion of PK.Seed in the HarakaS round constants. The PRF constructions that result from this should be considered "experimental".

The forward-secure construction for $\mathrm{PRF}_{\mathrm{kg}}$ requires recalculation of the round constants when switching Seeds. This may happen repeatedly while calculating a keypair/signature.

The regular construction for PRF requires invocation of $\mathrm{Haraka512}_{\mathrm{PK.Seed}}$. This is because we only use PRF once per signature, for the calculation of the message randomization element. A regular Pyramid implementation may therefore want to leave out code for $\mathrm{Haraka256}_{\mathrm{Seed}}$, *if* one decides to use the above construction for PRF.

The PRF instantiation comes at the cost of a slower PRF call. Hence, the following (perhaps counterintuitive) construction could be considered:
$\mathrm{PRF}_{\mathrm{kg}}(\mathrm{Seed}, \mathrm{ADRS}) = \mathrm{Haraka256}_{\mathrm{SK.PRF}}(\mathrm{ADRS})$

4.4 Discussion

Finally, we will provide some generic discussion that applies to all instantiations.

4.4.1 Function family independence

When comparing Pyramid instantiations to RFC 8391, perhaps the most striking difference is the lack of the unique value field that RFC 8391 uses to achieve independence of the different function families. An example is as follows: $\mathrm{H} : \mathrm{SHAKE256}(\mathrm{toByte}(1, 64)||\dots)$. In SPHINCS$^+$, the method of achieving independence is by ensuring that ADRS is unique, while appending ADRS at a fixed position within input buffers.

Because we would like to attain verification interoperability between SPHINCS$^+$ and Pyramid, the method of achieving independence of the different function families is tied to that of SPHINCS$^+$. We note that this is likely for the better; constructions that use both a precomputed state *and* such values become somewhat chaotic. Furthermore, including these values in the short-input hash function buffers requires additional changes as well.

4.4.2 Address lengths

We previously noted that FSG could compress ADRS to shorten its input length. This applies to $\mathrm{H}_{\mathrm{msg}}$ as well. $\mathrm{H}_{\mathrm{msg}}$ implements what RFC 8391 refers to as "index randomized hashing" [HBG$^+$18]. The $\mathrm{H}_{\mathrm{msg}}$ construction is also featured in [BHRvV21]; our approach should therefore also apply for the optimization that is shown in RapidXMSS. The combination of ADRS fields tree index and keypair index correspond to one WOTS-TW instance at the bottom layer. The index construction of RFC 8391 is at least as short. However, the independence of the different function families must again be taken into account.

4.4.3 Forward-secure instantiations

For forward-secure Pyramid instances, FSG generates seeds Seed. $\mathrm{PRF}_{\mathrm{kg}}$ expands Seed into WOTS-TW secret-key values. Because FSG incorporates PK.Seed into the calculation of Seed, $\mathrm{PRF}_{\mathrm{kg}}$ should not require PK.Seed for multi-target protection, in contrast to the regular Pyramid $\mathrm{PRF}_{\mathrm{kg}}$. The $\mathrm{PRF}_{\mathrm{kg}}$ for regular Pyramid is derived from a modification to XMSS in response to a comment by "TC CYBER WG QSC" in [oST20b].

4.4.4 OptRand

In SPHINCS$^+$, calculation of the hash randomization value R is as follows: $R = \mathrm{PRF}(\mathrm{SK.prf}, \texttt{OptRand}, M)$. `OptRand` is an n-byte value that allows one to make signing nondeterministic. The reason for including `OptRand` in SPHINCS$^+$ is to counteract side-channel attacks, which would "collect several traces for the exact same computation by asking for a signature on the same message multiple times." [ABB$^+$20b]. The attack in question does not apply to Pyramid, because the signature index is either unique or

invalid for every signature. For this reason, we do not currently incorporate `OptRand` in PRF.

4.4.5 XOF WOTS-TW secret generation

We left the discussion open on the topic of starting all SHA256 computations from the intermediate state. We do the same for the efficient generation of WOTS-TW secrets using an XOF. That is, given a function that can produce an output of arbitrary length, the following construction may be of interest:

$$(\mathcal{R}_i \| \mathrm{sk}_1 \| \dots \| \mathrm{sk}_l) = \mathrm{XOF}(\mathrm{Seed} \| \mathrm{ADRS} \| \mathcal{R}_{i-1}, (l+1) \cdot 8n)$$

Generating $\mathcal{R}_i$ first allows forwarding references at the same rate that we do currently, by not squeezing for sk_i. Alternatively, one may consider a straightforward alternative, which expands sk from $\mathcal{S}_i$ conform WOTS-TW:

$$(\mathcal{R}_i \| \mathcal{S}_i) = \mathrm{XOF}(\mathrm{Seed} \| \mathrm{ADRS} \| \mathcal{R}_{i-1}, 16n)$$

The first construction refrains from setting the chain address or the direction in ADRS; the second one refrains from setting the direction in ADRS. The FSG type field should be enforced.

4.4.6 Non-repudiation

We refrain from specifying parameter sets. However, we include a note that we exemplify using $\mathrm{H}_{\mathrm{msg}}$ for the case that $n = 16$. Note that RFC 8391 does not support this value for n, neither for SHA2 nor SHAKE. SPHINCS$^+$ does include example parameter sets in which $n = 16$.

As noted in errata EID 6024, incorporated in RFC 8391 [HBG$^+$18]: "SHAKE with an internal state of n bits and an output length of n bits achieves n/2 bit security against classical preimage, second-preimage and collision attacks". Therefore, when $n = 16$, a signer is able to find a collision in $\mathrm{H}_{\mathrm{msg}}$ in 2^{64} work in the classical setting. When we sign colliding messages using the same bottom OTS keypair, these lead to identical Pyramid signatures. In SPHINCS$^+$, this is less of a concern, because the FTS instance that one signs a message digest with is based on the residual message digest produced by $\mathrm{H}_{\mathrm{msg}}$. Therefore, finding a collision for $\mathrm{H}_{\mathrm{msg}}$ that results in identical SPHINCS$^+$ signatures, requires additional work compared to Pyramid.

4.5 Addressing scheme

The instantiations of hash functions consistently arrange ADRS at a specific location in input buffers. When ADRS is built correctly, this forces an attacker to target a specific function instantiation, at a specific "location"

within the hypertree. We will now describe the proposed addressing scheme for Pyramid.

<table>
<tr><td rowspan="3">layer address</td><td rowspan="7">tree address</td><td>F</td><td rowspan="2">key pair address</td><td>chain address</td><td>hash address</td></tr>
<tr><td>Th_λ</td><td colspan="2">-</td></tr>
<tr><td>H</td><td>-</td><td>tree height</td><td>tree index</td></tr>
<tr><td rowspan="2">0</td><td>H_{msg}</td><td rowspan="4">key pair address</td><td colspan="2" rowspan="2">-</td></tr>
<tr><td>PRF</td></tr>
<tr><td rowspan="2">layer address</td><td>PRF_{keygen}</td><td>chain address</td><td>0</td></tr>
<tr><td>FSG</td><td>direction</td><td>-</td></tr>
<tr><td>1</td><td>3</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

Figure 4.1: The Pyramid address scheme.

In Figure 4.1 we portray the Pyramid addressing scheme. The numbers at the bottom specify the number of 32-bit words that are allocated to each of the address fields. Note that addressing is byte-oriented; 32-bit words are however useful to describe integer limits for implementations. The address scheme depicted in Figure 4.1 is derived from- and compatible with that of SPHINCS$^+$. The upper three address spaces for F, H, and Th_λ are shared with SPHINCS$^+$. The other four are an extension of these. Please note that `-` and `0` both characterize zero-padding. We write the value `0` when this is intuitive in light of the other address spaces.

We have verified that the addressing scheme from SPHINCS$^+$ is compatible with that of RFC 8391 [HBG$^+$18]. The only notable difference between the two is that RFC 8391 allocates two 32-bit fields for a tree address. RFC 8391 refers to the remaining 32-bit field as “keyAndMask” and uses the field to diversify one address for the generation of both keys- and bitmasks. The robust instantiation of tweakable hash functions in SPHINCS$^+$ only processes one (non-bitmasked) address per tweakable hash function call, making the field obsolete for SPHINCS$^+$ and Pyramid. In conclusion, we can attain similar parameter sets to RFC 8391, while maintaining compatibility with SPHINCS$^+$ by using three 32-bit fields for a tree address. We forgo defining concrete parameter sets due to the lack of a formal proof.

4.6 Key format

We include the format of the Pyramid signature, public key, and secret key in Appendix A.1. These are the exact same as the formats that are used in RFC 8391 [HBG$^+$18].

Chapter 5

Pyramid implementations

As part of the Pyramid proposal, we provide a total of four C implementations, which aim to give insight into how they perform- and what they look like when instantiated. We will implement two of these in Jasmin, a framework for developing high-speed and high-assurance cryptographic software. The C implementations will be used to test the Jasmin implementations for functional correctness. We note that the C- and Jasmin implementations should be *treated with care*. Stateful schemes use complex algorithms and generalizations. At a *bare* minimum, a formal proof for the security of Pyramid, a proof of functional correctness, and a second, independent implementation, are obligatory for achieving trust in the implementations. We refer to the implementations as follows; we also provide references to similar implementations that they are inspired by:

	Regular	**Forward-Secure**
Naive	NFS-Naive [HRCW21]	FS-StackRestore
Algorithmic	NFS-Simple [KT21a]	FS-Simple [HKS20]

The implementations directly use the SPHINCS$^+$ and XMSS reference code from [RWW$^+$21] and [HRCW21]. The Simple implementations use the code located at [KT21b], created by the authors of [KT21a]. We leave complex parts of their implementation, such as scheduling, untouched. We solely shift their implementation from XMSS to Pyramid, and adapt the tree-traversal algorithms BDS/MMT to the use of FSG for forward-secure implementations. BDS and MMT are the tree-traversal algorithms that Simple uses; we explain these in Section 5.4.

5.1 Common

The four implementations have a great number of features in common. Files that implement such features are contained in `common/`. We will describe

notable ones in the following list, before moving to the specific implementations.

