A COMPARATIVE STUDY BETWEEN IMPLEMENTATION OF POST-QUANTUM CRYPTOGRAPHY DIGITAL SIGNATURE SCHEMES ON SMART CARDS

DEPARTMENT OF COMPUTING SECURITY
GOLISANO COLLEGE OF COMPUTING AND INFORMATION SCIENCES
ROCHESTER INSTITUTE OF TECHNOLOGY

March 14, 2023

Tanishq Borse
Department of Computing Security
Rochester Institute of Technology
tb7223@rit.edu

Suvam Barui
Department of Computing Security
Rochester Institute of Technology
sb9895@rit.edu

1 Abstract

The advent of Quantum Computing (QC) brings challenges that make the current cryptography standards unsustainable. The present-day cryptosystems and algorithms such as RSA (Rivest-Shamir-Adleman), ECDSA (Elliptic Curve Digital Signature Algorithm), ECDH (Elliptic-Curve Diffie-Hellman), and DSA (Digital Signature Algorithm) are based on integer factorization and discrete logarithmic problems, and this poses a huge risk in the existence of quantum computing. Current cryptography algorithms implemented on Smart Cards are vulnerable to Quantum Computers. Research on Post-Quantum Cryptography hardware implementation on smart cards is very limited because of the constraints it brings with it. Smart Cards use limited computational resources and memory, so implementing PQC algorithms on smart cards would be a challenge but would prove to be a very resourceful finding. We explore two PQC digital signature schemes: One lattice-based (CRYSTALS - Dilithium) and another based on stateless hash functions (SPHINCS+). We compared both the algorithms in terms of their signature size and execution times of each operation. In addition to the comparison we also propose a smart card environment design where all the computational overhead is transferred to the card reader and other components instead of the Smart Card.

2 Introduction

In 2022, there will be approximately 5.03 billion internet users, which makes up 63.1 percent of the World's population. This shows the degree of involvement and dependence humans have on the internet [1]. Cryptography standards have been employed for safe operations on the internet since 1973, when the US first adopted it as the national standard for encryption. Since then, cryptography implementations have come a long way.

2.1 Motivation

Asymmetric-key techniques such as the Diffie-Hellman key exchange protocol and DSS are implemented as industry standards today. These incorporate algorithms such as DSA, Elgamal, Elliptic-Curve Cryptography (ECDSA, ECDH, EdDSA (Ed25519 Ed448), ECDH/EdDH (X25519 X448)) and RSA.

RSA is a widely used public-key cryptosystem that has been around since March 1991 (Version 1.1-1.3) which was implemented through Public-Key Cryptography Standards (PKCS) by RSA Laboratories, which included all the protocols to use and implement RSA for public-key cryptography [2]. It incorporates the real-world problem of factoring the multiplicative product of two large prime numbers [3]. On February 28, 2020, Number field Sieve Algorithm factored RSA-250 comprised 250 decimal digits (829 bits), making it insecure. [4]. ECC (Elliptic Curve Cryptography) is another widely used approach to public-key cryptography, which leverages the in-feasibility of finding the discrete logarithm of a random elliptic curve element where the base point is publicly known.

Present-day cryptosystems rely on the complexity of integer factorization and discrete logarithm problems. It is evident that with the advent of the era of Quantum Computing, the current standards for cryptography will only sustain briefly when Quantum Computers are available or even mainstream. Quantum Computers are commercially available but are in their infancy; therefore, it will take a few more years for them to go mainstream.

They offer computational operations that can harness the phenomena of Quantum Mechanics: superposition, interference, and entanglement. With the number of increasing qubits in a Quantum Computer, the possibilities of its applications are endless. The standard cryptography algorithms would no longer be secure due to a quantum computer's capabilities.

2.2 Background

In the year 1994, American mathematician Peter Shor came up with a Quantum Computing Algorithm for finding prime factors of an integer which can be leveraged to compromise cryptography algorithms which use the problem of factorising product of two large prime numbers. It approximately takes quantum gates of order $O((logN)^2(loglogN)(logloglogN))$ using fast multiplication and other such techniques to solve integer factorization efficiently with the help of a Quantum Computer in polynomial time [5]. Another algorithm which brought about a second major breakthrough in Quantum Computing is Grover's Algorithm also known as Quantum search algorithm. This algorithm finds the input of a black box function with a high probability with only $O(\sqrt{N})$ function evaluations which cannot be solved in classical computation, where N is the function domain[6].

Asymmetric algorithms such as RSA and ECC are compromised by using shor's algorithm. ECC-256 has 128 bit security in classical computation world but with quantum computers ECC-256 has almost 0 bit of security. The security for symmetric cryptography is reduced by 50 percent with the use of Grover's algorithm. AES-128 has only 64 to 80 bits of security in the quantum computing world [7]. Shor's algorithm requires 4000 qubits to break 2048-bit RSA Keys [8]. Quantum Computers currently capable of running shor's algorithm have only about 20 logical qubits (72 physical qubits). It is also important to be aware of latest Quantum Computers such as IonQ Aria which achieved a record 20 algorithmic qubits which is considered as the most powerful Quantum Computer as of 2022 and can actually be utilized for industry standard applications. In addition to IonQ Aria, D-WAVE2000Q with 2048 qubits which is based on the technology of quantum annealing cannot be used to run shor's algorithm [5] [9] [10]. Therefore, algorithms such as RSA, ECDSA, ECDH and DSA which are industry standards would have to be replaced by post-quantum cryptography (PQC) alternatives due to the exponential growth of Quantum Computer hardware. Figure 1 gives a summary of the algorithms which are no more secure.

Cryptographic Algorithm	Type	Purpose	Impact from large-scale quantum computer
AES	Symmetric Key	Encryption	Larger Key Sizes Needed
SHA-2, SHA-3	----------	Hash Functions	Larger Output Needed
RSA	Public Key	Signatures, Key Establishment	No Longer Secure
ECDSA, ECDH (Elliptic Curve Cryptography)	Public Key	Signatures, Key Establishment	No Longer Secure
DSA (Finite Field Cryptography)	Public Key	Signatures, Key Establishment	No Longer Secure

Figure 1: Presents the impact from large-scale quantum computers on the standard present day cryptosystem algorithms

The National Institute of Standards and Technology is working towards standardizing proposed post-quantum cryptography algorithms in the near future [11]. NIST received

around 69 algorithms for the standardization process of Post-Quantum Cryptography in the year 2017. They started the evaluation of these algorithms internally and externally, and finally proposed to standardize CRYSTALS-Kyber for Public Key Encryption, CRYSTALS-Dilithium, Falcon, SPHINCS+ for Public Key Digital Signatures after the third round of evaluation [12].

2.3 Research Questions

- What computational resources does Dilithium require to function?
- What computational resources does SPHINCS+ require to function?
- Can both algorithms (Lattice Based and Stateless Hash Function based) be implemented on a smart card? If so then what are the computational requirements to process these algorithms on a smart card ?
- How does the implementation of a lattice based algorithm differ from a stateless hash based algorithm, constrained to the same type of device, in terms of performance (Execution time and CPU Clock Cycle Count)?

2.4 Objective

This paper delves into the research and implementation of Digital Signature based authentication on Smart Cards. Smart cards are very constrained with resources which makes it a difficult task to accomplish our goal. We explore the algorithms for Public Key Digital Signatures. For diversity, we have included the CRYSTALS-Dilithium and SPHINCS+. Dilithium uses the hardness of lattice problems over module lattices, whereas SPHINCS+ uses the strength of stateless hash functions. In this paper, we will compare the feasibility of these two algorithms in the smart card industry. The comparison metrics will be speed, operability, ease of implementation, and complexity on a constrained device. We will also determine approaches to implement these digital signatures using current Smart Cards specifications for authentication in high-security areas.

This research will help us understand Post-Quantum Cryptography algorithms and how they can be implemented in smart cards and also open dimensions of improvement and help with future prospects and implementations.

3 Literature Review

There has been extensive research on implementation of PQC schemes on hardware architectures. Smart cards have an embedded integrated chip which stores the cryptographic scheme needed to establish security and therefore has limited memory and computational power to work with, due to this reason only few papers exist which study the implementation of PQC schemes on smart cards [13]. Figure 2 shows the maturity of cryptosystems.

3.1 CRYSTALS - Dilithium

CRYSTALS-Dilithium is a digital signature scheme which utilizes the hardness of lattice-based computational problems over module lattices such as the Shortest Vector Problem (SVP) and the Ring Learning With Errors (RLWE) problem. This means that an adversary

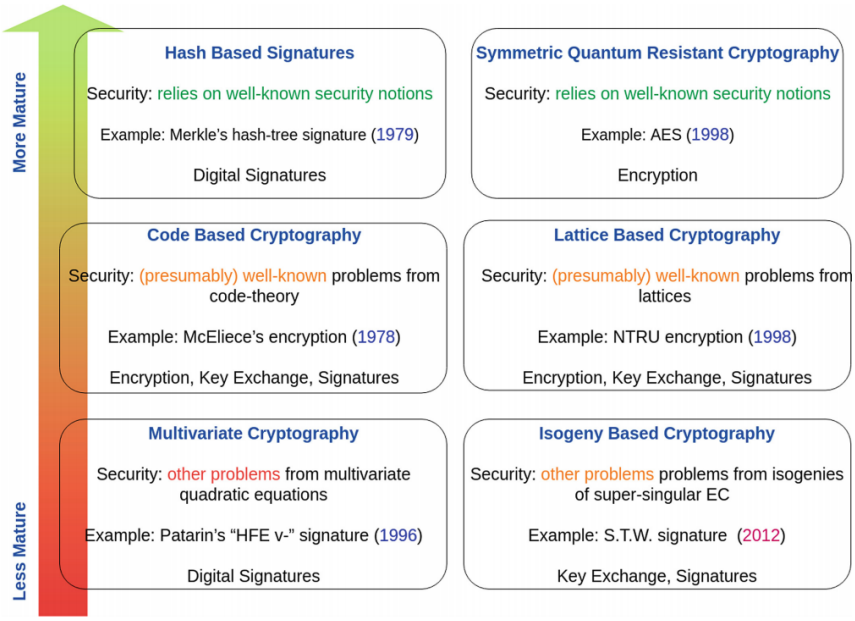

Figure 2: Represents the maturity graph of cryptosystems of different natures

who has access to a signing oracle won't be able to produce a signature of a message whose signature he hasn't come across yet, nor produce a different signature of a message that is already signed. A lattice L $\subset R^n$ is defined as the set of all integer linear combinations of basis vectors. Public key encryption, key exchange and digital signatures are usually implemented using Lattice-based cryptography. CRYSTALS-Dilithium has been finalized for the Public Key Digital Signature Scheme. Dilithium is designed on the basis of "Fiat-Shamir with Aborts [14]" technique of Lyubashevsky which makes lattice-based Fiat-Shamir schemes compact and secure using rejection sampling. Dilithium is said to have the smallest public key + signature size of any lattice-based signature scheme that only uses uniform sampling [15].

3.1.1 Algorithm Description

Dilithium avoids Gaussian sampling and uses uniform distribution to achieve an optimized public key size with the use of Fiat-Shamir with aborts [14]. It incorporates secure implementation against side-channel attacks and possesses efficiency which can be compared to the best lattice-based signature schemes in practice. Dilithium's security is proven in the Random Oracle Model (ROM), based on the hardness of two problems: (i) standard LWE and (ii) SelfTargetMSIS [16]. Several literatures exist which provide evidence of Dilithium being secure in the Quantum Random Oracle Model (QROM) [17],[18]. Dilithium has strong unforgeability under Chosen Message Attacks (SUF-CMA) secure in the ROM, with a non-tight reduction[19]. It has four variants having four security levels presented in Figure 2, three of these correspond to NIST security levels 1,2, and 3. Dilithium has components such as key generation, signature generation and signature verification which will be discussed consequently.

3.1.2 Key Generation

In the Key Generation algorithm, a private key or secret key is produced for signature generation and a public key is produced for signature verification. AES is integrated in the algorithm to create seeds $\rho, \rho^{'}$, and a key. Key Generation expands a (K x L) - size public matrix A using ρ. The matrix expansion is implemented by either AES or SHAKE depending upon the Dilithium variant. Matrix A entries have each a polynomial ring $R_q = Z_q[X]/(X^n + 1)$ where $q = 2^{23} - 2^{13} + 1$ and n = 256. $\rho^{'}$ and *nonce* are input to shake256 for the creation of vectors $s_1 and s_2$ with a uniform distribution. Many such operations occur within the algorithm and ρ and t_1 values are packed to create public key (pk). The shake256 function outputs a value tr with inputs of $pk, \rho, key, tr, s_1, s_2$, and t_0 which are combined to create secret key (sk). Therefore, Key generation produces the secret key and public key.

Algorithm 1: CRYSTALS-Dilithium: Key Generation

Data: None

Result: Secret Key $sk = (\rho, key, tr, s_1, s_2, t_0)$ and Public Key $pk = (\rho, t_1)$

procedure KEY GENERATION
$nonce \leftarrow 0$
Generation of seed (ρ, $\rho^{'}$, key) using AES
Expand matrix A from ρ.
Generation of short vectors s_1 and s_2 from $\rho^{'}$
$s_1 \leftarrow NTT(\mathrm{s}_1)$.
for $(k = 0; k < lengthof s_2; k++)$ **do**
    $t \leftarrow A * s_1$
    Reduce the coefficient of t.
    $t \leftarrow \mathrm{NTT}^{-1}(\mathrm{t})$.
end
$t \leftarrow t + s_2$
Reduce the coefficient of t.
$(t_0 + t_1) \leftarrow t$
$pk \leftarrow (\rho, t_1)$
$tr \leftarrow shake256(pk)$
$sk \leftarrow (\rho, key, tr, s_1, s_2, t_0)$
return(sk, pk)

3.1.3 Signature Generation

A message and secret key are used to produce a signature using the signature generation algorithm. Seed values (ρ, key) , $tr, s_1, s_2, and t_0$ are extracted from the secret key sk. The message m is copied in the last mlen entries of signature (sm). Using tr and m as input $shake256$ produces a collision resistant hash μ. Similar operations such as key generation are performed to expand matrix A using seed ρ. Several values are generated using complex mathematical operations which are ω , z, h, c and a hint vector n. The hint vector n is equated to ω to check whether the signature satisfies all the conditions and finally outputs signature (sm).

Algorithm 2: CRYSTALS-Dilithium: Signature Generation

Data: Message m, