



EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

A COMPARATIVE ANALYSIS OF POST-QUANTUM HASH-BASED SIGNATURE ALGORITHM

MR. PÉTER LIGETI

ASSISTANT PROFESSOR AT ELTE

MR. ANDREAS PETER

ASSOCIATE PROFESSOR AT UNIVERSITY OF
TWENTE

MR. ÁRON SZABÓ

IT SECURITY CONSULTANT AT E-GROUP ICT
SOFTWARE

KIMSUKHA SELVI S

COMPUTER SCIENCE

BUDAPEST, 2020.

STATEMENT OF THESIS SUBMISSION AND ORIGINALITY

I hereby confirm the submission of the Master Thesis Work on the Computer Science MSc course with author and title:

Name of Student: Kimsukha Selvi Sivasubramanian
Code of Student: G2T460
Title of Thesis: A Comparative Analysis of Post-Quantum Hash-based Signature Algorithm
Supervisor: Mr. Péter Ligeti
Assistant Professor

at Eötvös Loránd University, Faculty of Informatics.

In consciousness of my full legal and disciplinary responsibility I hereby claim that the submitted thesis work is my own original intellectual product, the use of referenced literature is done according to the general rules of copyright.

I understand that in the case of thesis works the following acts are considered plagiarism:

- literal quotation without quotation marks and reference;
- citation of content without reference;
- presenting others' published thoughts as own thoughts.

Budapest,



student



EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS



MASTER THESIS TOPIC DECLARATION FORM

Name of Student: KIMSUKHA SELVI SIVASUBRAMANIAN

Neptun code: G2T460

Training: Full-time

Technical major: CYBERSECURITY

Email Address: kimsukha@gmail.com

Phone Number: +36 706734056

Name of ELTE supervisor: Mr. Péter Ligeti

Workplace: ELTE University

Name of Entry-university supervisor: Mr. Andreas Peter

Workplace: University of Twente

Name of Industrial supervisor: Mr. Áron Szabó

Workplace: E-Group

Information about the internship

Name of the company: E-Group

Starting date of internship: 03/02/2020

Closing date of internship: 31st May 2020

Weekly schedule: 20 hrs/week

The acceptance statement is to confirm that the student at the Computer Science MSc course of ELTE Faculty of Informatics is able to accomplish his/her compulsory internship according to the appendix 4 of the 18 of 5 August 2016 decree of the Ministry of Human Capacities at the chosen institution and along the framework detailed below

Title of the thesis

A Comparative Analysis of Post-Quantum Hash-based Signature Algorithm

Topic of the thesis

As a new era of Quantum Technology is evolving, there grow new challenges in the existing cryptography used in every digital gateway. The potential of Quantum computers to solve any complex problem in a few minutes also weakens the security of the currently used public-key cryptographic solutions like RSA, ECDSA. To ensure secure solutions available against quantum computers attack, at the end of 2016, the National Institute of Standards and Technology (NIST) called for quantum-resistant asymmetric cryptographic algorithm proposals. The proposal submissions can be either an encryption algorithm or a signature algorithm, divided into five categories: code-based, lattice-based, hash-based, multivariate polynomial, other algorithms.

The motivation for our thesis is to work on signature algorithms submitted in the NIST competition, presenting a comparative study of Hash-based Signature algorithms - SPHINCS+ and LDWM (Lamport-Diffie-Winternitz-Merkle). SPHINCS+ is a stateless hash-based N-time signature that has been qualified by round 2 submissions. LDWM is an one-time signature scheme which can be used as a stateful hash-based N-time signature as well.

Research Questions :

1. A comparison of LDWM and SPHINCS+ algorithm based on the internal function calls in solving a problem. List out the Pros and Cons.
2. Secure parameters that a Developer has to take care of in applying an algorithm in Real-time.
3. Data required for generating a certificate for SPHINCS+.

A Mathematician creates a signature algorithm based on various parameters to ensure the algorithm renders strong authentication, non-repudiation, and integrity. Hence, while implementing the algorithm in the real-time application, a developer should pay attention to mathematical problems, parameterization, and environmental dependencies. The comparative study helps a developer to understand the pros and cons of the hash-based signature, its parameters and implement the SPHINCS+ algorithm for certificate creation.

To answer the questions, first I would start with a literature review for understanding the code of SPHINCS+ and LDWM. secondly, gather the parameter values used in the algorithms, To answer (3), based on the research has performed apply it in cryptographic scenarios like the creation of X.509 certificates using SPHINCS+.

Keywords: Quantum computers, NIST, Hash-based signature, SPHINCS+, LDWM.

References:

- [1] <https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions>
- [2] <https://sphincs.cr.yt.to/>
- [3] <https://huelsing.net/wordpress/?p=558>
- [4] <https://tools.ietf.org/html/rfc8554>
- [5] http://www.kormanyablak.org/it_security/2015-11-22.php

Encryption of the topic is necessary: **YES/NO**

I approve of the suggested topic of the Master's Thesis:

Budapest, 29/01/2020



ELTE supervisor

I approve of the suggested topic of the Master's Thesis:

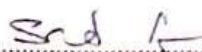
Budapest, 21/01/2020



Entry-university supervisor

I approve of the suggested topic of the internship, and in the name of the institution (organization, company) above I agree, that the named student will carry out his/her internship along the conditions detailed above:

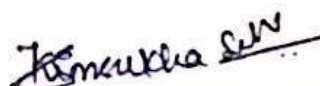
Budapest, 31/01/2020



Industrial supervisor

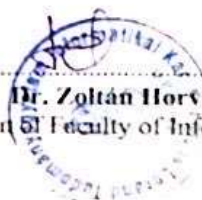
I ask for the acceptance of my thesis topic.

Budapest, 31/01/2020



Student

The topic of the thesis and internship is approved by the Department of ELTE Informatics
Budapest,/..../202



Dr. Zoltán Horváth
Dean of Faculty of Informatics

Acknowledgement

I would like to express my gratitude to my supervisors Mr.Péter Ligeti and Mr.Áron Szabó from E-Group for their guidance and support while I was working on my thesis. I would also like to thank my friends Ann Mariam and Veeraraghavan for reading my draft version and for providing valuable feedback, and my family for supporting me during my studies.

Thank you all.

Contents

1	Introduction	1
2	Preliminaries	4
2.1	Digital Signatures	4
2.1.1	Security Notions of Signature schemes	5
2.2	Post-quantum signatures	6
2.3	Hash Functions	10
2.4	One-Time Signature	11
2.5	Many-time signature	11
3	Basics of Post Quantum Signature Scheme	14
3.1	Hash-Based Signature Schemes	14
3.1.1	Lamport One-time signature (L-OTS)	15
3.1.2	Merkle Trees	16
3.1.3	Winternitz One-time signature (W-OTS)	17
3.2	Lamport-Diffie-Winternitz-Merkle Scheme (LDWM)	19
3.3	SPHINCS+	22
3.3.1	Components of SPHINCS+ Framework	23
3.3.2	SPHINCS+	25
4	Analysis of LDWM and SPHINCS+	29
4.1	Implementation Details	29
4.2	Key Attributes from NIST	30
4.2.1	Security	30
4.2.2	Cost/Performance Analysis	31
4.2.3	Algorithm	32
5	Conclusion	35

CONTENTS

6 Reference	36
6.1 Source code	36

Chapter 1

Introduction

Cryptography serves as a tool which enables encryption and decryption on an information, for secure communication in the universal public network such as the Internet. A cryptosystem is built with a unique structure combining encryption, decryption algorithms and generate a pair of keys that is mathematically linked. The Information is transformed into ciphertext and to plaintext with the encryption key(public key) and decryption(private key) respectively for a secure transmission between two parties over the frameworks like Internet Key exchange(IKE), Internet Security(IPSec),Transport layer Security(TLS) etc.This sort of cryptosystem is called "Public-key cryptography"[DS07] which includes a security prerequisite that the computation of private key from public key must be computationally infeasible. The degree of infeasibility is based on the size of the key(bit-level), to open the trapdoor called "hard" problem within the cryptosystem. The hard problem is intractable as there's no satisfactory way to solve it ,other than to brute-force. But brute-forcing with an existing classical computer takes a long time because the cryptosystem is bounded by the bit-level security wrapped around the hard problem i.e., as the size of the key increments, more computational assets are required for an adversary to discover the size of an input to break the trapdoor.

Most of the data transmission over Internet happens through Hypertext transfer protocol secure(HTTPS) and Transport layer Security(TLS), which consoles that the client information is transmitted to their intended website securely. The TLS contains a cipher suite, a combination of public-key cryptography schemes such as Rivest–Shamir–Adleman(RSA), Diffie-Hellman key exchange, Elliptic Curve Digital Signature Algorithm (ECDSA), Digital Signature Algorithm(DSA) that are strengthened by "hard" problems like integer factorization or discrete log. The cipher suite is mainly used for the key establishment, signature generation, message encryption, and authentication over HTTPS are also immune to classical computers attacks.The proficient degree for any computer to break the hard

problem in a desired polynomial-time is used to quantify the measure of data protected in the framework. So far, we believe that our cryptosystem with key size of 80 bit-level of security are an highly proficient degree. But, The National Institute of Standards and Technology(NIST) Report[Bar20] claims that a cryptosystem should now hold 112 to 128 bit-level of security for data protection from classical computers and the upcoming quantum computers. With the transition in the bits of security paradigm, the challenge down the lane in cryptography is the development of quantum innovation devising a quantum computer. The quantum computation is performed based the physical properties of matter and energy for an efficient calculation to solve any complex problem faster than classical systems. This innovation threatens the security of the existing public-key cryptographic solutions used to protect the data privacy. Based on the Shor's discovery in 1994, on quantum algorithm can break the integer factorization in polynomial-time. Hence, to ensure state-of-art security solutions available against quantum computers attack, at the end of 2016 the National Institute of Standards and Technology (NIST) called for submissions of "Public-Key Post-Quantum Cryptographic Algorithms". The Report[CJL⁺] from NIST has given an overview of the post-quantum cryptography family from which the proposed algorithm primitives are developed. There were 82 Quantum-resistant asymmetric cryptographic algorithm proposals were submitted and was evaluated on their secureness against quantum as well as classical computers. Eventually 26 algorithms were considered as an acceptable algorithm based on their design attributes and mathematical foundations for Round 2.

Our Goal. An adversary with Quantum computer would target any online transactions such as e-commerce, legal records/confidential data, internet banking etc., handled by the Public Key Infrastructure(PKI)[Tha] in HTTPS. The PKI is a part of TLS Handshake used for end-to-end encrypted communication between client and the server by issuing a SSL/TLS certificate digitally signed, assuring the identity of the server. SSL/TLS certificate is vulnerable to certificate based attacks, where an adversary would spoof an legitimate authority's certificate signature by eavesdropping and forge it to their own certificate thereby attaining the target network/machine. This indeed rise to the need of quantum-safe signature certificates in PKI.

As Daniel J. Bernstein et.al [Ber09] proposed that a Quantum collision search algorithm could find a collision in time complexity of $O(2^{n/3})$ with a use of $O(2^{n/3})$ hardware components and whereas with the same hardware, a classical computer along with multiple parallel units could find collisions in a time complexity of $O(2^{n/6})$. This implies Quantum computers are slower than classical computers to find the collision in hash, and so we concentrate on Hash-based signature(HBS) schemes. Our work focus on signature algorithms

submitted in the NIST competition, to present a comparative study of Hash-based Signature algorithms - SPHINCS⁺ and LDWM (Lamport-Diffie-Winternitz-Merkle). SPHINCS⁺ is a stateless hash-based N-time signature that has been qualified by Round 2 submissions and LDWM, a draft version of Leighton-Micali HBS (RFC8554) is an one-time signature scheme which can be used as a stateful hash-based N-time signature as well.

An Algorithmic complexity designer creates a signature algorithm based on various parameters to ensure that the algorithm renders the security properties like strong authentication, non-repudiation, and integrity in quantum era. Hence, while implementing the algorithm in the real-time application, a protocol designer should pay attention to mathematical problems, parameterization, and environmental dependencies. As any dedicated adversary could break the system, due to code vulnerability in library implementation. This comparative study helps a protocol designer to understand the hash-based signature, its parameters and implement a quantum resistant infrastructure.

Chapter 2

Preliminaries

As our goal of the thesis is to understand hash-based signature scheme, it is therefore necessary to first understand the basics of cryptography. This Chapter provides the knowledge about Hash functions, Digital signature, One-time and Many-time signatures. We will also briefly cover the introduction to post-quantum cryptography and its necessity for this decade along with the evolution of hash-based signature schemes. The above basics will help us to understand the objective of our thesis in Chapter 3.

2.1 Digital Signatures

Digital Signatures are more like physical signature that are unforgeable, ensures the authentication and integrity of any information transmitted over an secure channel with a public-key cryptography setting. The Public-key cryptography [SSD] setting has a pair of keys: a private key known only to the user and an equivalent public key known to the public. When an contract is signed between the two parties, it ensures the data in the contract is validated and agreed by both parties. Henceforth, the document cannot be modified intentionally or unintentionally as it is duly signed by the parties. A Digital signature is a virtual fingerprint on a digital document or message so that the document cannot be modified from the time its signed. Generally, the original digital document is hashed into a message digest and signed by the sender's private key before its transmitted to the receiver. The receiver generates the hash of the same document, decrypts the message digest using the sender's public key and compare it with the receiver's hash value generated earlier. If the hash value match, then the document is digitally unaltered. This ensures the following properties achieved by the digital signature scheme:

- Authentication : As the document is approved by the sender.

- Integrity : Due to the avalanche effect of the hash function, any change to the document will change the hash value with which the receiver can find whether the document is trustworthy.
- Non-Repudiation : The document is signed by the sender's private key and so, the sender cannot deny in case of loss of integrity.

2.1.1. Definiton. *The Digital signature scheme is a triple of algorithms (Gen, S, V) for a message space M .*

- $Gen(1^n)$, a random algorithm which generates a key pair: private signing key sk and public verification key pk with a security parameter n .
- $S(sk, m)$, a signing algorithm that takes a message and private key sk as input and outputs a signature σ . i.e., $\sigma \leftarrow S_{sk}(m)$.
- $V(pk, m, \sigma)$, a verification algorithm that takes a public key pk , message m , signature σ as input and outputs b as either '1' or '0', based on verification of valid signature on the Message m using the public key pk . i.e., $b \leftarrow V_{pk}(m, \sigma)$.

For all (pk, sk) generated by $Gen(1^n)$, if $\forall m \in M$ that holds $V_{pk}(m, S_{sk}(m)) = '1'$ then the Verification of the digital signature is deterministic and consistent.

2.1.1 Security Notions of Signature schemes

The notions are built based on the goal of an adversary A and the attack model provided to A to explore the system. In general, the goal of an adversary is to create a valid signature for a new message (Forgery) and the attack model will enable A to learn signature on messages of his choice with a known public key from a signing oracle. The standard security notion for signature scheme is the Existential forgery under chosen message attack (EU-CMA) [BH16], used in practice as any adversary cannot achieve the forgery attempt in this Chosen Message attack model. The EU-CMA is defined in [BH16] as an experiment, where $Dss(1^n)$ denotes signature scheme with security parameter n and q queries passed to the signing oracle by A .

Experiment $Exp_{Dss(1^n)}^{EU-CMA}(A)$

$(sk, pk) \leftarrow Gen(1^n)$

$(m^*, \sigma^*) \leftarrow A_{sk}^S(\cdot)(pk)$

Let $(m_i, \sigma_i)_1^q$ be the queries and results pair of $S_{sk}(\cdot)$ from the oracle.

Output 1 iff $V_{pk}(\sigma^*, m^*) = 1$ and $m^* \notin Mi_q$

else 0

From the experiment, the probability of an Adversary A succeeding can be written as

$$Pr[Exp_{Dss(1^n)}^{EU-CMA}(A) = 1]$$

2.1.2. Definiton. *The Digital signature scheme $Dss(1^n)$ [BH16], is (t, ϵ, q) existentially unforgeable under chosen message attacks(EU-CMA) if for all PPT adversaries A , the algorithm runs with a time complexity of at most t and generates at most q queries to the signing oracle. Then the probability of producing a valid signature for a message not previously sent to the signing oracle is $\leq \epsilon(t)$, a negligible function to break the scheme.*

$$Pr[Succ_{Dss(1^n)}^{EU-CMA}(n) = 1] \leq \epsilon(t)$$

With the above experiment and definition, the negligible probability of succeeding in breaking the scheme made the digital signature to be used widely in all building blocks of Internet and other infrastructure protocols.

2.2 Post-quantum signatures

With the wide spread of sensitive data online e.g legal records or software updates, Digital signatures are widely used for all electronic communication over security protocols like Transport layer(TLS) ensuring authenticity and identity of the server. Most of the Digital signatures are created by the asymmetric key cryptography relies on computational hard problems. This section briefs about the quantum attacks on classical systems, the need for NIST competition procedure and proposals.

Quantum-based attacks

Though the computational hard problems are efficient for classical algorithms, the recent advancement in quantum processing has broken such assumption. Most of the cryptographic algorithms like RSA, ECDSA, Diffie-Hellman key exchange are insecure on quantum computer as proposed in Shor's algorithm[Sho75], that could solve the integer factorization and discrete log problems in polynomial time. The Shor's algorithm is comprised of two parts: a Reduction part, where a classical system solves the factor structure to a period problem and a Quantum part, for finding the period value using quantum fourier transform.

A classical computer perform logical operations on the physical state of the switch with 1's and 0's called bits, whereas a quantum computer uses Quantum bits(qubits) to perform

operations based on the quantum state of the switch. This qubits have two property making the computer efficient than classical system.

- Superposition: Multiple states are possible at the same time.
- Entanglement: Change in one state would predict the change in another state.

Superposition property is used in the Quantum part of Shor's algorithm to calculate the period by iterating all the possible combinations of states, breaking the N value of RSA in polynomial time. Hence, we have to replace the existing digital signature, key exchange based on asymmetric algorithms.

According to another quantum-search algorithm called "Grover's algorithm"[Gro96], which can attack symmetric algorithms like AES128. Basically, Grover's algorithm performs an exhaustive search by traversing through an unsorted database of N entries, for a single matching entry within $O(\sqrt{N})$. This analogy could be used to explain an attack on a symmetric algorithm for a k -bit key, $2^{k/2}$ are required in polynomial time. This attack can be mitigated by increasing the key value of the algorithm from AES128 to AES256.

An adversary with a quantum computer with 4000 bits, use Shor's algorithm and Grover's algorithm can break both symmetric and asymmetric cryptosystems. The recent achievement of Google supremacy[EC] with 53 qubit device called "Sycamore", performed an operation in 200 seconds which would take 10,000 years for a supercomputer to compute.

The National Institute of Science and Technology(NIST) competition

The NIST is an U.S based non-regulatory federal agency governing advancement in sectors like electric power grid, nanomaterials, computers, physics etc.,. Their primary mission [Her] is to promote innovation and industrial competitiveness by advancing measurement science, standards and technology and thereby improving the economy. Due to the recent advancement in the quantum, a new branch of a scientific study Post-quantum Cryptography initiated by NIST in 2006, to provide a standardisation suite for algorithms against quantum computers. NIST gathered various proposals for evaluation in two rounds. The proposals were mainly focus on five categories of algorithms, which includes lattice-based cryptography, code-based cryptography, multivariate polynomial cryptography, hash-based signatures and others interim standard, with each focus on different mathematical problem that are quantum-resistant. The evaluation criteria [TZ19] of NIST for standardisation involves the Security, Cost and Algorithm/Implementation factors. Out of 82 proposals submitted, 69 algorithms were selected from Round 1 and then narrowed down to 26 algorithms with 9 signature schemes in Round 2. The brief description of the evaluation criteria[NIS]:

- *Security* Definition should match IND-CCA2 for encryption algorithm, EUF-CMA for signatures scheme, Side channel attack resistant for key exchange. Their security level to withstand any quantum attack complexity is well-explained in Table 2.1.
- *Cost* Computational requirements with respect to time(speed) and memory(size). Speed considers the hardware and software used to key generation/exchange, encryption/decryption, sign/verify and Memory requirements considers the key size, parameters for the security.
- *Algorithm* Possible Implementation in various platforms with tunable parameters including parallelism and should be resistance to any attacks.

Level	Description	Classical Bits	Quantum Bits
I	At least as hard to break as AES128.(Exhaustive key search)	128 bits	64 bits
II	At least as hard to break as SHA256.(Collision search)	128 bits	80 bits
III	At least as hard to break as AES192.(Exhaustive key search)	192 bits	96 bits
IV	At least as hard to break as SHA384.(Collision search)	192 bits	128 bits
V	At least as hard to break as AES256.(Exhaustive key search)	256 bits	128 bits

Table 2.1: Security Level Categories by NIST [TZ19]

To provide a state of art security solution for the digital signature with respect to the resource and time constraint, the following Figure 2.1 explains the classification of Hash-based signature schemes submitted to NIST proposal.

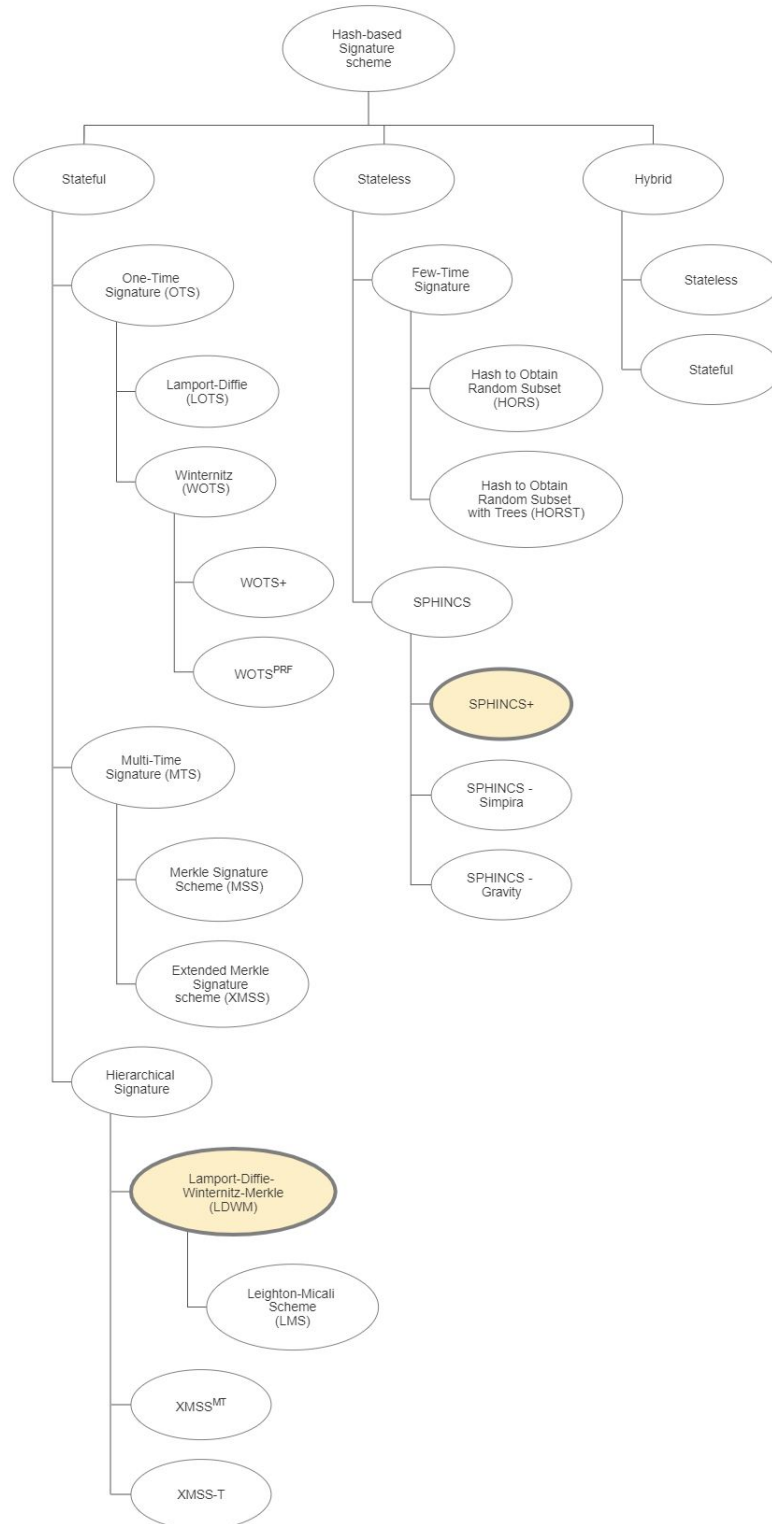


Figure 2.1: Classification of Hash-based Signature Schemes

2.3 Hash Functions

A Hash function is defined as a key-value pair that maps data of arbitrary input size to a fixed size output called hash code. For a function f and given a hash code value y represented as $f(x) = y$, it is computationally infeasible for an adversary to find the input key x . This is called the "One-way property". A cryptographic hash function H , is a combination of key-value pair and one-way property (OWF) which is represented as $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$ adheres the following properties, refer [RS04] for overview :

- For a given output $y = H(x)$, it is infeasible to find the input value x .
- For a given input x , it is infeasible to find a second pre-image value x' where $x \neq x'$ and $H(x) = H(x')$.
- It is infeasible to find two inputs x and x' where $x \neq x'$ to produce same value as $H(x) = H(x')$.

2.3.1. Definiton. *A cryptographic hash function is a hash function defined as $H : \{0, 1\}^{m*} \rightarrow \{0, 1\}^n$ where for the given input of length m , the function outputs a fixed size n output such that the following property holds :*

- *Pre-image Resistance : for any given output $y \leftarrow \{0, 1\}^n$ i.e., $y = H(x)$, it is computationally infeasible to find $x \leftarrow \{0, 1\}^m$.*
- *Second pre-image Resistance : for any given input $x \leftarrow \{0, 1\}^m$ i.e., $y = H(x)$, it is computationally infeasible to find an input x' so that $x \neq x'$ but $H(x) = H(x')$.*
- *Collision Resistance : finding two inputs $x \leftarrow \{0, 1\}^m$ and $x' \leftarrow \{0, 1\}^m$ so that $x \neq x'$ but $H(x) = H(x')$ is computationally infeasible.*

There is no efficient way to invert back the output from the function to retrieve the input. This computational infeasibility in hash function, holds a shield against any cryptanalytic attacks. Generally, a hash function maps a larger domain set (input) $\{0, 1\}^*$ to a smaller co-domain (output) set $\{0, 1\}^n$, there is possible to find a pre-image value based on the birthday paradox concept. The Birthday Paradox is termed on a process of finding at least two people sharing same birthday in a group of n people. With the same analogy for the hash function, the probability of finding an equivalent pre-image and the likelihood of collision is high, for example $n = 64$ bit input results in 2^{64} different hash outputs then 4 billion (2^{32}) attempts of brute-forcing will break the system with the probability of 50% though 2^n pre-images are generated, $2^{n/2}$ collisions can be found. This is Birthday Attack

achieved by brute-force strategy, can be time-consuming for a classical computer. Hence, a Collision resistance hash function should have a long hash output than second pre-image resistance to achieve high security level.

2.4 One-Time Signature

The concept of the hash-based signature scheme are constructed on One-time signature (OTS) scheme. One-time scheme is a digital signature scheme which uses a collision resistant hash function to sign messages in $\{0,1\}^*$. OTS scheme generates an unique key pair (public key, private key) that can be used only once for sign and verify a message. Using the same private key to sign multiple messages would debilitate the security by half, as any probabilistic polynomial-time(PPT) adversary could compare two signatures from same private key and forge the signature from the revealed part of private key.

But by the definition of EU-CMA in Digital signature, a $(t, \epsilon(t))$ EU-CMA secure digital signature scheme is also an one-time signature that is $(t, \epsilon(t), 1)$, where the adversary is limited to query the signing oracle only once produce a valid signature different from one already queried by the oracle with probability $\leq \epsilon(t)$.

2.4.1. Definiton. *The One-time signature scheme (Gen, S, V) is (t, ϵ) existentially unforgeable under chosen message attacks(EU-CMA) if for all PPT adversaries A , the algorithm runs with a time complexity of at most t and generates only one query to the signing oracle. Then the probability of producing a valid signature for a message not previously sent to the signing oracle is $\leq \epsilon(t)$, a negligible function to break the scheme.*

$$Pr[Succ_{OTS(1^n)}^{EU-CMA}(n) = 1] \leq \epsilon(t)$$

Though OTS is highly efficient in generating key pair for sign and verify, the main limitation is their validity which is inadequate for most applications.

2.5 Many-time signature

As one-time signature scheme is EU-CMA secure, it can be extended from $\{0,1\}^n$ to sign many messages $\{0,1\}^*$ thereby new keys are generated to sign new message.

Stateful scheme Consider an one-time signature scheme $(Gen, Sign, Ver)$ where the length of the signature plus the length of the public key is less than the length of the messages to be signed.

- $Gen(1^n)$: Generates a key pair (pk_0, sk_0) to sign first message m_1 .

- $\text{Gen}(1^n)$: Generates a key pair (pk_1, sk_1) for next new message.
- Sign : Creates a signature $\sigma_1 = S_{sk_0}(m_1 \parallel pk_1)$, concatenation of message and the public key.

Hence, the signature of m_1 has $(1, \sigma_1, m_1, pk_1)$. Similarly for message m_2 , $\text{Gen}(1^n)$ generates a key pair (pk_2, sk_2) and the signature $\sigma_1 = S_{sk_1}(m_2 \parallel pk_2)$. Thus, the signature sequence of m_2 will be $(2, \sigma_1, \sigma_2, m_2, pk_2)$, where σ_1 is included as every signature sequence is attested by the next public key.

Though Many time signature scheme can sign many new messages with new key pair, the signature size increases linearly with respect to the message count. Also, the signer should keep track of the signature state information includes the previously generated keys, signature, signed messages count thereby increasing the cost of the storage and its is less efficient, as it reveals information about previously used signature. Hence, a new concept using a tree structure to allow two key pairs to be attested at one step was introduced rather than one key pair for each step. The tree construction shown in Figure 2.2 is a binary tree of height d , where each leaf node has one public-private key pair (pk, sk) and every non-leaf node has the hash of its child nodes. The tree uses 2^n leaf nodes to sign a message, generating 2^n signatures with a signature size of n .

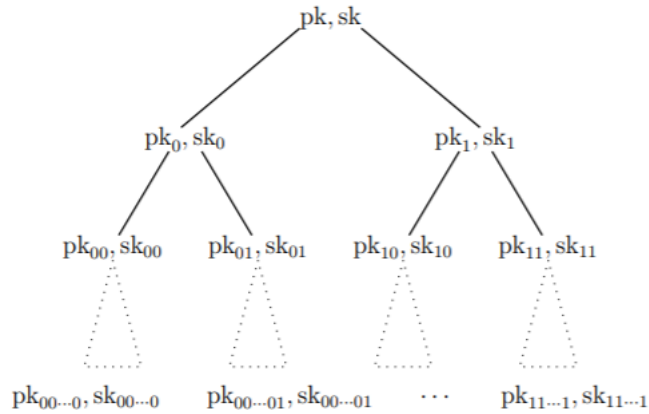


Figure 2.2: Balanced Binary Tree of height d [Pas]

In this approach, the signer initially generates n key pairs as $pk_0, sk_0, pk_{00}, sk_{00}, \dots, pk_{0^n}, sk_{0^n}$ and with other siblings they are stored in the tree. To sign a message m_0 , the public key pk_0 and pk_1 are signed with sk creating a signature σ_0 , pk_{00} and pk_{01} are signed with sk_0 creating a signature σ_1 likewise. Finally signature function $\text{Sign}_{sk_{0^n}}(m)$ and returns

a signature $\sigma = (pk, \sigma_0, pk_0, \sigma_1, pk_{00}, \dots, \sigma_{n-1}, pk_{0^{n-1}})$ and the verification function Ver , checks whether $Sign_{sk_{0^n}}(m)$ holds valid signature of m by verifying σ_0 attests to pk_0 using pk , σ_1 attests to pk_{00} using pk_0 and so on till pk_{0^n} . This scheme is one-time secure as every signature is used only once in an efficient way.

Stateless scheme The requirement to keep track of the previous keys and their signatures is a considerable drawback of the stateful schemes. In order to eliminate this requirement and primarily focusing on time-memory trade-off, a stateless scheme was proposed with an idea to use a pseudo-random function for regeneration of the keys every time. The algorithm $Gen(1^n)$ generates the key pair : public key, pk and secret key, sk . Along the secret key sk , two seeds are also generated s_1 and s_2 for two pseudo-random functions f and g . For node i , pk_i and sk_i is assigned by using $f_{s_1}(i)$ as a randomness. Similarly in signing algorithm $Sign_{sk_i}(m)$, random function $g_{s_2}(m)$ is used. Then, the scheme regenerates the authentication path of the trees without any state by the signer to be shared with the verifier. Hence, the Stateless scheme is a N -time signature scheme to sign up to N signatures but if more than N signatures are generated then the security degrades as stateless consumes more power with slower signature generation.

Chapter 3

Basics of Post Quantum Signature Scheme

Due to the depleting security in classical crypto-systems, cryptographers has to work on hard problems even for quantum attacks before the creation of well-versed quantum computers lead us to do research on “Post-quantum Cryptography”[BBD]. Though quantum-resistant alternatives includes lattice-based cryptography, code-based cryptography, multi-variate public key cryptography and hash-based signatures, we chose hash based signatures for the following reasons. They are constructed based on the definition of cryptographic hash function and its property explained in section 2.2.1. According to the “Grover’s algorithm”, though quantum computers could find pre-image of hash functions are faster than classical computers but the speedup process is less. For a quadratic computer to find a pre-image of a n-bit hash takes $O(2^{n/2})$ time but for a classical computer it is $O(2^n)$. So, on an average the Grover’s algorithm needs $\sqrt{2^n}$, then for n-bit hash output equivalent to $2^{n/2}$, which can be satisfied by just increasing the internal capacity and doubling the output size of the hash function. Hence, finding a pre-image takes longer for quantum system than a classical computer, makes hash-based signature safest signature.

In this thesis we expand the hash-based signature schemes for our comparative analysis of stateful LDWM and stateless SPHINCS⁺ algorithm. We will have a short overview of components used to built LDWM and SPHINCS⁺ in this section.

3.1 Hash-Based Signature Schemes

Hash based signature are fast and simple remarkably when compared to other cryptographic alternative proposed due to the evaluation of hash function and easy implementation in lightweight devices. The first hash based signature was developed by Leslie Lamport

[Lam16] in 1979 is described below.

3.1.1 Lamport One-time signature (L-OTS)

The scheme rely on one-way function, typically hash functions for their collision resistant property. The one-way function is defined as $F : \{0, 1\}^n \rightarrow \{0, 1\}^n$, efficient to compute but hard to invert without a trap-door setting.

Key pair Generation The private key of size $2m$ random bit-string $(sk_{i,0}, sk_{i,1})$ are generated as a sequence $(sk_{1,0}, sk_{1,1}, \dots, sk_{m,0}, sk_{m,1})$ to sign m bit string. Then, one way function F is applied to the private key to generate the public keys pk , as a sequence of $2m$ random bit string.

$$(F(sk_{1,0}), F(sk_{1,1}), \dots, F(sk_{m,0}), F(sk_{m,1})) = (pk_{1,0}, pk_{1,1}, \dots, pk_{m,0}, pk_{m,1})$$

Signature With key pair and message, we sign a message digest $M^* \in \{0, 1\}^m$, we divide the string into individual bits and sign it with their corresponding private key bits. i.e., for every message bit m_i , we map $sk_{i,0}$ with if $m_i = 0$ and $sk_{i,1}$ with if $m_i = 1$ then concatenate all the bits to form signature σ .

$$(sk_{1,M^*}, \dots, sk_{m,M^*}) = (\sigma_1, \dots, \sigma_m) = \sigma.$$

Verification The resultant signature σ , has half of the private key values. The verifier checks the validity of signature whether its mapped to the elements of public key using the function F :

$$(F(\sigma_1), \dots, F(\sigma_m)) \equiv (pk_{1,M^*}, \dots, pk_{m,M^*}).$$

This scheme is EU-CMA secure only if the key pair is used only once as defined in section 2.4.1. The downside is that if a same key pair is used to sign two different messages then both the signature will reveal parts of the private keys used and is vulnerable for signature forgery. For this reason, it is called “One-time signature”. In order to avoid forgery and extend OTS to sign multiple messages, a user should generate $2m$ key pairs to sign m bit message. Hence, the signature size and verification time increases with respect to the arbitrary number of signatures to be signed for messages. But, indeed creation of larger keys to sign multiple messages, would also gets exhausted in linear time proportional to the sign operations. Due to the storage issues, L-OTS was not used in practice.

3.1.2 Merkle Trees

Merkle extended L-OTS, to sign multiple messages by reducing the size of the public key to a single key using a tree structure concept. This innovation is a solution to eliminate large storage requirements of L-OTS. The tree structure is called “Merkle hash tree” [Mer82] patented in 1982. Merkle hash tree is a balanced binary tree with each node is a hash of its child node see Figure 3.1. The leaf nodes are the generated OTS public key and the root node of the binary tree is the main public key, so the verifier has to store only the root node rather than all the leaf node.

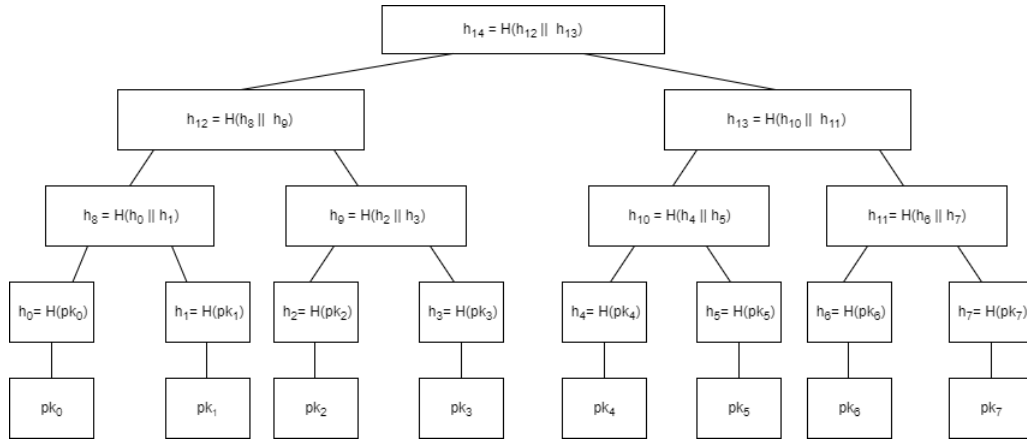


Figure 3.1: Merkle hash tree of height 3 based on [Mer82]

The tree structure eliminates the large storage requirements as for a possible N messages, a signer will generate N -OTS key pair, a $\log_2(n)$ value to create the Merkle hash tree. Every node is a hash of the child nodes and so, N public keys(leaf node) are compressed using a Hash function H which can be represented as OTS public key pk_i , a leaf node as $h_i = H(pk_i)$. The root node is the concatenation of the child nodes enabling the signer to have one root node thereby the Merkle hash tree is used as a many-time public key. To prove the authenticity of the signature generated has never used before, the signer applies a divide and conquer technique to add an authentication path to the signature. The Authentication path is a list of hashes to compute the root node. The Signer publishes a list of values consists of the index of the used leaf nodes(public key), signature of the secret key, leaf node(public key) value, authentication path from which the verifier computes the root node. For example as shown in Figure 3.2 : the signer use pk_3 for signing sig , then the path for verification sent to verifier include h_2, h_8, h_{13} to compute the root node. The signer publishes $(3, sig, pk_3, h_2, h_8, h_{13})$ so the verifier can compute from leaf progressing till the root node as $h_3 = H(pk_3), h_9 = H(h_2 || h_3), h_{12} = H(h_8 || h_9), h_{14} = H(h_{12} || h_{13})$,

indeed the signature sig is originated from the main public key.

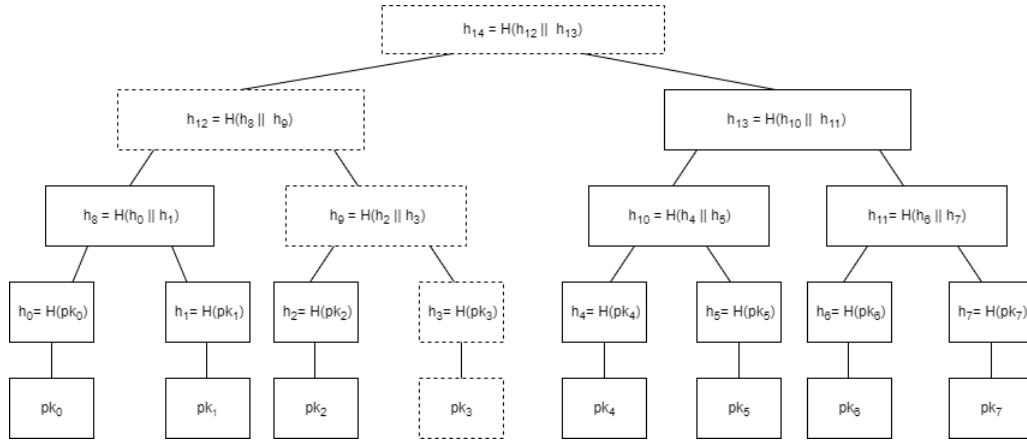


Figure 3.2: Merkle Hash tree representing Authentication path for public key pk_3 with dashed border nodes are computed for the verification.

Though it is not necessary for the verifier to know all the OTS public key to verify the main public key. For a signer to generate more signatures, he has to create large tree with key pairs for many-time signature. As per the computation cost with the time and space complexity, it increases with respect to the traversing through the large Merkle hash tree. Hence, Merkle has foreseen this issue and also proposed a deterministic way to generate private keys using pseudo-random generator and a seed value. This technique reduces the large storage issue as to short seed value to obtain many-time public key.

3.1.3 Winternitz One-time signature (W-OTS)

The W-OTS published in Merkle's paper [Mer89], is a modification of Lamport one-time signature for shorter signature and it was further analysed by Dods et al. [DSS]. The W-OTS is parameterised by the variable w , is a power of 2 denotes the number of bits to be signed simultaneously rather than sign-per-bit. The W-OTS scheme uses an one way function $f : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ and a hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^m$, as the basic idea is to apply the function f repeatedly 2^{w-1} times on an input $x \in \{0, 1\}^n$, to form a function chain like $f(f(f(x))) = f^3(x)$. The generation of signature for a message M of m -bit length using W-OTS is as follows with reference to [BDE⁺]:

Key Generation Let $n \in \mathbb{N}$ be a security parameter, Choose a Winternitz parameter $w \in \mathbb{N}$, $w \geq 2$ for compression level and an input value $x \in \{0, 1\}^n$.

The private key sk , for $i=0, \dots, l-1$ is list of randomly chosen values of length l with n bits

as

$$(sk_0, \dots, sk_{l-1}) \leftarrow \{0, 1\}^{(n,l)}$$

where l represents the number of m -bits values in an uncompressed private key, public key and signature, computed as

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

The public key pk , is computed by applying function f on every private key sk_i , $w-1$ times to form Winternitz hash function chain.

$$(pk_0, pk_1, \dots, pk_{l-1}) = (x, f_{sk_1}^1(x), f_{sk_2}^2(x), \dots, f_{sk_{l-1}}^{2^{w-1}}(x))$$

Signature Generation With the message digest $d = h(M)$, add necessary zeros to the left of the d such that the length of d is divisible by w . Then split the hash d , into l_1 binary blocks of size w resulting in $d = (m_0 || \dots || m_{l_1-1})$. We compute the checksum $c = \sum_{i=0}^{l_1-1} (2^w - m_i)$, add necessary zeros to the left of c so that the length is divisible by w . The resulted C string is split into l_2 blocks of size w as $c = (c_0 || \dots || c_{l_2-1})$. The signature is the concatenation of Message digest and checksum, where $b = (b_0 || \dots || b_{l-1}) = (m_0 || m_1 || \dots || m_{l_1-1} || c_0 || \dots || c_{l_2-1})$.

$$\sigma = (\sigma_0, \dots, \sigma_{l-1}) = (f_{sk_0}^{b_0}(x), \dots, f_{sk_{l-1}}^{b_{l-1}}(x))$$

Verification To verify the signed message (M, σ) , we calculate the checksum as same as it was calculated in signature generation step for constructing $b = (b_0 || \dots || b_{l-1})$. If the comparison holds $pk'_i = pk_i$ for $i=0, \dots, l-1$, then the signature is accepted.

$$(f_{\sigma_0}^{2^{w-1}-b_0}(pk_0), \dots, f_{\sigma_{l-1}}^{2^{w-1}-b_{l-1}}(sk_{l-1})) = (pk'_0, pk'_1, \dots, pk'_{l-1})$$

3.1.1. Example. For a message value "Hello" hashed using SHA-256 outputs a message digest "185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969", set $w=4$ where the message digest of 256-bit is split into bit string b_i of w length forming 64 Winternitz hash chain. The signer then publishes the public key as $f^{15}(sk_{16})$ and sign the 7th byte value(0111) with $f^7(sk_7)$. The verifier computes the signature of f^7 by computing $f^{15-9}(f^9(sk_9))$.

Security of W-OTS : Based on the above example 3.1.1, in order to create a signature, we reveal the intermediate values of the function chain of the public key as $15-9 = 7$ i.e.,

Any adversary eavesdrop the message and its signing nature, could modify the message or signature by increment of f accordingly to sign the next byte. Hence, Winternitz applied checksum to prevent attacker from modifying the signature. Also, iterating one-way function f on the message provides shorter signatures than L-OTS but increases the number of function applied from $1, \dots, 2^w - 1$ times. The larger w value, shorter signature and longer the signing, verification time makes W-OTS secure.

3.2 Lamport-Diffie-Winternitz-Merkle Scheme (LDWM)

This section describes the one-time signature scheme of the LDWM scheme in detail as proposed in draft-mcgrew-hash-sigs-02[MC]. The LDWM scheme is a draft version of Leighton-Micali Hash-based signature scheme (LMS)[MCF19] published in 2019. LDWM is a stateful scheme, a combination of one-time signature along with Merkle-Winternitz tree structure, allows to keep track(state) of the signatures generated.

Functions LDWM scheme uses OTS as the building block as explained in Section 2.4.1, also has two components H , collision-resistant hash function and F , one-way pre-image resistant function. The Hash function H , takes any message of arbitrary length (in bytes) as input and returns fixed n -byte value represented as $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$ whereas the one-way function F accepts m -byte string and outputs m -byte represented as $F^i : \{0, 1\}^m \rightarrow \{0, 1\}^m$. Let F be an i -folded iterative value for LDWM denoted as

$$F^i(x) = \begin{cases} x, & \text{if } i = 0 \\ F(F^{i-1}(x)), & \text{if } i > 0 \end{cases}$$

Parameters The security parameters of LDWM are m and n values, as they determine the byte size of the private key, public key and the signature. The Winternitz parameter w , is the number of bits of the message to be signed simultaneously rather than per-bit signature. Usually $w \in \{1, 2, 4, 8\}$, larger the w value few elements are included in the signature, providing short signatures but the key generation, signing and verification slows down. However, there is no impact on security but the value of w is considered to be a trade-off between the size of the signature to the computational effort. The parameters are illustrated in the Table 3.1 below.

Parameter	Description
m	The length in bytes of each element of an LDWM signature.
n	The length in bytes of the result of the hash function.
w	Winternitz parameter.
p	The number of m-byte string elements that make up the LDWM signature.
ls	The number of left-shift bits used in the checksum function C.

Table 3.1: Lamport-Diffie-Winternitz-Merkle(LDWM) Scheme Parameters.

Key pair Generation The LDWM private key denoted as x , is an array of size p containing m -byte strings. As the nature of the OTS, the private key can be used only once to sign one message. The pseudo code to generate the unique private key randomly is explained in Algorithm 1.

Algorithm 1 Generating a Private Key

```

for (  $i = 0; i < p; i = i + 1$  ) do
    set  $x[i]$  to a uniformly random  $m$ -byte string
end for
return  $x$ 
    
```

The LDWM public key is the hash value of the private key x , where each element of x is passed through the function F , $(2^w - 1)$ times and the resultant y , is hashed altogether. The function F is defined as $F^{(2^w-1)}$ and Algorithm 2 generates the public key.

Algorithm 2 Generating a Public Key From a Private Key

```

 $e = 2^w - 1$ 
for (  $i = 0; i < p; i = i + 1$  ) do
     $y[i] = F^e(x[i])$ 
end for
return  $H(y[0] || y[1] || \dots || y[p-1])$ 
    
```

Signature Generation In order to avoid forgery, signature generated is proofed with checksum function C , as proposed in Algorithm 3. The LDWM signature is generated by hashing the message using H and concatenate with the checksum value calculated. Then the resultant is split to the sequence of w -bit value and according to each bit value, the function F is applied with the corresponding private key. The output of F are concatenated as signature and the pseudo code is as follows in Algorithm 3 and Algorithm 4 :

Algorithm 3 Checksum Calculation

```

sum = 0
for ( i = 0; i < u; i = i + 1 ) do
    sum = sum + (2w - 1) - coef(S, i, w)
end for
return (sum « ls)                                     ▷ ls is the left-shift operation

```

Algorithm 4 Generating a Signature From a Private Key and a Message

```

V = (H(message) || C(H(message)))
for ( i = 0; i < p; i = i + 1 ) do
    a = coef(V, i, w)
    y[i] = Fa(x[i])
end for
return (y[0] || y[1] || ... || y[p-1])

```

Finally, the Signer provides the signature, message and the public key to the verifier for verification.

Verification The Signature is an array of m -byte of strings denoted as y where the verifier will use function F on w -bit string of message hash and checksum value computed as shown in the pseudo code below :

Algorithm 5 Verifying a Signature and Message Using a Public Key

```

V = (H(message) || C(H(message)))
for ( i = 0; i < p; i = i + 1 ) do
    a = (2w - 1) - coef(V, i, w)
    z[i] = Fa(y'[i])
end for
if public key is equal to H(z[0] || z[1] || ... || z[p-1]) then
    return 1                                           ▷ message signature is valid
else
    return 0                                           ▷ message signature is invalid
end if

```

With the hash value of resultant value z equal to the public key, then the message signature is considered as valid.

Security of LDWM : The concrete security quantifies the success probability of an adversary break the scheme using any of the security parameters value, running for a specific time. It focus on the increasing the difficulty level to prevent forging the signature ensure " k -bit security" i.e., for a large m, n value, in order to forge a signature an adversary should use $2^k, k$ bits. With reference to the security analysis[Kat16] of LDWM scheme, having Hash function H and Function F as independent random oracle which runs q -times and k -bits are calculated to ensure security for various scenario as explained below.

- Hash Collision : A signature can be forged if an adversary is able to find a collision. To avoid the Birthday attack, the output size of Hash function H should be at least $2k$ bits. In order to ensure k -bit security, the adversary has to run $O(2^k)$ computations to break the scheme i.e., then to break the hash output of $2k$ the adversary has to perform $O(2^{2k})$ computations, which is infeasible.
- Assume N instance of LDWM scheme run by the same signer or multiple signer, then there could be a chance of some hash value to be equal to the existing public key generated.

Consider any i^{th} public key $pk^i = H(y_0^i, \dots, y_{p-1}^i)$ formed by computing y from distinct x value as $y_0^* = F^e(x_0^*), \dots, y_{p-1}^* = F^e(x_{p-1}^*)$. For any of the x' value, hash is equal to the pk^i then it is vulnerable for forge signature with respect to the public key. The probability of success for this scenario is $q.N/2^n$. Hence, to ensure k -bit security the output size of Hash function H should be at least $k + \log N$ bits.

3.3 SPHINCS+

The basic concept of stateless scheme is to primarily eliminate the states in the signature. The stateful schemes like XMSS[HBG⁺18](eXtended Merkle Signature scheme), keeps track(state) of One-time key pairs used in generating signatures. If any key-pair state lost in an adversary's procession, then he will be available to predict a valid signature. In order to ensure security, stateless schemes called SPHINCS was proposed to eliminate state. SPHINCS[BHH⁺15] is 128-bit post quantum secure scheme constructed with Winternitz OTS and Merkle tree structure. The randomized leaf (index) selection from Merkle tree reduces the chance of choosing the same key-pair and also the use of hypertree where every node is considered to be a Merkle tree itself, increases the security level of SPHINCS. Thus, it is considered to be easy replacement for the existing cryptosystems but SPHINCS provides large signature, performing long computation. The upgraded version of SPHINCS with improvements was published as SPHINCS+[BHK⁺19].

3.3.1 Components of SPHINCS⁺ Framework

SPHINCS⁺ is a stateless hash-based signature scheme which was selected for second Round among the nine signature schemes submitted for NIST competition. Similar to LDWM, SPHINCS⁺ relies on the properties of cryptographic hash function. The SPHINCS⁺ hash tree is a hypertree structure consisting of multiple layers of hash trees to it. The Winternitz OTS keys signs the leaf nodes to the each root nodes of the lower level tree. A few-time signature FORS is used for signing the messages and the keys are in the leaf node of the lowest level of the Hypertree. This section describes SPHINCS⁺ components shown in Figure 3.3 in detail, as proposed in [BHK⁺19] with Winternitz OTS⁺, hypertree and Forest of Random subsets(FORS).

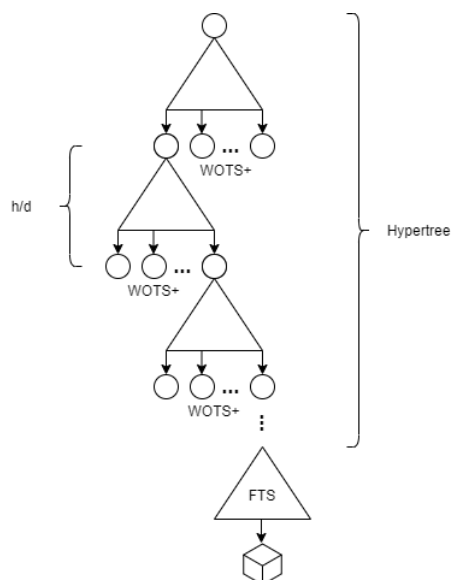


Figure 3.3: Hypertree Model of SPHINCS⁺

Winternitz-OTS⁺ (WOTS⁺)

It is an hash-based Winternitz Onetime-signature(OTS) variant proposed by Hülsing[Hül17] generates shorter signature and targeted to prevent Multi-target attack, refer section 3.1.3 for WOTS. WOTS⁺ has replaced hash function f , with keyed second pre-image resistant hash function $f_k: \{0,1\}^n \rightarrow \{0,1\}^n$, where $k(\text{key space}) \in \{0,1\}^n$, i.e., for every key k , the function f_k computes bitwise XOR of the input with a randomization element r as $f_k(x \oplus r_i)$ for any i , reducing the signature size by 50% at 80bits security level.

Key Generation Let $n \in \mathbb{N}$ be a security parameter, Choose a Winternitz parameter $w \in \mathbb{N}$, $w \geq 2$ for compression level and random elements $r \in \{0,1\}^{(n,w-1)}$. This algorithm uses an input seed s , to reduce the storage issue as every time unique key can be regenerated from the seed itself.

The private key sk , for $i=0, \dots, l-1$ is a n -byte value pseudo randomly generated from s as

$$(sk_0, \dots, sk_{l-1}) \leftarrow \{0, 1\}^{(n,l)}$$

where l represents the number of m -bits values in an uncompressed private key, public key and signature, computed as

$$l_1 = \lceil \frac{m}{\log_2(w)} \rceil, l_2 = \lfloor \frac{\log_2(l_1(w-1))}{\log_2(w)} \rfloor + 1, l = l_1 + l_2$$

The public key pk , is computed by applying function f on every private key sk_i , $w-1$ times to form Winternitz hash function chain.

$$(pk_0, pk_1, \dots, pk_{l-1}) = (x \oplus r, f_{sk_1}^1(x \oplus r), f_{sk_2}^2(x \oplus r), \dots, f_{sk_{l-1}}^{2^w-1}(x \oplus r))$$

Signature Generation With the message digest $d = h(M)$ and randomization element r , compute the base w representation of $d = (m_0 || \dots || m_{l_1-1})$, $m_i \in \{0, \dots, w-1\}$. Also, Compute the checksum $c = \sum_{i=0}^{l_1-1} (2^w - 1 - m_i)$, generate $c = (c_0 || \dots || c_{l_2-1})$, and append c of length l_2 resulting in $b = (b_0 || \dots || b_{l-1}) = (m_0 || m_1 || \dots || m_{l_1-1} || c_0 || \dots || c_{l_2-1})$. The signature is:

$$\sigma = (\sigma_0, \dots, \sigma_{l-1}) = (f_{sk_0}^{b_0}(x \oplus r), \dots, f_{sk_{l-1}}^{b_{l-1}}(x \oplus r))$$

Verification To verify the signed message (M, σ) , calculate the checksum as same as calculated in signature generation for constructing $b = (b_0 || \dots || b_{l-1})$. If the comparison holds $pk'_i = pk_i$ for $i=0, \dots, l-1$, then the signature is accepted.

$$(f_{\sigma_0}^{2^w-1-b_0}(pk_0 \oplus r), \dots, f_{\sigma_{l-1}}^{2^w-1-b_{l-1}}(pk_{l-1} \oplus r)) = (pk'_0, pk'_1, \dots, pk'_{l-1})$$

Hypertree

A single Merkle tree structure of height h is used to sign a message $N = 2^h$, 2^h public keys has to be generated as leaf nodes and hash it every two nodes to the root node for the public key. For a large tree of $h = 256$, the cost of the operation increases by time and size with respect to signing and verification operation as traversing from leaf to root for authentication path. A new construction of tree chaining was introduced in SPHINCS[BHH⁺15] called Hypertree. It is a tree of trees construction of height h , which serves as certification tree for verification and as WOTS+ key pair in the leaf node of each tree. The Authentication path and the signature from the leaf node to the root is attached

to the intermediate trees, so that the top-most root holds the complete authentication path. Also, the leaf nodes(WOTS+ public key) attached to the intermediate tree is deterministic in nature. During the key generation algorithm, the tree is generated virtually with height h and intermediate layers d as there is no dependencies among the roots of intermediate tree.

FORS

SPHINCS⁺ as a few-time signature scheme, uses Forest of Random subsets(FORS), a concept of decision trees defined in terms of k values and $t=2^a$, to sign ka bits of string.

Key pair Generation The private key is a group of k sets of t value each, are generated randomly from a secret seed using Pseudo-random function together form a kt n -bits value. The Public key is a k binary hash trees of height a with a t values of private key as elements.

Signature For message of ka bits, extract k strings for a bits, where each bit is assigned as a leaf node in each of the k binary hash tree(FORS). The Signature consists of the index of the nodes and the authentication paths.

Verification The verifier reconstructs the root node using the authentication path and uses the tweakable hash function Th_k to reconstruct the public key.

3.3.2 SPHINCS⁺

SPHINCS⁺ has two types of construction namely robust, based on XMSS and simple, based on LMS. The robust construction generates a pseudo random bit-masks which are later XOR-ed with the input whereas simple construction doesnot support bit-masks and so they are faster, reduction in compression calls when combined with compressed address. There are three instantiations of hash function for respective construction, are SHA256, SHAKE256 and Haraka. Further they are categorized into two as size-optimized represented as "s" and speed-optimized represented as "f" at the end of the signature method. For NIST submission, SPHINCS⁺ includes 36 instantiations, a combination of 3 security levels, 3 hash functions, simple and robust instants, speed and size optimization. Here we focus on SPHINCS⁺-SHA-256 with 128s,192s and 256s for a simple hash function.

Function SPHINCS⁺ use two pseudo-random functions(PRF, PRF_{msg}), a keyed function (F), a keyed hashed function(H_{msg}) defined and explained below.

3.3.1. Definiton. *The Pseudo-random function PRF , is function whose output is difficult to differentiate from any other functions within the same domain and range of the input.Hence, it is used in key generation. The Pseudo-random function PRF_{msg} is used in adding randomness to the message compression.*

$$\begin{aligned} PRF &: \{0, 1\}^n \times \{0, 1\}^{256} \rightarrow \{0, 1\}^n \\ PRF_{msg} &: \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^* \rightarrow \{0, 1\}^n \\ H_{msg} &: \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^n \times \{0, 1\}^* \rightarrow \{0, 1\}^n \end{aligned}$$

It also has a special function called Tweakable hash function Th , a function to hold specific information and public parameters to be added to the input.The keyed function and keyed hash function H_{msg} , process any length of messages for compression that are applied to special cases in tweakable hash function.

$$\begin{aligned} F &: \{0, 1\}^n \times \{0, 1\}^{256} \times \{0, 1\}^n \rightarrow \{0, 1\}^n \stackrel{\text{def}}{=} Th_1 \\ H &: \{0, 1\}^n \times \{0, 1\}^{256} \times \{0, 1\}^{2n} \rightarrow \{0, 1\}^n \stackrel{\text{def}}{=} Th_2 \end{aligned}$$

3.3.2. Definiton. *The Tweakable hash function is defined as a mapping of α -bit message M to an n -bit hash value.Let $\alpha \in \mathbb{N}$, P be the public parameters and T be the tweak parameter.*

$$Th : P \times T \times \{0, 1\}^\alpha \rightarrow \{0, 1\}^n$$

For SPHINCS⁺-SHA256 the functions are defined as follows:

$$\begin{aligned} H_{msg}(R, PK.seed, PK.root, M) &= MGF1 - SHA - 256(SHA - 256(R \parallel PK.seed \parallel \\ & \hspace{15em} PK.root \parallel M), m) \\ PRF(SEED, ADRS) &= SHA - 256(SEED \parallel ADRS^c) \\ PRF_{msg}(SK.prf, OptRand, M) &= HMAC - SHA - 256(SK.prf, OptRand \parallel M) \end{aligned}$$

The simple variant, the tweakable hash functions are defined as follows:

$$\begin{aligned} F(PK.seed, ADRS, M_1) &= SHA - 256(PK.seed \parallel toByte(0, 64 - n/8) \parallel ADRS^c \parallel M_1) \\ H(PK.seed, ADRS, M1 \parallel M_2) &= SHA - 256(PK.seed \parallel toByte(0, 64 - n/8) \parallel ADRS^c \\ & \hspace{15em} \parallel M_1 \parallel M_2) \\ Th_l(PK.seed, ADRS, M) &= SHA - 256(PK.seed \parallel toByte(0, 64 - n/8) \parallel ADRS^c \parallel M) \end{aligned}$$

The output of F, H, PRF, PRF_{msg} depends on the parameter set, if it demands the length $n < 256$ then the first n bits of the output is considered.

Parameters The parameters are the variables from the various components used in construction of SPHINCS⁺ illustrated in Table 3.2. Winternitz parameter w , is a trade-off between time and size in signature operation. Usually $w \in \{4, 16, 256\}$, larger the value is set, then smaller the signature and slower the signature generation.

Parameter	Description
m	The length of message digest in bytes
n	The length of SPHINCS ⁺ signature in bytes
h	The height of the hypertree
d	The layers count in the hypertree
k	The number of trees in FORS
t	The number of leave in FORS tree
w	Winternitz parameter.
PK.root	Top root node of n bytes length [Public]
PK.seed	Random public seed of n bytes length [Public]
SK.seed	Random private seed generates WOTS ⁺ of n bytes length [Private]
SK.prf	To randomized message digest of n bytes length [Private]
Th	Tweakable Hash Function

Table 3.2: SPHINCS⁺ Scheme Parameters.

Key pair Generation The public key is a set of two n -bit values, where the root node of the hypertree is generated using WOTS+, and a random public seed value $PK.seed$. The private key is a set of two n -bit values: $SK.seed$ to generate pseudo-random secret keys(WOTS+ and FORS) and $SK.prf$ to generate unpredictable index and randomize the message digest.

Signature The Signature in message $M \in \{0, 1\}^*$, is generated using pseudo randomizer represented as $R = \text{PRF}(SK.prf, \text{OptRand}, M)$, R is applied on the message and $SK.prf$. To sign the randomised message hash H_{msg} , the index of the leaf node is chosen using R . The index chooses the tree in the hypertree as well as the leaf node in the chosen tree for signing, as the message digest MD with idx can be represented as $(MD || idx) = H_{msg}(R, PK.seed, PK.root, M)$. Therefore, signature contains the index value idx , randomness R , FORS signature, one WOTS+ signature and authentication path per layer

of trees. The signature generation is deterministic in nature, and so randomness is quite important.

Verification To verify the SPHINCS⁺ signature, the verifier must compute the message hash H_{msg} , and verify the FORS signature, WOTS⁺ signature and authentication path per tree for root computations. By this the verifier will be able to compute the root node, and if it is equal to the public key, then the signature is valid.

Chapter 4

Analysis of LDWM and SPHINCS⁺

The Evaluation of the Algorithms basically focus on their implementation on Public key infrastructure either as quantum resistant certificates or cryptographic library over TLS that will be suitable for quantum era. The NIST evaluation criteria explained in section 2.2 is considered with the key parameters focusing on the classical and quantum security level(Security), size of the signature, public key and private key (Cost) and availability of code(Algorithm Implementation). Some of the other aspects considered are the category of Hash based schemes chosen: Stateful and Stateless and their maximum number of signatures limited in each scheme. This section briefs about the evaluation methodology and provides empirical evidence based on the Key generation time, signature generation and verification time for LDWM and SPHINCS⁺.

4.1 Implementation Details

Though the algorithms LDWM and SPHINCS⁺ differ from each other as stateful and stateless respectively, they share the OTS signature structure as a basic foundation for their scheme and also applies WOTS variants on the tree structure for signature, verification. We have implemented the LDWM algorithm in JAVA language with reference to the draft-mcgrew-hash-sigs-02[MC] is attached in the Chapter 6 implemented with the details explained in section 3.2. The original implementation of SPHINCS⁺ written in C can be found in [SPH], is utilised for the comparative study. We have considered SHA256 as hash functions, as it outputs 128,192 or 256 bit length, assuring the security level of 1,3 or 5 respectively. The parameter selection plays a role in time and space complexity in any applications, the study focus on implementing quantum hash-based signature in a Public key Infrastructure for the test parameters for LDWM and SPHINCS⁺ as given in Table 4.1 and 4.2 respectively.

Signature Method	H	F	m	n	w	p	ls
LDWM_SHA256_M32_W1	SHA256	SHA256	32	32	1	265	7
LDWM_SHA256_M32_W2	SHA256	SHA256	32	32	2	133	6
LDWM_SHA256_M32_W4	SHA256	SHA256	32	32	4	67	4
LDWM_SHA256_M32_W8	SHA256	SHA256	32	32	8	34	0

Table 4.1: Signature Methods for Testing LDWM code

Instantiations	n	h	d	log(t)	k	w	security level	public key	secret key
SPHINCS ⁺ -128s	16	64	8	15	10	16	1	32	64
SPHINCS ⁺ -192s	24	64	8	16	14	16	3	48	96
SPHINCS ⁺ -256s	32	64	8	14	22	16	5	64	128

Table 4.2: Signature Methods for Testing SPHINCS⁺ code

4.2 Key Attributes from NIST

4.2.1 Security

The available cryptosystems so far are susceptible to attacks in one or another, so the security level of the schemes built has to be estimated. The common way of security level estimation is using the notion of "bit security" i.e., any attack in the system requires $O(2^b)$ operations to break. For a quantum level cryptography, "quantum bit security" needs to be estimated for an adversary with quantum computer access. This Security assessment is based on the building blocks of an algorithm, security assumption they hold against known attacks, amount of parameters selections. From the NIST report[Bar20], the security level 1(128 bits) has to be achieved by the proposed submission. To achieve 128 bits security level with classical systems, use LDWM signature methods in Table 4.2 with $m=32$.

The implementation of LDWM generates one-time signature using Winternitz concept of grouping message bits according to Winternitz parameter and sign it. The Winternitz OTS along with Merkle tree, reduce the key size providing flexibility between signature size and generation time. But this implementation doesnot have Merkle tree construction. As an initial work of LMS, the LDWM rely on the one-way function F , to be pre-image resistant function and the Hash function H , to be collision-resistant. With Winternitz parameter $w=1$, LDWM is existentially unforgeable under an adaptive chosen message attack(EU-CMA) with c bit of Checksum mapped using F added to the hash message, making an adversary difficult to attempt forgery. The LDWM is post-quantum resistant with m and

n value is doubled so that it is protected against Grover's attack.

Similar to LDWM, SPHINCS⁺ is post-quantum existentially unforgeable under an adaptive chosen message attack(EU-CMA) using a single function F , second-preimage resistant and a tweakable Hash function Th , collision resistant. Both the functions in multi-target setting focus on mitigating multi-target attack. The Multi-target attack is where an adversary can invert the function to find the target value and if succeeds then he can attack factor of the number of targets.Hence, for mitigation the keyed hash function was used as for each hash call different key and bit-masks are used.The hash function with second-preimage property is used instead of one-way function i.e., even if an adversary accessing Quantum random oracle model and able to receive public parameters, it is difficult to find a second pre-image from the previous query for the same public parameter P and Tweak T . With FORS security, the adversary rights to chose the index, cannot learn or replace any secret and finding a message from a key set or indexes is impossible that the the tweakable hash function is generated one per hash call and selects a FORS key pair, so an adversary will not be able to find the weakest point in the key pair.

4.2.2 Cost/Performance Analysis

The Performance of the LDWM and SPHINCS⁺ code is analysed by generating several random messages given as input and the cost of signature, verification operation was measured on an Intel(R) Core(TM) i5-8250 CPU @1.60GHz 1.80GHz machine running Windows 10.

The LDWM code was developed in JAVA language in Eclipse IDE, with a capability to sign and verify for 100 random messages which is each of size 1 KB and tested with the signature methods as test parameters given in Table 4.2. We have used SHA-256 for Hashing function and also as one-way functions for the generation of public key, signature and verification. refer section 3.2 for Algorithms.

Time Taken			
Signature Method	Key Generation	Signature	Verification
LDWM_SHA256_M32_W1	29.7ms	5.44ms	5.89ms
LDWM_SHA256_M32_W2	28.31ms	7.15ms	7.74ms
LDWM_SHA256_M32_W4	35.11ms	13.25ms	14.37ms
LDWM_SHA256_M32_W8	167.57ms	73.54ms	78.45ms

Table 4.3: LDWM Time Analysis

The original implementation of SPHINCS⁺ written in C Language, for the thesis work we converted the C code to JAVA language using SWIG tool. The tool generates an

interface through which the C methods can be called. Same methodology of LDWM was used for SPHINCS⁺ as generating 100 random messages each of size 1KB was implemented and analyzed the running time taken for key pair, signing and verification for Table 4.2 test parameters.

Time Taken			
Signature Method	Key Generation	Signature	Verification
SPHINCS ⁺ _SHA256_128s	11.1ms	164.2ms	0.19ms
SPHINCS ⁺ _SHA256_192s	19.88ms	427.15ms	0.367ms
SPHINCS ⁺ _SHA256_256s	38.21ms	448.5ms	0.681ms

Table 4.4: SPHINCS⁺ Time Analysis

Based on the results given in Table 4.3 and Table 4.4, Each public key, private key and signature are calculated using the n parameter. For $n=32$, LDWM generates key and signature size of 64 bytes value whereas For $n=16,24,32$, SPHINCS⁺ generates $2n$ private keys and $4n$ public keys accordingly. Hence, we compare the LDWM_SHA256_M32_W8 with SPHINCS⁺-256s as they provide same key length of public keys, SPHINCS⁺-256s is faster in key generation and verification when compared to LDWM.

4.2.3 Algorithm

The goal of the Post Quantum cryptographic algorithm is to ensure the availability of its implementation either as a replacement or enhancement for the existing cryptosystems and also flexible enough to support public key encryption, key exchange, Signature and verification. In order to analysis the compatibility of SPHINCS⁺ and LDWM, we will discuss on their implementation in a real-time application. As though the algorithm is efficient and provide security, the algorithm should fit in the business requirements. The algorithm chosen are of Digital signature category, we chose Public Key Infrastructure to discuss more.

Every organisation relies on the security of their Infrastructure, as their operations involve in and out transactions of data, the identity of the sender and receiver are kept in compact using keys. Hence, to secure the keys involved in the every transaction, a wise way to keep in them in a form of certificates specifically as X.509 certificates. Thereby, the establishment and management of public keys ensures secure channel across a public network providing both authentication and encryption.

Public Key Infrastructure(PKI) provides certificates is considered to be digital passports provides identity of the participant with their public key, attested from a mutually

trusted third party claimed to be Certificate Authority(CA). The International Telecommunication Union(ITU) maintains a X.509 structure [XT19] as a standardization for public key certificates generation. A X.509 format [CP08] consists of three elements:

Components of X.509

- `tbscertificate` : The certificate to be signed.
- `signatureAlgorithm` : An identifier used by CA to sign the certificate.It also has Object Identifier and various other parameters for the algorithm to be identified.
- `signatureValue` : Digital Signature encoded as Bit string. Other components are version, serial number, signature, issuer(CA), Validity period, subject, subjectPublicKeyInfo.

The certificate binds the public key with the sender and identity is preserved, this process is done by the Registry Authority(RA).Once, the identity is verified, the RA sends confirmation to the CA to generate the signature and sign it using the CA's private key. The issued certificate can be verified by the CA's public key included in the CA certificate. The PKI has an hierarchical structure of CA's depicts as a trust model between sender and receiver. The root node has the *RootCA*, the leaf nodes are the identifiers and the other nodes form a certificate chain with intermediate CAs. For a transaction between two parties, the receiver will send a certificate chain to the sender consisting of his own certificate and all the other certificates that reach up to the RootCA. Thereby, the sender could verify the authenticity of the receiver and if the other nodes are interlinked with each other with valid signature, so the sender can trust receiver's certificate. Two signature schemes used for verification are RSA and Elliptic curve Digital Signature Algorithm. From [Bid06, p. 968], a Symmetric Key size of 128 bytes provides a security level of AES 128 equivalent to RSA 3072 with three parameters generating private key and public key approximately 1800 bytes and 390 bytes approximately. ECDSA algorithm NIST P-256 generates a signature size of 512 bits for $n = 256$, public key $2n = 512$ bits and private key as 768 bits. X.509 standard is widely used as it does not have any restrictions on the public key type and algorithm used for signature. This flexibility allows any length of signatures to be generated for the public keys. The PKI certificate issuance is used in TLS protocol on the Internet, to support confidential channel between client and server. For client authentication, client sends its certificate along with its certificate chain to the server for verification during the TLS handshake. For server authentication, all the certificates along the way to the root, certificate chain will be sent to the client for verification. Hence, the

amount of data transaction depends on the certificates containing signature's and public key's size that are transferred.

Discussion As X.509 certificates were already implemented in LDWM by the Industry, we focus on SPHINCS⁺ algorithm. In general, the size of a digital certificate depends on the algorithm used, key size and signature components. There is no upper limit set for X.509 certificate file so for a 512 byte private key, the size increases by 4/3, providing a size of 1420 bytes for an unencrypted private key. The size will increase for an encryption and the file containing private key is limited to hold less than 2048 bytes. SPHINCS⁺ is a good candidate for hash-based signature to be adopted by the Industry as it is available as API provides smaller signature size and has multiple variants to chose according to the developer's notions on size and speed. SPHINCS⁺ being a stateless with small key size can be easily maintained and transferred as certificates with minimal security assumptions with Hash function.

Chapter 5

Conclusion

As the Quantum threat rise in cryptography, it eventually affects the PKI. But, we are preparing ourselves with the research on post-quantum cryptography instructed by NIST. Several alternatives were chosen by NIST, to replace the existing system working with RSA and other classical schemes. Based on the LDWM and SPHINCS⁺ benchmark results, we found post-quantum digital signature scheme potentially can be used in Infrastructure. Thereby it forms as a recommendation for replacement of algorithms. The SPHINCS⁺ analyzed can be implemented on X.509 certificates and also as self-signed certificates can be viable option with available libraries based on the security, cost and implementation analysis. As the goal of the thesis, in conclusion we have provided the parameter set and the empirical results by testing LDWM, SPHINCS⁺ for a developer to consider while implementing an algorithm along with the security benefits.

Chapter 6

Reference

6.1 Source code

```
// Algorithm 0: Generating a Private Key
private static ArrayList<String> generate_private_key(Parameter p)
    throws NoSuchAlgorithmException {

String value = "";
for(int counter=0; counter < p.getParameter_p(); counter++)
{
// Random seeds are generated for private keys
byte[] seed = generatePRNG();
MessageDigest algorithm = MessageDigest.getInstance(p.getAlgorithm_F());
byte[] encodedhash = algorithm.digest(seed);
for (byte b : encodedhash) {
sb.append(String.format("%02x", b));
}
if(p.getAlgorithm_F().equals("SHA256-20"))
{
value = sb.substring(0, 20);
}
else {
value = sb.toString();
}
private_Key.add(value);
sb.delete(0, sb.length());
}
}
```

```
return private_Key;
}

// Algorithm 1: Generating a Public Key From a Private Key
private static ArrayList<String> generate_public_key(Parameter p,
ArrayList<String> private_Key) throws NoSuchAlgorithmException {
int parameter_e = (int) (Math.pow(2, p.getParameter_w()) - 1);
String result_0 = "";
String value="";
for(int counter = 0; counter < p.getParameter_p(); counter++)
{
String data = private_Key.get(counter);
for(int counter_e = 0; counter_e < parameter_e ; counter_e++)
{
MessageDigest algorithm = MessageDigest.getInstance(p.getAlgorithm_F());
byte[] encodedhash = algorithm.digest(data.getBytes());
for (byte b : encodedhash) {
sb.append(String.format("%02x", b));
}
if(p.getAlgorithm_F().equals("SHA256-20"))
{
value = sb.substring(0, 20);
}
else {
value = sb.toString();
}
data=value;
sb.delete(0, sb.length());
}
public_Key.add(data);
result_0 = result_0.concat(data);
}
MessageDigest algorithm = MessageDigest.getInstance(p.getAlgorithm_H());
byte[] encodedhash = algorithm.digest(result_0.getBytes());
for (byte b : encodedhash) {
sb.append(String.format("%02x", b));
}
```

```
}
//concatenated public key hash value
public_Key.add(0, sb.toString());
sb.delete(0, sb.length());
return public_Key;
}

// Algorithm 2: Checksum Calculation
private static String create_checksum(Parameter p, String data)
    throws NoSuchAlgorithmException {

    int parameter_e = (int) (Math.pow(2, p.getParameter_w()) - 1);
    int parameter_ls = 16 - (p.getParameter_v() * p.getParameter_w());
    int parameter_sum = 0;
    List<Integer> decimals = new ArrayList<>();
    MessageDigest algorithm = MessageDigest.getInstance(p.getAlgorithm_H());
    byte[] encodedhash = algorithm.digest(data.getBytes()); // Message hashed
    for (byte b : encodedhash) {
        sb.append(String.format("%8s", Integer.toBinaryString(b & 0xFF))
            .replace(' ', '0'));
    }
    String hashedData = sb.toString();
    //Split to w-bit values [Coeff(S,i,w)]
    decimals = coeff(hashedData,p.getParameter_w());

    for (int counter = 0; counter < p.getParameter_u(); counter++)
    {
        parameter_sum = parameter_sum + parameter_e - decimals.get(counter);
    }
    sb.delete(0, sb.length());
    String i = String.format("%16s", Integer.toBinaryString(parameter_sum))
        .replace(' ', '0');
    i = i.substring(parameter_ls,i.length()).replace(' ', '0');
    return i;
}
```

```
// Algorithm 3: Generating a Signature From a Private Key and a Message
private static ArrayList<String> create_signature(Parameter p,
ArrayList<String> private_Key, String data) throws NoSuchAlgorithmException {
String value="";String result_0 ="";
List<Integer> decimals = new ArrayList<>();
MessageDigest algorithm = MessageDigest.getInstance(p.getAlgorithm_H());
byte[] encodedhash = algorithm.digest(data.getBytes());
for (byte b : encodedhash) {
sb.append(String.format("%8s", Integer.toBinaryString(b & 0xFF))
                .replace(' ', '0'));
}
String hashedData = sb.toString();
sb.delete(0, sb.length());
//Create checksum
String C_H_message = create_checksum(p, data);
String parameter_V = hashedData.concat(C_H_message);
decimals = coeff(parameter_V, p.getParameter_w());
for(int counter=0; counter < p.getParameter_p();counter++)
{
int parameter_a = decimals.get(counter);
data = private_Key.get(counter);
for(int counter_a=0; counter_a < parameter_a ;counter_a++)
{
algorithm = MessageDigest.getInstance(p.getAlgorithm_F());
encodedhash = algorithm.digest(data.getBytes());
for (byte b : encodedhash) {
sb.append(String.format("%02x", b));
}
if(p.getAlgorithm_F().equals("SHA256-20"))
{
value = sb.substring(0, 20);
}
else {
value = sb.toString();
}
}
data=value;
```

```
sb.delete(0, sb.length());
}
created_signature.add(data);
result_0= result_0.concat(data);
}
return created_signature;
}
```

```
// Algorithm 4: Verifying a Signature and Message Using a Public Key
private static ArrayList<String> verify_signature(Parameter p,
ArrayList<String> public_Key, String data, ArrayList<String> created_signature)
throws NoSuchAlgorithmException {
String value="";String result_0 ="";
List<Integer> decimals = new ArrayList<>();
MessageDigest algorithm = MessageDigest.getInstance(p.getAlgorithm_H());
byte[] encodedhash = algorithm.digest(data.getBytes());
for (byte b : encodedhash) {
sb.append(String.format("%8s", Integer.toBinaryString(b & 0xFF))
.replace(' ', '0'));
}
String hashedData = sb.toString();
//Create checksum
String C_H_message = create_checksum(p, data);
String parameter_V = hashedData.concat(C_H_message);

//Split to w-bit values [Coeff(V,i,w)]
decimals = coeff(parameter_V,p.getParameter_w());

for(int counter=0; counter < p.getParameter_p(); counter++)
{
int parameter_a = ((int)(Math.pow(2, p.getParameter_w()) - 1))
- decimals.get(counter);

data = created_signature.get(counter);
for(int counter_a=0;counter_a < parameter_a;counter_a++)
{
algorithm = MessageDigest.getInstance(p.getAlgorithm_F());
```



```
encodedhash = algorithm.digest(data.getBytes());
for (byte b : encodedhash) {
sb.append(String.format("%02x", b));
}
if(p.getAlgorithm_F().equals("SHA256-20"))
{
value = sb.substring(0, 20);
}
else {
value = sb.toString();
}
data = value;
sb.delete(0, sb.length());
}
verified_signature.add(data);
//concatenated signatures
result_0= result_0.concat(data);
}
algorithm = MessageDigest.getInstance(p.getAlgorithm_H());
encodedhash = algorithm.digest(result_0.getBytes());
for (byte b : encodedhash) {
sb.append(String.format("%02x", b));
}
boolean flag = matching_signature(sb.toString(),public_Key.get(0));
sb.delete(0, sb.length());
return verified_signature;
}
```

Bibliography

- [Bar20] Elaine Barker. Recommendation for key management, part 1: General. <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf/>, 2020.
- [BBD] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmén. Post-quantum cryptography. OCLC: ocn297797507.
- [BDE⁺] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the security of the winternitz one-time signature scheme.
- [Ber09] Daniel Bernstein. Cost analysis of hash collisions: Will quantum computers make shares obsolete. 01 2009.
- [BH16] Leon Groot Bruinderink and Andreas Hülsing. "oops, i did it again" – security of one-time signatures under two-message attacks. Cryptology ePrint Archive, Report 2016/1042, 2016. <https://eprint.iacr.org/2016/1042>.
- [BHH⁺15] Daniel J Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O’Hearn. Sphincs: practical stateless hash-based signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 368–397. Springer, 2015.
- [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, CCS ’19*, page 2129–2146, New York, NY, USA, 2019. Association for Computing Machinery.
- [Bid06] Hossein Bidgoli. *Handbook of Information Security, Key Concepts, Infrastructure, Standards, and Protocols (Handbook of Information Security)*. John Wiley & Sons, Inc., USA, 2006.

BIBLIOGRAPHY

- [CJL⁺] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography.
- [CP08] Santesson S. Farrell S. Boeyen S. Housley R. Cooper, D. and W. Polk. "internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile", May 2008. <https://www.rfc-editor.org/info/rfc5280>.
- [DS07] Christos Douligeris and Dimitrios N Serpanos. *Network security: current status and future directions*. John Wiley & Sons, 2007.
- [DSS] C. Dods, N. P. Smart, and M. Stam. Hash based digital signature schemes. In Nigel P. Smart, editor, *Cryptography and Coding*, pages 96–115. Springer Berlin Heidelberg.
- [EC] Science News/ Emily Conover. Google claimed quantum supremacy in 2019 — and sparked controversy. <https://www.sciencenews.org/article/google-quantum-supremacy-claim-controversy-top-science-stories-2019-yir/>.
- [Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.
- [HBG⁺18] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018.
- [Her] Paul Hernandez. NIST general information. Last Modified: 2018-08-22T15:21:04:00 Library Catalog: www.nist.gov.
- [Hül17] Andreas Hülsing. Wots+ – shorter signatures for hash-based signature schemes. Cryptology ePrint Archive, Report 2017/965, 2017. <https://eprint.iacr.org/2017/965>.
- [Kat16] Jonathan Katz. Analysis of a proposed hash-based signature standard. In *International Conference on Research in Security Standardisation*, pages 261–273. Springer, 2016.
- [Lam16] Leslie Lamport. Constructing digital signatures from a one way function. 2016.
- [MC] David McGrew and Michael Curcio. Hash-Based Signatures. Internet-Draft draft-mcgrew-hash-sigs-02, Internet Engineering Task Force. Work in Progress.
- [MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, April 2019.

BIBLIOGRAPHY

- [Mer82] Ralph C. Merkle. Method of providing digital signatures., 1982. US4309569A.
- [Mer89] Ralph Merkle. A certified digital signature. volume 435, pages 218–238, 08 1989.
- [NIS] NIST/. Status report on the first round of the nist pqc standardization process. <https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf/>.
- [Pas] Rafael Pass. Digital signatures ii. <https://www.cs.cornell.edu/courses/cs687/2006fa/lectures/lecture20.pdf/>.
- [RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, pages 371–388, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [Sho75] Peter W. Shor. *Algorithms for Quantum Computation: Discrete Logarithms and Factoring*. 1975.
- [SPH] Software - github repository. <https://github.com/sphincs/sphincsplus/>.
- [SSD] SSD. A deep dive on end-to-end encryption: How do public key encryption systems work? <https://ssd.eff.org/en/module/deep-dive-end-end-encryption-how-do-public-key-encryption-systems-work/>.
- [Tha] Jay Thakkar. What is pki? a crash course on public key infrastructure (pki). <https://www.thesslstore.com/blog/what-is-pki-a-crash-course-on-public-key-infrastructure-pki/>.
- [TZ19] Teik Guan Tan and Jianying Zhou. A survey of digital signing in the post quantum era. *IACR Cryptol. ePrint Arch.*, 2019:1374, 2019.
- [XT19] X.LTU-T. "information technology–open systems interconnection– the directory: Public-key and attribute certificate frameworks". Cryptology ePrint Archive, Report 2016/1042, 2019. :<https://www.itu.int/rec/T-REC-X.509-201910-I/en>.