Pseudorandom function In Chapter 4 we make a distinction between the functions PRF and $\mathrm{PRF}_{\mathrm{kg}}$. The distinction shows that $\mathrm{PRF}_{\mathrm{kg}}$ does not always *require* a PK.Seed argument, and highlights usage of different keys between the two functions. In the reference implementation, we choose not to make the distinction; we implement one function `prf` that takes an argument `pub_seed`. The function `prf` copies `pub_seed` into the input buffer for the hash function unconditionally. However, we guarantee unique addresses between `prf` purposes, by letting its caller set the type field of its `addr` argument. This modification aligns with the proposed $\mathrm{SPHINCS}^+$ PRF implementation and simplifies our implementation a great amount[1].

Hash function initialization As mentioned in Sections 4.2 and 4.3, the SHA256- and Haraka implementations perform preprocessing, either by precomputing an intermediate state or by tweaking round constants. This is realized by the function `initialize_hash_function`, in the exact same manner as shown in [RWW+21]. Taking SHA256 as an example, functions in the files `hash_sha256.c` and `thash_sha256_*.c` utilize the precomputed state.

Because we resort to one PRF implementation, there is no reason to perform `tweak_constants` with a secret input; therefore Haraka256 remains unused in this function.

Forward-secure generator The function FSG is currently the only hash-function instantiation that does not require a fully completed address `addr`. Instead, we copy the relevant fields directly into the buffer `buf`, by treating it as a `uint32_t[8]`. The caller's address remains unmodified. The function signature of `fsg` is inspired by that of `hash_prg` in [HKS20] and `FsGen` in literature.

SHA256 hashing We define the `h_msg` function for SHA256 in `hash_sha256.c` in terms of `shaX`, which expands to `sha512` when `PYR_N` is `32`. This is a feature that is currently included in the $\mathrm{SPHINCS}^+$ repository. We preserve the construction, but note that it is not part of the proposal for $\mathrm{H}_{\mathrm{msg}}$ in Pyramid.

We also point out that `ADDR_POS_256_PRE` is the byte-position of the compressed address *having already processed* the intermediate state. We only preserve SHA512 for `h_msg`, which does not use PK.Seed. Therefore we do not define the macro for SHA512.

[1] We use a macro `FORWARD_SECURE`. This is not available in `common/` due to the project structure.

Addressing scheme The implementation maintains the basic addressing scheme from SPHINCS$^+$, located in `address.c`. We enlarge the the maximum subtree size from 2^{16} to 2^{24}, which was previously limited by the 2-byte maximum of `set_keypair_addr`.

We extend `address.c` with adapted getter/setters that are used by the Simple algorithm, along with utility functions. The file `address.h` contains the four additional types that Pyramid defines over SPHINCS$^+$.

Root computation The file `merkle.c` contains a generic implementation for `compute_root`. The verification routine of Pyramid does not differ between implementations. Hence, we include it in `common/`. We intentionally chose to include the routine that permits offset- and height arguments. While not strictly necessary for verification, this allows external signature algorithms to reuse the implementation to compute roots at custom heights.

We do not share the verification routine between implementations, but it is currently the same for all implementations. This way, we keep the possibility of reverting to a less flexible `compute_root` function within the individual implementations open.

5.2 External algorithm & format

We introduce two files that aim to improve the structure of the implementations, called `format.*` and `ext_algo.*`.

External algorithm The implementations that use Simple, dubbed an "external algorithm", use a file called `ext_algo.h`. Its usage is twofold.

First, `ext_algo.h` defines the functions that `sign.c` requires to finalize a secret key, finalize a signature, and update the state. It is up to the external algorithm to implement these routines. In Appendix A.4, we show the simplicity of the `crypto_sign_signature` skeleton from FS-SIMPLE as a result of these changes.

Traditionally, `ext_finalize_sig` and `ext_update_state` are merged into a single function. The two are not intertwined and have different purposes. We separate them to make the code more manageable. This also allows us to guard an `sk` update behind a check for the last signature. Finally, we perform `forget_seeds` when the external algorithm is done, which forwards the reference for FSG. Note that the external algorithm could still somehow erroneously cache an expired reference; we merely forward the references, when appropriate, inside the core of the secret key.

In `ext_algo.h`, the external algorithm declares whether it is forward-secure, the number of bytes that its state occupies, and a bound on the maximum index that it can process. This bound is a safeguard for implementations

that may not want to process a tree that crosses the 64-bit index boundary (which should never be reached in practice).

While using `format.h` is convenient, we will note that this is not a recommended approach for a final Pyramid implementation: The file `api.h` includes `format.h`, thereby exposing it. A final implementation (which commits to one external algorithm) should set all values for `api.h` manually.

Format The file `format.h` defines the format of the public- and secret key, the signature, and bounds concerning the aforementioned. Note that the size of the secret key depends on the external algorithm being forward-secure. We admit that this is counterintuitive. Future work could consider including "FS" or "REG" into the parameter string of the `Makefile`.

The file `format.h` defines `LAST_VALID_IDX`. To enumerate all valid indices, we require $\lceil \frac{h}{8} \rceil$ bytes in the signature/secret key. When $8 \nmid h$, the 2^hth secret key update spills onto the leftover bits, which tells us that the secret key is invalid. Then, deletion of the key occurs, which ensures that the "spillage" bits remain set. When $8 \mid h$, we can either prepend a byte to detect spillage or end one iteration in advance. We do the latter for simplicity, leading to a single definition of `PYR_IDX_BYTES`. A potential critique of this approach is that there now exists an index, for which we do not produce a signature, yet the signature itself is verifiable in the implementation.

In `format.c` we implement basic Pyramid functions that are consistent, regardless of an external algorithm. The function `forget_seeds` has commonalities with big integer addition.

5.3 Stepping stone implementations

5.3.1 NFS-Naive

Our first implementation is NFS-Naive, which shows a basic implementation of Pyramid without additional optimizations, nor does it include a way of achieving forward security. The implementation is kept relatively straightforward on purpose. Apart from the `treehash` function, we implement the signature routine directly into `sign.c`. To increase readability, we perform the handling of special cases via predicates. We include pseudocode for the Treehash algorithm in Algorithm 1.

In Algorithm 1, note that a forward-secure Treehash algorithm differs only in line 3, possibly also returning $\mathcal{R}_{\varphi+2^{h/d}}$, as shown in:
$\text{Node}_{\text{right}}, \mathcal{R}_{i+1} \leftarrow \text{GenLeaf}(i, \mathcal{R}_i)$.

Also, while we denote the active node $\text{Node}_{\text{right}}$, it is only a right node for τ iterations. When storing $\text{Node}_{\text{right}}$ in line 12, it is either a left node or the root of the tree.

Algorithm 1 Treehash

```
1: procedure TREEHASH(φ)
2:     for i = 0 to 2^{h/d} do
3:         Node_right ← GenLeaf(i)
4:         if i ⊕ 1 = φ then              ▷ Check: is leaf i the sibling of leaf φ?
5:             Auth_0 ← Node_right
6:         τ ← get_tau(i)                  ▷ Number of trailing ones
7:         for j = 0 to τ do
8:             Node_left ← Arr_j
9:             Node_right ← H(Node_left, Node_right)
10:            if ⌊i/2^{j+1}⌋ ⊕ 1 = ⌊φ/2^{j+1}⌋ then
11:                Auth_{j+1} ← Node_right
12:        Arr_τ ← Node_right
13:    Root ← Arr_{h/d}
14:    return Root, Auth
```

5.3.2 FS-StackRestore

The second implementation is FS-StackRestore. The implementation aims to show a naive method of implementing Pyramid in a forward-secure manner. An alternative naive implementation that attains forward security that we considered, is one that caches all nodes in a tree. Both implementations do *not* scale well with large subtrees.

We base FS-StackRestore upon a basic observation concerning the Treehash algorithm. We first note that Treehash can always generate nodes to the right of the current `idx`, as expensive as these may be. Now, let S_i denote the Treehash stack that includes and/or is combined up to leaf i. For example: $S_{-1} = \emptyset$, $S_0 = \{N_{0,0}\}$ contains the first leaf node, $S_1 = \{N_{1,0}\}$ contains the first combined node on layer 1, etc. We observe that the left authentication path nodes, for leaf i, are a subset of S_{i-1}.

By restoring the Treehash state, and storing it again after a single iteration, we can continuously (1) take left authentication path nodes from the state, and (2) compute right nodes by finishing the Treehash instance from the state. The Treehash intermediate state contains nodes, these are not considered secret, and we can safely forward the FSG references. We include pseudocode for the modified Treehash algorithm in Algorithm 2.

In Algorithm 2, we highlight the different approaches that we take for obtaining left nodes (lines 3-5), right nodes (lines 6-13), and the current leaf node (line 7, $i = \text{idx}$).

Algorithm 2 Treehash StackRestore

1: **procedure** $\textsc{Treehash}(\varphi, \mathrm{Arr}^{\varphi-1}, \mathcal{R}_\varphi)$
2: $\quad$ $\mathrm{Arr} \leftarrow \mathrm{Arr}^{\varphi-1}$ ▷ Restore the stack of $\varphi - 1$
3: $\quad$ **for** $j = 0$ to h/d **do**
4: $\quad\quad$ **if** $\lfloor \varphi/2^j \rfloor \;\&\; 1 = 1$ **then** ▷ Restore left Auth nodes
5: $\quad\quad\quad$ $\mathrm{Auth}_j \leftarrow \mathrm{Arr}_j$
6: $\quad$ **for** $i = \varphi$ to $2^{h/d}$ **do** ▷ Start from the current leaf.
7: $\quad\quad$ $\mathrm{Node}_{\mathrm{right}}, \mathcal{R}_{i+1} \leftarrow \mathrm{GenLeaf}(i, \mathcal{R}_i)$
8: $\quad\quad$ **if** $i \oplus 1 = \varphi$ **then**
9: $\quad\quad\quad$ $\mathrm{Auth}_0 \leftarrow \mathrm{Node}_{\mathrm{right}}$
10: $\quad\quad$ $\tau \leftarrow \mathrm{get_tau}(i)$ ▷ Number of trailing ones
11: $\quad\quad$ **for** $j = 0$ to τ **do**
12: $\quad\quad\quad$ $\mathrm{Node}_{\mathrm{left}} \leftarrow \mathrm{Arr}_j$
13: $\quad\quad\quad$ $\mathrm{Node}_{\mathrm{right}} \leftarrow \mathrm{H}(\mathrm{Node}_{\mathrm{left}}, \mathrm{Node}_{\mathrm{right}})$
14: $\quad\quad\quad$ **if** $\lfloor i/2^{j+1} \rfloor \oplus 1 = \lfloor \varphi/2^{j+1} \rfloor$ **then**
15: $\quad\quad\quad\quad$ $\mathrm{Auth}_{j+1} \leftarrow \mathrm{Node}_{\mathrm{right}}$
16: $\quad\quad$ $\mathrm{Arr}_\tau \leftarrow \mathrm{Node}_{\mathrm{right}}$
17: $\quad\quad$ **if** $i = \varphi$ **then**
18: $\quad\quad\quad$ $\mathrm{Arr}^\varphi \leftarrow \mathrm{Arr}$ ▷ Store the stack of φ after 1 iteration
19: $\quad$ $\mathrm{Root} \leftarrow \mathrm{Arr}_{h/d}$
20: $\quad$ **return** $\mathrm{Root}, \mathrm{Auth}, \mathrm{Arr}^\varphi, \mathcal{R}_{\varphi+1}$

5.4 NFS-Simple & background

Simple is the algorithm presented in [KT21a] by Kosuge and Tanaka. The third implementation uses the implementation from [KT21b] as its external algorithm. We adapt the implementation from XMSS to Pyramid, and separate concerns like we noted in Section 5.2. We leave scheduling untouched. We will now give an informal overview of the Simple algorithm, covering the notions that we require for the last implementation, which is FS-Simple. We will first cover the BDS algorithm, a component of Simple. We describe BDS in its original context of XMSS, but the description also applies to Pyramid.

5.4.1 BDS & HRB

An XMSS signature includes an unique authentication path for every signature. The authentication path consists of a node on every height $0, \ldots, h-1$. A tree-traversal algorithm aims to store a limited amount of nodes, to prevent recomputation at a later point in time. Another aim is to balance computation time, regardless of the index of the node that we compute the authentication path for.

One could review the FS-StackRestore implementation in Section 5.3.2 for an extreme example of why computation time may vary. Because this naive implementation recomputes the entire tree to the right for every signature ("right nodes"), the average signature time should improve when moving further to the right.

BDS is a tree-traversal algorithm that is presented in [BDS08], in the context of XMSS; it is a construction for $d = 1$, a single tree. We quote: "[BDS] *balances* the number of *leaves* that are computed *in each round*." [BDS08].

Right nodes BDS computation balancing is a consequence of how it computes right nodes. BDS defines `TreeHash` "instances" for several heights. We task a Treehash instance $\texttt{TreeHash}_h$ with the computation of a right node on layer h. For every signature computation, BDS schedules some of these `TreeHash` instances to "update". A `TreeHash` instance updates by computing one leaf, combining the nodes like regular Treehash, and incrementing a value pointing to the next leaf that it should compute. The algorithm ensures that it performs the right *amount* of updates, on specific layers, to ensure that an inner node is present when required for the authentication path.

Scheduling unit A crucial detail is that the scheduling unit is the creation of *one* leaf (via update). BDS aims to balance leaf node generation because inner node computation (H) is insignificant compared to leaf node computation (OTS public key generation, F).

Left nodes Left nodes result from saving previous authentication path nodes, in a fashion similar to that of FS-StackRestore.

State initialization BDS initializes its state during key generation. Because we have to compute the root node during key generation, all nodes are available at this time. We retain expensive upper right nodes, save the first authentication path, and `TreeHash` instances store their "first" targets.

Finally, BDS is extended to the hypertree setting as mentioned in RFC 8391 [HBG+18]; the Simple paper refers to this adaptation as "HRB". The strategy is implementated in `xmss_core_fast.c` in [HRCW21]. Vertically, HRB grants every layer $0, \ldots, d-1$ a BDS instance. The instances update normally, i.e. whenever one requires a new authentication path. Horizontally, HRB prepares a *second* BDS state for the next tree in the layer. This time, the BDS state does not follow "for free" from root computation during XMSS key generation. However, balancing computation of the next BDS state is easily attainable. The *current* tree performs $2^{h/d}$ signatures and initializing the state for the *next* tree takes $2^{h/d}$ leaf computations. Thus, instead of $h/2-1$ updates [BDS08] on a layer that changes its authentication path, we require $h/2$ [HRCW21].

5.4.2 Simple

The Simple algorithm is a (hyper)tree-traversal algorithm that is specified for XMSS^{MT}. Its performance is comparable to that of HRB, but the state size is cut in half. Simple uses a modified BDS algorithm on the bottom hypertree layer, which contains two major changes compared to BDS:

- Scheduling for right nodes continues beyond tree borders;
- The usage of nodes from XMSS key generation is recognized as a special case for the first tree. BDS did not require `TreeHash` instances on layers $h-1, h-2$ because of reuse. Simple does use these additional `TreeHash` instances.

Modified BDS schedules $h/(2d)$ updates every round.

Finally, on the upper $d-1$ layers, Simple proposes a tree-traversal algorithm called MMT. MMT is based around recomputing *every* node that changes in the authentication path, therefore requiring at most $2^{h/d}-1$ updates for one leaf hop. Because Simple defines MMT on hypertree layers greater than 0, MMT can disperse these updates over $2^{h/d}-1$ updates in the tree below it, similar to HRB (but vertically). We include pseudocode for MMT in Algorithm 3.

In Algorithm 3, note that we take $\min\{\tau, h/d-1\}$ for readability; implementations may mask φ before get_tau to achieve the same result.

Algorithm 3 MMT.update

```
 1: procedure MMT.UPDATE(φ, Auth, State)
 2:     τ ← get_tau(φ)
 3:     for j = min{τ, h/d − 1} to −1 do
 4:         if j = τ then
 5:             ρ ← φ − 2^j + 1                          ▷ Left target
 6:         else
 7:             ρ ← φ + 2^j + 1                          ▷ Right target
 8:         for i = 0 to 2^j do
 9:             Node_right ← GenLeaf(ρ)
10:             ρ = ρ + 1
11:             while height(S.top) = height(Node_right) do
12:                 Node_right ← H(S.pop||Node_right)
13:             S.push(Node_right)
14:     for j = 0 to min{τ, h/d − 1} do
15:         Auth_j ← S.pop
16:     return Auth                      ▷ Updated Auth for leaf φ + 1
```

Unless $\varphi \equiv -1 \mod 2^{h/d}$, the next authentication path requires one left node and τ right ones. These are scheduled in lines 4-7.

The computation in lines 8-13 is normally dispersed over updates in lower layers, and performed by a `TreeHash` instance. ρ denotes the leaf index that the `TreeHash` instance is working at, within the layer of this MMT instance. S is the stack that is shared between MMT instances; the stack keeps track of the height of nodes.

5.5 FS-Simple

5.5.1 Forward-secure BDS

The original paper on BDS features a strategy for computing leaves using a forward-secure PRG. The strategy follows from the fact that on layer i, every 2^{i+1} leaf hops, we initialize $\texttt{TreeHash}_i$ *at a fixed distance* from the current leaf φ. The distance $d(i) = 3 \cdot 2^i$ depends on the layer. Let $\mathcal{R}_\varphi$ be the current FSG reference. We keep h forwarded references $\mathcal{R}_{\varphi+d(0)}, \ldots, \mathcal{R}_{\varphi+d(h-1)}$ around, which we forward once for every signature, just like $\mathcal{R}_\varphi$. These are referred to as `SeedActive` by BDS. Then, when we initialize a Treehash instance $\texttt{TreeHash}_i$, it is given a copy of $\mathcal{R}_{\varphi+d(i)}$, referred to as `SeedNext` by BDS. $\texttt{TreeHash}_i$ computes 2^i leaf nodes from this and overwrites the reference afterwards. The modifications to BDS in Simple do not clash with the above construction. Notice that creating a forward-secure version for BDS is relatively easy because it never needs to recompute a left leaf.

5.5.2 Forward-secure MMT

In contrast to BDS, MMT does recompute left leaf nodes, i.e. once for every authentication path change. We will split our strategy into two parts.

Left nodes

MMT recomputes *any* node that we require for the next authentication path. This change may be visualized by incrementing $\varphi + 1 = (b_{h/d-1}, \ldots, b_0)_2 + 1$: bitflips $b_i = 1 \to b_i = 0$ require a new right node on layer i. The carry lands on $b_j = 0 \to b_j = 1$; layer j requires a new left node. This recomputation is impossible while using FSG.

Luckily, we can easily work around the issue with some minor modifications to MMT. We describe our approach using Figure 5.5.2.

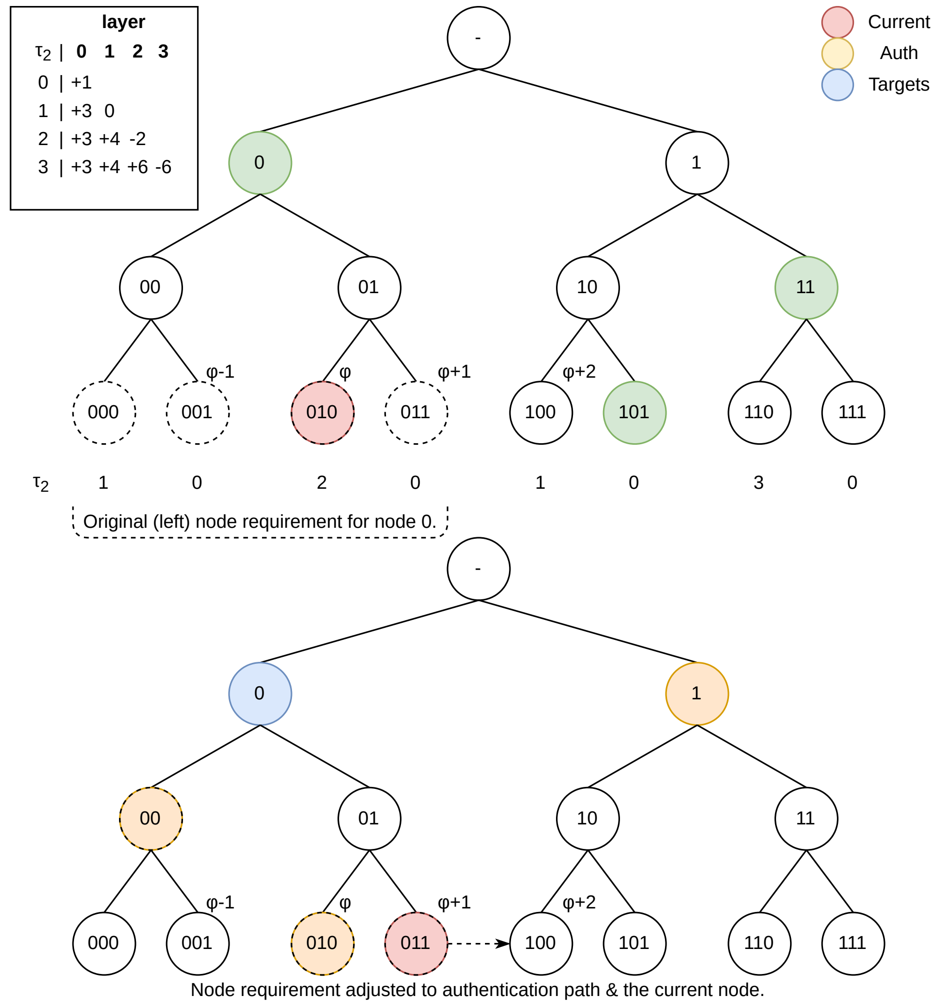

Figure 5.1: A FS MMT scheduling example of leaf hop 011 → 100.

The upper tree in Figure 5.5.2 depicts the dependence on nodes $000, \ldots, 011$ to compute node 0 in regular MMT. The lower tree depicts the dependence on nodes $00, 010, 011$ to compute node 0 in forward-secure MMT; we have forwarded the "current" node to highlight leaf availability. Note that we schedule the authentication path change for leaf hop $011 \to 100$ right before we perform hop $010 \to 011$. We disperse leaf computation over updates on lower layers, while we reside at leaf 011.

τ denotes the height of the first left parent of leaf $\varphi = 010_2$ or the carry position in our analogy. τ_2 is defined analogously for the next leaf $\varphi + 1$. The value τ_2 is equal to the number of authentication path changes after the upcoming leaf. One must schedule τ_2 `TreeHash` instances before leaving leaf φ. These complete over the course of staying at leaf $\varphi + 1$, thereby ensuring that the authentication path for leaf $\varphi + 2$ is ready when leaving leaf node $\varphi + 1$. In an authentication path update from index $\varphi - 1$ to index φ, MMT:

- Collects the nodes that it scheduled at transition $\varphi - 2 \to \varphi - 1$;
- Schedules the 2^{τ_2+1} nodes for transition $\varphi + 1 \to \varphi + 2$.

The target node at height τ_2 is depicted in green; this is a left authentication node that we require for transition $\varphi + 1$ to $\varphi + 2$. Now we observe that the target node also follows from *current* authentication nodes on height $0, \ldots, \tau_2 - 1$ and node $\varphi + 1$.

The most straightforward approach to computing the target node on heights τ_2, is to compute it just before we change the authentication path. One computes leaf $\varphi + 1$ and combines it with the *current* authentication path, to then update the authentication path. The problem with this approach is that this changes the worst-case signing time for MMT: a leaf may already be scheduled. Also, disabling the `TreeHash` instance that originally computes the node may be problematic, because of the shared stack.

We circumvent these issues simply by simulating the problematic `TreeHash` instance that we task with generating a left node. When the original `TreeHash` instance would have combined a node that is readily available in our authentication path, we push the node directly from there. For any other push/pop operation, we supply empty values. We leave the final update that would lead to the creation of the target node untouched. The result is our goal node and shared stack usage consistent with vanilla MMT. We compute *at most* as many leaves as vanilla MMT, at the cost of slightly more complex code. We include updated MMT pseudocode in Algorithm 4.

In line 8 of Algorithm 4, we test whether a left node is scheduled, which we do not have the FSG references for. If this is the case, we simulate a `TreeHash` instance w.r.t. its stack usage. Treehash combines τ_ρ stack leaves after generating a leaf at position ρ, simulated in lines 11-12. We push the result back onto the stack. We only care about reducing Auth nodes after the loop at line 9; when H would not have produced an Auth node, we push

Algorithm 4 Forward-secure MMT.update

1: **procedure** MMT.UPDATE(φ, Auth, State)
2: $\quad \tau \leftarrow$ get_tau(φ)
3: $\quad$ **for** $j = \min\{\tau, h/d - 1\}$ to -1 **do**
4: $\quad\quad$ **if** $j = \tau$ **then**
5: $\quad\quad\quad \rho \leftarrow \varphi - 2^j + 1$
6: $\quad\quad$ **else**
7: $\quad\quad\quad \rho \leftarrow \varphi + 2^j + 1$
8: $\quad\quad$ **if** $j = \tau$ **then**
9: $\quad\quad\quad$ **for** $i = 0$ to $2^j - 1$ **do**
10: $\quad\quad\quad\quad \tau_\rho \leftarrow$ get_tau(ρ)
11: $\quad\quad\quad\quad$ **for** $k = 0$ to τ_ρ **do**
12: $\quad\quad\quad\quad\quad$ S.pop ▷ Pop a placeholder
13: $\quad\quad\quad\quad$ **if** $\lfloor \rho/2^{\tau_\rho} \rfloor \oplus 1 = \lfloor \varphi/2^{\tau_\rho} \rfloor$ **then**
14: $\quad\quad\quad\quad\quad$ S.push(Auth$_{\tau_\rho}$) ▷ Push a valid Auth node
15: $\quad\quad\quad\quad$ **else**
16: $\quad\quad\quad\quad\quad$ S.push(0^n) ▷ Push a placeholder
17: $\quad\quad\quad\quad \rho = \rho + 1$
18: $\quad\quad\quad$ Now $\rho = \varphi$; perform GenLeaf($\varphi, \mathcal{R}_\varphi$) and combine with the
19: $\quad\quad\quad$ Auth nodes that we pushed. $\mathcal{R}_\varphi$ is in SK (current reference).
20: $\quad\quad$ **else**
21: $\quad\quad\quad$ **for** $i = 0$ to 2^j **do**
22: $\quad\quad\quad\quad$ Node$_{\text{right}}$, $\mathcal{R}^j_{\rho+1} \leftarrow$ GenLeaf($\rho, \mathcal{R}^j_\rho$)
23: $\quad\quad\quad\quad \rho = \rho + 1$
24: $\quad\quad\quad\quad$ **while** height(S.top) = height(Node$_{\text{right}}$) **do**
25: $\quad\quad\quad\quad\quad$ Node$_{\text{right}} \leftarrow$ H(S.pop$\|$Node$_{\text{right}}$)
26: $\quad\quad\quad\quad$ S.push(Node$_{\text{right}}$)
27: $\quad$ **for** $j = 0$ to $\min\{\tau, h/d - 1\}$ **do**
28: $\quad\quad$ Auth$_j \leftarrow$ S.pop
29: $\quad$ **return** Auth ▷ Updated Auth for leaf $\varphi + 1$

a garbage value in line 16. We omit the code that assures references are consistent between GenLeaf calls.

Right nodes & seeds

Without the exceptional case in MMT for left nodes, a $\texttt{TreeHash}_i$ instance in MMT remains at a fixed distance $d(i) = 2^i$ from current index φ. Each $\texttt{TreeHash}_i$ instance calculates every second right root on height i.

Our strategy is as follows. First, we grant every $\texttt{TreeHash}_i$ instance a forwarded reference $\mathcal{R}_{\varphi+d(i)}$, akin to `SeedActive` in BDS. Then, when we transition from leaf φ to $\varphi + 1$, MMT schedules the right nodes that we need for the authentication path of leaf $\varphi + 2$. We compute these while at leaf $\varphi + 1$, per vanilla MMT. When we arrive at leaf $\varphi + 2$, we then have $\mathcal{R}_{\varphi+d(i)+2^i}$ in $\texttt{TreeHash}_i$, if we required a new right authentication node on layer i. The seed reference has advanced 2^i positions. Now, notice that we have until leaf $\varphi + 2^{i+1}$ before we schedule another right node on layer i. The next node requires the reference $\mathcal{R}_{\varphi+d(i)+2^{i+1}}$. This gives us 2^{i+1} leaf steps to advance the reference the remaining 2^i times on layer i.

With this, we achieve the goal of keeping reference forwarding dispersed over subtrees. We choose to "freeze" an instance $\texttt{TreeHash}_i$ whenever it is in a forwarded state. Upon reaching leaf $\varphi + d(i)$, we unfreeze the instance and it receives one update per leaf hop, like in BDS. This keeps the reference synchronized until one uses $\texttt{TreeHash}_i$ again. This is the simplest approach; one could also attempt to further disperse the freezing scheduling.

In conclusion, we only require one additional seed for every `TreeHash` instance, which is not the case in forward-secure (modified) BDS. The modified MMT pseudocode may be found in Algorithm 4. We give two visualizations in Appendix A.3. Appendix A.3.1 depicts an example of MMT node scheduling. Appendix A.3.2 gives a counterexample, showing why we require an additional `SeedNext` per `TreeHash` instance in BDS, in contrast with MMT.

Chapter 6

Implementations in Jasmin

We previously mentioned that the C implementations should be treated with care. We can currently only cross-verify the signature output of implementations against each other. Even then, the implementations could accidentally veer from the specification, which may be pointed out by an independent reference implementation. Ideally, we implement Pyramid in a language that can achieve a good performance, while (1) verifying that the implementation computes according to the Pyramid specification (functional correctness) and (2) verifying that the implementation computes its results safely (constant-time, memory safety). Jasmin is a framework for developing high-speed and high-assurance cryptographic software, which can achieve the above when combined with EasyCrypt [Mei21a]. It is presented in [ABB+17] and currently being prepared for a first release.

In this work, our goal is to provide Jasmin counterparts to the Pyramid C implementations NFS-Naive and FS-Simple. We will not leverage the Jasmin/EasyCrypt framework further; no proof of functional correctness or memory safety is given. We hope to achieve two results:

1. Provide two Jasmin implementations for Pyramid that one can expand upon, with both having been *tested* for functional correctness;
2. Provide implementational notes pointing to constructions that could be of interest, e.g., to the designers of Jasmin. To the best of our knowledge, this is the first attempt for an implementation of a hash-based signature scheme in Jasmin.

Please consider the fact that the development of Jasmin is ongoing at the time of writing. We hope that we have considered all language features at the time of implementation. However, parts of the implementation could already be dated, due to the rapid development of Jasmin.

We stress that our implementation is *not* optimized for speed. As the following sections will point out, overcoming certain hindrances was of greater concern than speed. Also, we expect the speed of the implementation to be

heavily dependent on the speed of the underlying hash function, along with the tree traversal algorithm that we use.

Our current implementation compiles for commit aa031ef of the branch "glob_array3" of Jasmin. During development, several changes to register allocation in the Jasmin compiler were made. These can break implementations that do not maximally spill. To avoid this, all functions spill all active registers onto the stack when calling another function that involves hashing. This allows the SHAKE256 implementation to use a maximum amount of registers. The speed of the hashing algorithm (e.g. SHAKE256) should heavily influence the performance of a Pyramid implementation.

We will now enumerate the constructions that may be of interest. Note that we use a Jasmin branch titled "glob_array3" [LGS+21]. A concise tutorial on setting up Jasmin can be found on [Sch21]. Apart from the "Further resources" listed in the tutorial, one may find Jasmin repository directories `compiler/examples/` and `compiler/tests/` helpful, along with the Jasmin Wiki [LKG22].

6.1 General design choices

Typing Throughout the two implementations, our main aim is to take full advantage of the Jasmin typing system, which eases verification of the implementation and improves readability. An example is that we always prefer `reg ptr u8[PYR_N] foo` over `reg u64 foop` when we know the size of the region that we want to access through the pointer `foo`. The former construction is required to use Jasmin stack arrays. We prefer the most specific typing possible.

Looping In Jasmin, we have two looping constructs: `for` loops and `while` loops. `for` loops are always unrolled. When looping bounds are determined at runtime, the natural choice is a `while` loop. When looping bounds are available at compile-time, one must choose between the two constructs.

The file `utils.jahh` contains implementations for commonly used routines, like copying `PYR_N` bytes. These are currently all `inline` functions and fully unrolled. This is infeasible for a realistic implementation; in the future these can be tweaked on a function-by-function basis.

In general, we use `for` loops in most places, combined with noinline functions. This is partially a consequence of our aim to achieve appropriate typing, combined with sub-arrays requiring a constant starting index; examples of such cases will follow.

Functions In Jasmin, we have three types of functions. `export fn` functions can be used outside of Jasmin. We use these functions for unit testing and for exposing the three main signing/verification functions to C:

`crypto_sign_seed_keypair`, `crypto_sign_signature`, and `crypto_sign_verify`.

`inline fn` functions are inlined in the callers code. We use these functions when calling them once (outside of a `for` loop), or when the functions are only a conceptual abstraction (e.g. a setter) that we would have written out in full, had we not cared about code structure.

`fn` functions compile to one sequence of assembly instructions, similar to a regular function in C. We regularly use these for functions with many instructions that are repeated in a `for` loop due to typing restrictions. We often couple this with the annotation `#[returnaddress="stack"]`. Without the annotation, the return address is kept in a register that we are not able to access/spill. Before calling `fn` functions, we spill active register values onto the stack most of the time.

Exit points In Jasmin, a function ends with a `return` statement. This is the only exit point. Returning from another location requires conditional statements; a statement like `goto` in C is not available. While this hurts readability in some cases (e.g. in `crypto_sign_signature`), this is not a problem for most functions. However, we did omit the translation of the `treehashx1` Treehash implementation from [RWW+21] into Jasmin, due to its control flow statements. The `treehashx1` function is written in C. The conditional `return`- and `break` statements inside of the `while` loop do not seem to translate to an elegant Jasmin implementation.

Compilation termination In C, we can use the `#error` directive to terminate compilation. In implementations that specify parameters at compile-time, this allows us to refuse compilation for certain parameter choices. We have not found an alternative for this in Jasmin as of yet, meaning that parameter sets have to be implemented with additional care compared to C.

6.2 Directory structure

The directory for the Jasmin implementations `Simple-Jasmin/` (FS-Simple) and `Naive-Jasmin/` (NFS-Naive) is as follows:

`ref/` Contains the C reference implementation that we test against. Header files include function declarations for Jasmin export functions (suffix `_jazz`). Header files also expose static functions from the C implementation, so we can perform tests for functional correctness against these.

`ref2/` Contains a C skeleton that uses the three main Jasmin export functions for key generation, signature generation, and signature verification.

`src/` Contains Jasmin source files. `.jahh` files contain internal Jasmin functions. `.jazz` files contain the Jasmin export functions. We use most export

functions to test the functional correctness of internal Jasmin functions. The file `export.jazz` contains the three export functions that we use in `ref2/`.

`test/` Contains tests for functional correctness.

6.3 NFS-Naive

6.3.1 SHAKE256/FIPS 202

We limit the hash function instantiations to those of SHAKE256. This is primarily because at the time of implementation, FIPS 202 was the only hashing standard that was implemented in Jasmin and supported by Pyramid. SHAKE256 in Jasmin is part of the Saber implementation in Jasmin [Mei21b]. We reuse the SHAKE256 implementation with minor modifications. In general, the Saber Jasmin implementation has been a source of inspiration for the Pyramid Jasmin implementation, showing off several features of the glob_array3 branch.

Like the Saber implementation, we include one function for every absorb/squeeze with a different constant input/output length. This also implies one function for every SHAKE256 call that uses these functions, along with three similar implementations for F, H and Th_λ. This allows us to preserve the most specific typing possible, at the cost of a (prohibitive) increase in code size. A language construct similar to the generic size argument in C++ could be of help for this problem.

We will note that, specifically for this implementation, we could reduce the amount of boilerplate code. Notice that in `pyramid_params.jahh`, the input lengths for PRF, the $\mathrm{H}_{\mathrm{msg}}$ constant part, and F are the same. Thus, one could remove two absorption lengths. However, this being a first implementation, paired with the simplification for PRF in our C implementation, we have chosen not to make this change.

6.3.2 Treehash

Our second point of interest is the Treehash function from NFS-Naive, shown in `ref/treehash.c` and mimicked in `src/pyramid_treehash.jahh`. The Treehash function generates- and combines $2^{h/d}$ leaves. In C, we implement Treehash using a `for` loop that generates one leaf per loop iteration. Because this requires an exponential number of iterations, for most values of h/d this loop should not be unrolled. In Jasmin, this means that a `while` loop is appropriate; `for` loops are always unrolled.

Our loop variable in C and Jasmin is called `idx`. The Treehash implementation in C is based on the Treehash implementation in the XMSS reference implementation [HRCW21]. During the node combining phase, the Treehash implementation in C “slides” over the buffer `stack` to combine pairs of adjacent nodes. The result is directly put back into the buffer

to continue combining, therefore spending a minimal amount of time moving nodes around.

In Jasmin, performing this same strategy elegantly seems unattainable. This is because it is currently impossible to compile a sub-array that has a non-constant start index. Therefore, we cannot mirror the following C call:

```
thash(stack + (offset-2)*PYR_N, stack + (offset-2)*PYR_N,
           2, pub_seed, hashtree_addr);
```

`offset` is based on `idx`, which is a runtime variable. Because we cannot perform the Treehash strategy that we use in the C reference code, we revert to copying nodes from the stack into a dedicated hashing buffer `current`. Note that any reading/writing from/to the stack still uses an offset that is based on `idx`. Luckily, Jasmin does allow non-constant indexing as shown in the following code segment.

```
for i = 0 to PYR_N {
    a = current[PYR_N+i]; /* result at offset PYR_N */
    th_stack[i + (int) offset] = a;
}
```

This allows one to copy the object from a non-constant offset to a temporary location for which the offset is known at compile time. This way, we do not have to alter functions that use the object, which keeps the "badness" contained to the problematic location.

Finally, we give three implementations of `nto`, the number of trailing ones. Through these, we give a minimal example of the possibilities of the intrinsic operators that are available in Jasmin. Please note that optimization of this function is by no means imperative.

6.3.3 WOTS

Seed expansion The function `expand_seed` expands the WOTS-TW secret key into the first chain elements. The number of output seeds is known at compile-time. This is an example of a case where we use a `for` loop to take advantage of sub-arrays.

Because we chose to use a `for` loop, the chain address is known at compile-time. To emphasize that this is the case, we define additional getters/setters in `pyramid_address.jahh` that accept an inline integer argument, as opposed to an argument that we pass through a register. Such functions end with an underscore. Note that the `type` field is naturally known at compile-time. We omit the underscore in this case.

Also, please consider that even though we implement these getters/setters with compile-time arguments in several places, we do not necessarily think that the loops that use them should be unrolled in the first place.

Chain generation The function `gen_chain` applies F `steps` times. `steps` and `start` are determined by the message that is signed/verified. The bound is determined at runtime. This is an example of a scenario in which we are required to use a `while` loop.

Address returns A problem in the previous WOTS C implementations is that it was not always clear whether a function would alter the `addr` argument. In Jasmin, this is not a concern: address alterations through a pointer are reflected by the function signature.

Jasmin warnings While compiling `pyramid_wots.jahh`, we currently encounter warnings of the form:

```
"src/pyramid_wots.jahh", line 227 (4-7):
warning: cannot ensure that the type u8[PYR_WOTS_PK_BYTES] is
         compatible with u8[PYR_WOTS_BYTES]
```

In XMSS, it is tradition to define the byte-length of a WOTS public key and a WOTS signature separately, even though they are equal. The distinction emphasizes the intention of the code. However, there seems to be no obvious way to inform the compiler that the two are equal.

6.3.4 Root computation

In the file `pyramid_compute_root.jahh`, we encounter a situation similar to that of Treehash in Section 6.3.2. Depending on the node that we are creating being a left/right node, we would like to generate it in the appropriate location of `buffer`, in preparation for the next H call. Intuitively, one may define H as follows:

```
fn thash2(reg ptr u8[PYR_N] out, reg ptr u8[2*PYR_N] in,
          reg ptr u8[PYR_N] pub_seed, reg ptr u32[8] addr)
          -> reg ptr u8[PYR_N]
```

Then, for `lr` being `0` or `1`, we call the function via:

```
buffer = thash2(buffer[lr*PYR_N:PYR_N], buffer, pub_seed, hashtree_addr);
```

This construction is not possible, because `thash2` does not specify that it modifies the region that is pointed to by its second argument. The writable register pointer `out` is not disjoint from at least one non-writable register pointer. This is a concern within several locations of the implementations in Jasmin. We currently solve this by defining a compound version for functions in which the above occurs. In the case of `thash2`, we define:

```
inline fn thash2_comp(reg ptr u8 [2*PYR_N] inout,
                      reg ptr u8 [PYR_N] pub_seed, reg ptr u32[8] addr,
                      inline int lr) -> reg ptr u8[2*PYR_N]
```

6.3.5 Parameters & compilation

Test parameter sets are located in `src/params/`. The files in `src/` require a file called `pyramid_params.jahh`. This file is a symbolic link that we create before compilation. The symbolic link allows one to compile against different parameter sets, without making changes to the source files. We employ a similar mechanism for the requirement of `pyramid_thash_shake256`. This is a symbolic link that either points to the robust Th instantiations, or to the simple Th instantiations that are both located in `src/`.

6.4 FS-Simple

Most of the points of interest from NFS-Naive are relevant for FS-Simple too. However, the C implementation for Simple uses one additional language construct that we would like to comment on.

6.4.1 Structures

The C implementations for Simple defines three structure types: `treehash_type`, `stack_type`, and `state_type`. These consist mostly of pointers, which we direct towards the appropriate parts of the secret key in `state_deserialize`. `TreeHash` instances are stored in a "columnar" fashion, i.e. h/d nodes for layers $0, \ldots, h/d - 1$, followed by h/d boolean values, etc.

Jasmin does not specify an abstraction like structures. Instead, we pass around a pointer `reg ptr u8[MMT_SIZE] state`, and define getters/setters for this object. The getters/setters are found in `simple_*_state.jahh`; the offsets (columnar) that are used by these are found in `simple_*_state_format.jahh`. Note that we do this for BDS and MMT separately, because the states do not follow the same format, nor are they of equal size. As a consequence, functions that take an argument `state_type *state` in C, require two implementations in Jasmin. Luckily, these are of limited amount.

Finally, we can observe another artefact of the approach in `simple_MMT_update_treehash.jahh`. In C, we could directly use the struct pointers as an argument in function calls. However, the `TreeHash` instances that we update are known at runtime. In Jasmin, this means that some members of the state, such as nodes, must be copied out of the state before we can use them in functions that expect array pointers for constant size regions. Again, this is due to sub-arrays requiring a constant starting index. When the index is known only at runtime, we must copy the object from the state into an array of the object's real size. Listing 1 gives a concrete example.

Listing 1 `MMT_th_get_sda` requires a runtime index.

```
/* From simple_MMT_state.jahh: */
inline fn MMT_th_get_sda(reg ptr u8[PYR_N] sda, reg ptr u8[MMT_SIZE] state,
                         reg u64 i)
-> reg ptr u8[PYR_N]
{
    reg u64 offset j;
    offset = MMT_TREE + MMT_TH_SDA;
    j = i;
    j *= PYR_N;
    offset += j;
    /* Copy with a runtime offset (in[i + (int) offset]). */
    sda = MMT_cpy_N(sda, state, offset);
    return sda;
}

/* From simple_MMT_update_treehash.jahh: */
/* The following copy is not required in C. */
sda = MMT_th_get_sda(sda, state, index);

/* treehash_comp expects sda to be typed: reg ptr u8[PYR_N] */
current[PYR_N:PYR_N], sda = treehash_comp(current[PYR_N:PYR_N], sda, sk_psd,
                                          idx_leaf, t, hashtree_addr);
```

6.5 Performance evaluation

To get an impression of the performance of the FS-Simple Jasmin implementation, we compare its speed, measured in CPU time, against its counterpart in C. We measure performance for two parameter sets from RFC 8391 [HBG+18]:

Name	n	h	d	w
`XMSS-SHAKE_10_512`	64	10	1	16
`XMSSMT-SHAKE_60/12_512`	64	60	12	16

While testing the Pyramid implementations, we test for one XMSS- and one XMSS^{MT} parameter set; the latter allows one to get an indication of the performance difference between the C- and Jasmin implementations for MMT. The former only uses BDS, because $d = 1$.

For the tests, we perform one key generation and 10.000 signing/verification iterations. To obtain accurate measurements, we first time the generation of 10.000 signatures in succession. Then, we time the verification of the produced 10.000 signatures in succession. Because signing- and verification speed depends on the message, we randomize every message. We fix the message length at 32 bytes. We note that we do not deploy a CSPRG

for this test. We iterate each test three times and we use the average time in the results.

6.5.1 System information

We use the following platform for the performance evaluation:

```
Kernel: Linux
Version (partial): 20.04.1-Ubuntu
HW platform: x86_64
OS: GNU/Linux
```

We use the following gcc version for the performance evaluation:

```
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
```

We use Jasmin commit aa031ef of the branch "glob_array3" for compilation; this is the same Jasmin version that we use to test for functional correctness.

6.5.2 Results

We summarize the performance test results in Table 6.1. In Table 6.1, the performance of the Jasmin implementation for the parameter set `XMSSMT-SHAKE_60/12_512` is missing. We were unable to compile the code for this parameter set. The error that is thrown during compilation is shown in Listing 2.

This compilation error is expected, given the fact that a large amount of unnecessary unrolling takes place in the implementation. We have not experienced this error for the test parameter sets that we use to test for functional correctness for $d > 1$. Again, we note that our usage of `for` loops aims to preserve code reusability through typing correctness. As shown by the test results in Table 6.1, excessive unrolling slows down the code, and may even prevent compilation of the code.

In general, the performance of the Jasmin implementation is about 1.7 times worse than that of the C implementation, using SHAKE256. This is purely based on computation performed by the BDS algorithm, and for a relatively large value of n.

Name	C (s)	C (r)	Jasmin (s)	Jasmin (r)
XMSS-SHAKE_10_512	✓	✓	✓	✓
- keygen	200.18	319.87	339.28	530.81
- sign	213.9	341.25	362.63	579.63
- verify	14.85	24.45	25.55	41.52
XMSSMT-SHAKE_60/12_512	✓	✓	×	×
- keygen	1.19	1.84	-	-
- sign	451.77	699.30	-	-
- verify	176.44	281.56	-	-

Table 6.1: Performance of the C and Jasmin implementations for simple (s) & robust (r) Th strategies in seconds.

Listing 2 Stack overflow during Jasmin compilation for parameter set XMSSMT-SHAKE_60/12_512.

```
jasminc -lea -pasm  ../src/export.jazz -o jexport.s
"../src/pyramid_wots.jahh", line 232 (4-7):
warning: cannot ensure that the type u8[PYR_WOTS_PK_BYTES]
    is compatible with u8[PYR_WOTS_BYTES]
(we have redacted 5 more param int compatibility warnings)
Fatal error: exception Stack overflow
make: *** [Makefile:20: jexport.s] Error 2
```

Chapter 7

Related Work

We have already pointed to several related (stateful) hash-based signature schemes that we have considered for this proposal. Therefore, we will use this section solely for a brief comment on LMS.

The second NIST-approved stateful hash-based signature scheme is LMS, a description may be found in RFC 8554 [MCF19]. LMS uses the WOTS-like LM-OTS as its OTS: its security argument relies on the random oracle model. The security of LMS is proven in the quantum random oracle model in the multi-user setting [Eat17]. SPHINCS$^+$'s "simple" tweakable hash function instantiations are inspired by the constructions that are used in LMS.

To make hash-function calls unique for every user and instance, LM-OTS appends a string like $s = I\|Q\|i$ to its hash function input. I is similar in purpose to Seed, Q identifies the OTS (similar to the combination of layer, tree, and keypair in SPHINCS$^+$), and i identifies the chain in the OTS. The chaining function definition then appends a string $b\|00$; b specifies the chain position and "00" is akin to the type field in Pyramid, or the padding for the independence of the different function families in RFC 8391.

Inconveniently, even though the construction of LMS is much like that of XMSS and SPHINCS$^+$, a code-size optimized verification implementation for LMS and SPHINCS$^+$ needs to be larger than strictly necessary, only to achieve the same properties in two distinct ways. This is the main argument behind the proposal for Pyramid, *or* a stateless signature scheme that is (out of the box) compatible with LMS.

Chapter 8

Conclusions and Future Work

We have provided a first proposal for the stateful hash-based signature scheme Pyramid. The proposal is compatible with the addressing scheme and the hash-function instantiations of SPHINCS$^+$. As such, including Pyramid support in a verification routine for SPHINCS$^+$ signatures should result in a relatively small increase in code size. Additionally, we extend the addressing scheme and provide constructions that take advantage of the optimizations in SPHINCS$^+$. This being the first proposal, we include discussion for potential improvements when we deem a construction experimental to some degree.

We provide four reference implementations, which gradually introduce optimizations and use known tree-traversal algorithms that were defined in the context of XMSS. These implementations indicate what a forward-secure implementation encompasses, provide a baseline for future implementations, and show how one could keep a signature routine structured. Again, we warn that these are the first implementations for Pyramid, and should therefore be treated with care.

Finally, we implement a naive regular implementation of Pyramid in Jasmin, along with a space-optimized forward-secure Simple implementation. We point out minor hurdles for these implementations. We hope that parts of these implementations may be of future use in the verification process of Pyramid and SPHINCS$^+$.

Bibliography

[ABB$^+$17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

[ABB$^+$20a] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020.

[ABB$^+$20b] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS$^+$: Submission to the NIST post-quantum project, v.3. October 2020.

[BDG$^+$13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of security analysis and design vii*, pages 146–166. Springer, 2013.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS – Practical Forward Secure Signature Scheme based on Minimal Security Assumptions. In *International Workshop on Post-Quantum Cryptography*, pages 117–129. Springer, 2011.

[BDS08] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle Tree Traversal Revisited. In *International Workshop on Post-Quantum Cryptography*, pages 63–78. Springer, 2008.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ Signature Framework. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 2129–2146, 2019.

[BHRvV21] Joppe W. Bos, Andreas Hülsing, Joost Renes, and Christine van Vredendaal. Rapidly Verifiable XMSS Signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 137–168, 2021.

[BM99] Mihir Bellare and Sara K Miner. A Forward-Secure Digital Signature Scheme. In *Annual international cryptology conference*, pages 431–448. Springer, 1999.

[BY03] Mihir Bellare and Bennet Yee. Forward-Security in Private-Key Cryptography. In *Cryptographers' Track at the RSA Conference*, pages 1–18. Springer, 2003.

[DOTV08] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital Signatures Out of Second-Preimage Resistant Hash Functions. In *International Workshop on Post-Quantum Cryptography*, pages 109–123. Springer, 2008.

[Eat17] Edward Eaton. Leighton-Micali Hash-Based Signatures in the Quantum Random-Oracle Model. In *International Conference on Selected Areas in Cryptography*, pages 263–280. Springer, 2017.

[Flu17] Scott Fluhrer. Reassessing Grover's Algorithm. *Cryptology ePrint Archive*, 2017.

[Gro96] Lov K. Grover. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[HBG+18] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018.

[HK21] Andreas Hülsing and Mikhail Kudinov. Security of WOTS-TW scheme with a weak adversary. In *pqc-forum*. 2021. `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/91GRrrnXBuY/m/QnGMM9fKBgAJ`, accessed 18-02-2022.

[HKS20] Andreas Hülsing, Matthias Kannwischer, and Peter Schwabe. Forward-secure XMSS based on RFC 8391, 2020. `https://github.com/mkannwischer/xmssfs`, accessed 18-02-2022.

[HRB13] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal Parameters for $XMSS^{MT}$. In *International Conference on Availability, Reliability, and Security*, pages 194–208. Springer, 2013.

[HRCW21] Andreas Hülsing, Joost Rijneveld, David Cooper, and Bas Westerbaan. XMSS reference code, 2021. `https://github.com/XMSS/xmss-reference`, accessed 18-02-2022.

[HRS16] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating Multi-target Attacks in Hash-Based Signatures. In *Public-Key Cryptography–PKC 2016*, pages 387–416. Springer, 2016.

[Hül13a] Andreas Hülsing. *Practical Forward Secure Signatures using Minimal Security Assumptions.* PhD thesis, Technische Universität, Darmstadt, August 2013. `http://tuprints.ulb.tu-darmstadt.de/3651/`.

[Hül13b] Andreas Hülsing. W-OTS^{+} – Shorter Signatures for Hash-Based Signature Schemes. In *International Conference on Cryptology in Africa*, pages 173–188. Springer, 2013.

[KT21a] Haruhisa Kosuge and Hidema Tanaka. Simple and Memory-efficient Signature Generation of $XMSS^{MT}$. In *Selected Areas in Cryptography*, 2021.

[KT21b] Haruhisa Kosuge and Hidema Tanaka. Simple and Memory-efficient Signature Generation of $XMSS^{MT}$, 2021. `https://github.com/HaruCrypto54/xmss_simple`, accessed 18-02-2022.

[Lam79] Leslie Lamport. Constructing Digital Signatures from a One Way Function. Technical report, Citeseer, October 1979.

[LGS+21] Vincent Laporte, Benjamin Grégoire, Pierre-Yves Strub, Adrien Koutsos, Manuel Barbosa, Tiago Oliveira, Benoît Viguier, Jean-Christophe Léchenet, José Bacelar Almeida, Peter Schwabe, Jan Gilcher, `artart78`, `Simoncd89`, and `jba-uminho`. Jasmin: glob_array3. 2021. `https://github.com/jasmin-lang/jasmin/tree/glob_array3`, accessed 18-02-2022.

[LKG22] Vincent Laporte, Adrien Koutsos, and Benjamin Grégoire. Jasmin Wiki. 2022. `https://github.com/jasmin-lang/jasmin/wiki`, accessed 22-03-2022.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, April 2019.

[Mei21a] Matthias Meijers. Formal Verification of Post-Quantum Cryptography. 2021. `https://csrc.nist.gov/Presentations/2021/formal-verifcation-of-post-quantum-cryptography`, accessed 18-02-2022.

[Mei21b] Matthias Meijers. SABER-Jasmin. 2021. `https://github.com/MM45/SABER-Jasmin`, accessed 18-02-2022.

[Mer89] Ralph C. Merkle. A Certified Digital Signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[oST20a] National Institute of Standards and Technology. Stateful Hash-Based Signatures. 2020. `https://csrc.nist.gov/Projects/stateful-hash-based-signatures`, accessed 18-02-2022.

[oST20b] National Institute of Standards and Technology. Stateful Hash-Based Signatures: Public Comments on Draft SP 800-208. 2020. `https://csrc.nist.gov/CSRC/media/Publications/sp/800-208/draft/documents/sp800-208-draft-comments-received.pdf`, accessed 18-02-2022.

[oST21] National Institute of Standards and Technology. PQC: Round 3 Submissions. 2021. `https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions`, accessed 18-02-2022.

[Rij19] Joost Rijneveld. *Practical Post-Quantum Cryptography*. PhD thesis, Radboud University Nijmegen, 2019. `https://joostrijneveld.nl/thesis/`.

[RWW+21] Joost Rijneveld, Bas Westerbaan, Thom Wiggers, Peter Schwabe, Scott Fluhrer, Ruben Niederhagen, Stefan Kölbl, and MrPugh. SPHINCS+, 2021. `https://github.com/sphincs/sphincsplus`, accessed 18-02-2022.

[Sch21] Peter Schwabe. Getting started with Jasmin. 2021. `https://cryptojedi.org/programming/jasmin.shtml`, accessed 18-02-2022.

[Sho94] P.W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

Appendix A

Figures

A.1 Pyramid format

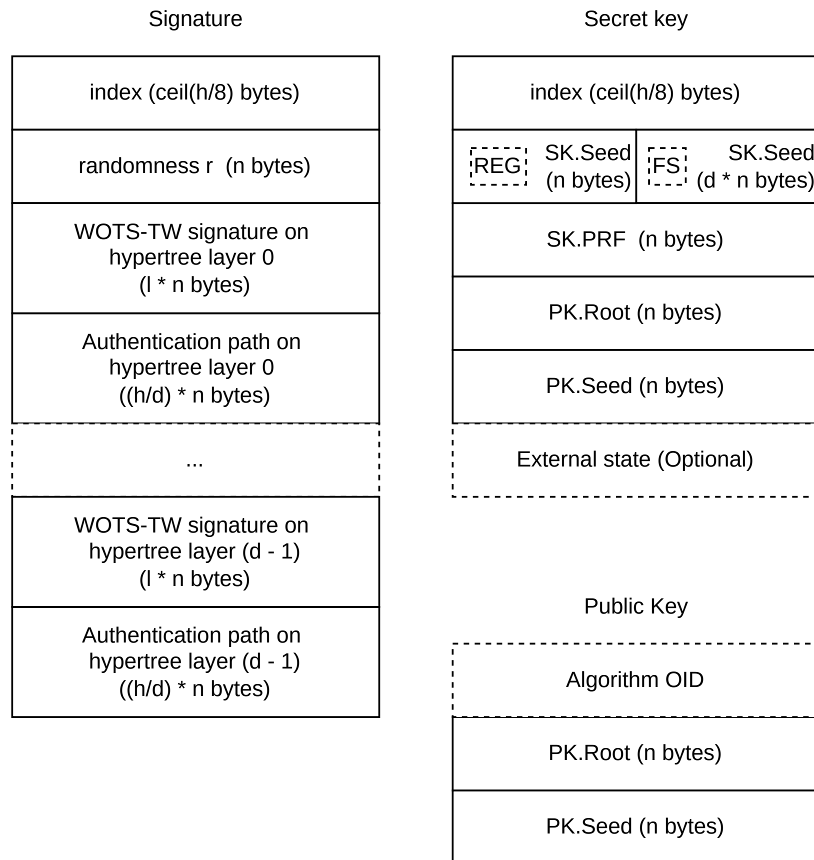

Figure A.1: The Pyramid signature, public key, and secret key format.

A.2 Implementation key & state structures

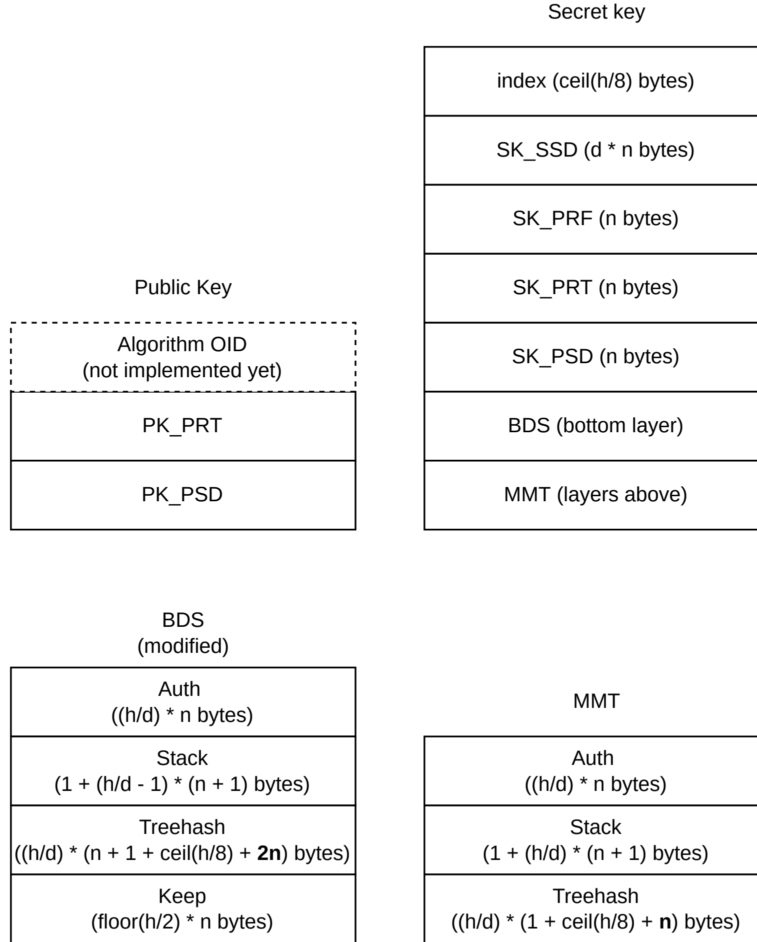

Figure A.2: Key- and state structures for Pyramid and FS simple.

Figure A.2 shows the public- and secret key structures, using the names of the offsets that we use in the implementation. `SK_SSD` is n bytes (1 seed) for regular Pyramid; we use d references in forward-secure Pyramid. The `BDS/MMT` states are optional; these can be disregarded or substituted for other tree-traversal algorithm states. The additional references that are stored by `TreeHash` instances in forward-secure BDS/MMT have their storage cost depicted in **bold**. We have not added the algorithm OID to our public key in our implementations, but a finalized implementation/proposal should include these at all costs. A consideration for the future could be to re-organize the secret key in categories “secret-state”, “secret-constant”, “public-state” and “public-constant”.

A.3 MMT & BDS scheduling examples

A.3.1 MMT

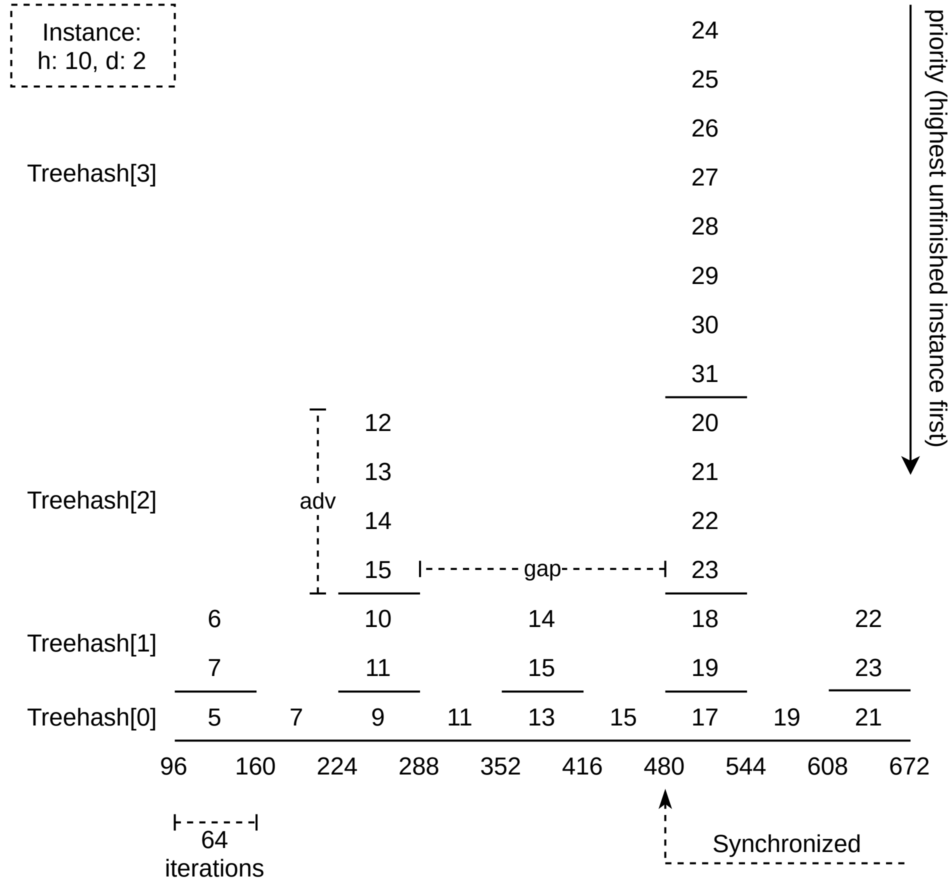

Figure A.3: MMT seed forwarding example.

Figure A.3.1 provides an additional example of MMT scheduling. Every second $2^{h/d}$ global updates, a `TreeHash` instance is scheduled. `TreeHash` updates advance the active reference beyond a fixed distance (`adv`). In `gap`, we ensure that we forward a reference (e.g. *from* leaf `15`) to the one that we require next (e.g. *to* leaf `23`), taking into account `adv`.

A.3.2 BDS

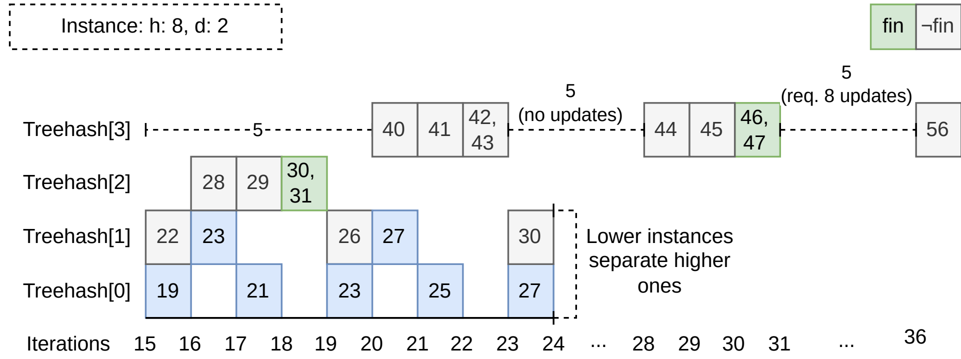

Figure A.4: BDS seed forwarding counterexample.

Figure A.3.2 provides an additional example of BDS scheduling. In forward-secure BDS, we keep two references for every `TreeHash` instance. In forward-secure MMT, we only need one reference for every `TreeHash` instance. The above Figure shows a counterexample of using one reference per `TreeHash` instance in modified BDS while maintaining evenly distributed FSG calls.

Because lower `TreeHash` instances interrupt the higher ones, one must retain an active reference for longer in higher instances. An example is the `5` update gap in rounds `23-28`. When the higher `TreeHash` instance finishes, it must then make up for the interruption by performing undispersed updates. For example, $\texttt{TreeHash}_3$ required `8` updates in just `5` rounds after it completed its first target node.

Note that it may be worth exploring whether/when the additional reference in $\texttt{TreeHash}_0$ is required, given that this instance should not be interrupted.

A.4 Sign signature skeleton

```c
/**
 * Returns an array containing a detached signature.
 */
int crypto_sign_signature(uint8_t *sig, size_t *siglen,
                          const uint8_t *m,
                          size_t mlen, uint8_t *sk) {
  const unsigned char * const sk_prf = sk + SK_PRF;
  const unsigned char * const sk_psd = sk + SK_PSD;
  unsigned char mhash[PYR_N];
  unsigned char r[PYR_N];
  uint64_t idx;
  int retval = 0;

  idx = bytes_to_ull(sk, PYR_IDX_BYTES);

  if(invalid_idxp(idx)){
    *siglen = 0;
    return -2;
  }

  initialize_hash_function(sk_psd, sk_prf);

  sig_mhash(mhash, r, sk, m, mlen);
  sig_base(sig, siglen, r, idx);
  retval = ext_finalize_sig(sk, sig, siglen, mhash);

  if(last_idxp(idx)){
    delete_sk(sk);
    return -1;
  } else {
    /* Algo-specific state update. */
    retval |= ext_update_state(sk);
    /* Forward-secure state update. */
#ifdef FORWARD_SECURE
    forget_seeds(sk);
#endif
    /* Post-algo: increment sk index by one. */
    ull_to_bytes(sk, PYR_IDX_BYTES, idx + 1);
  }

  return retval;
}
```

Appendix B

Software rescources

We accompany this document with C- and Jasmin implementations. These are available in the repository Pyramid proposal.

The C implementation directories are `FS-Simple`, `FS-StackRestore`, `NFS-Simple`, and `NFS-Naive`. The C implementations share the `common` directory. The directories that contain Jasmin implementations are `Simple-Jasmin` and `Naive-Jasmin`. We accompany the implementations with `README.md` files; these include instructions for building the software, along with other practical details that may be of use. We reiterate instructions for building the Jasmin project in Section B.2. The instructions describe the build process for `Simple-Jasmin`, which is similar to that of `Naive-Jasmin`. We use Jasmin commit aa031ef of the branch "glob_array3" for compilation.

From here, we omit the (active) directory name `Simple-Jasmin/`.

B.1 Directory structure

The directory `Simple-Jasmin` is structured as follows:

`asm/` contains Jasmin-generated assembly files produced by `Makefile`.

`Makefile` specifies recipes for tests for functional correctness.

`ref/` contains the Pyramid `FS-Simple` C reference implementation, modified to test for the functional correctness of the `Simple-Jasmin` implementation.

`ref2/` contains the Pyramid `FS-Simple` Jasmin-based implementation. More precisely, the directory contains a C skeleton to call the functions `crypto_sign_seed_keypair`, `crypto_sign_signature`, and `crypto_sign_verify`. These functions are implemented in Jasmin. These are exposed by `src/export.jazz`. Furthermore, the implementation contains its own minimal `ref2/Makefile`.

src/ contains the .jazz and .jahh source files.

params/ contains .jahh files that specify test parameter sets. A symbolic link in src/ is used to compile against one of these.

test/ contains randomized tests for functional correctness, along with testing utilities.

The files in src/ use the following prefix convention:

fips202_ files implement a part of FIPS 202, the SHA-3 standard. Most of these are a slightly modified version of the ones provided by the Saber implementation in Jasmin [Mei21b].

pyramid_ files implement core constructions like WOTS, addressing, etc. These are more or less the same, regardless of the tree-traversal algorithm that we use.

simple_ files implement the components of the Simple algorithm; examples include implementations of BDS and MMT.

crypto_ files implement Pyramid key generation, signing, and verification.

Finally, the prefixes are also conforming the prefixes that are used in the Saber implementation in Jasmin [Mei21b]:

.jazz files contain Jasmin export functions. Except for src/export.jazz, these provide a wrapper around functions in .jahh files, used for testing. Functions in .jazz files, suffixed _jazz, are exposed to C. Their function declaration is found in the analogous header files in ref/.

.jahh files implement the logic from the C implementation in ref/ in Jasmin.

B.2 Jasmin build

Makefile's primary goal is to build tests for functional correctness, i.e. every target test/bin/test_*. Example usage scenarios, including those for ref2/Makefile, are as follows. Instead of manually setting the variables PARAMS and THASH in Makefile, one may also choose to include variable assignments directly from the command-line:

```
make THASH=simple PARAMS=pyramid-shake256-test target.
```

Listings 3, 4, and 5 show examples of building parts of the Simple-Jasmin project.

Listing 3 Testing all individual Jasmin functions for parameter set `test_d_3`, using the "simple" tweakable hash function instantiations.

```
# Set PARAMS = pyramid-shake256-test_d_3 in Makefile
# Set THASH  = simple in Makefile
make clean # Clear the previous tests.
make prep  # Create symbolic links for (PARAMS, THASH) choice.
make test  # Build every target in TESTS.
```

Listing 4 Testing Jasmin-based Pyramid `FS-Simple` for test parameter set `test_n_32`, using the "robust" tweakable hash function instantiations.

```
# Set "PARAMS = pyramid-shake256-test_n_32" in Makefile
# Set "THASH  = robust" in Makefile
make clean # Clear the previous symbolic links.
make prep  # Create symbolic links for (PARAMS, THASH) choice.
cd ref2
make test
```

Listing 5 Testing the functional correctness of `crypto_sign_verify` located in `crypto_sign_verify.jahh`, which is exposed by `crypto.jazz`. Beforehand, we ensure that there are no basic discrepancies between `src/pyramid_params.jahh` and `ref/params.h`, using the sanity check.

```
# Assume a clean environment with symbolic links prepared.
make test/bin/test_sanity_check
# Output on stderr if accidental parameter mismatch.
./test/bin/test_sanity_check
make test/bin/test_crypto_sign_verify
./test/bin/test_crypto_sign_verify
```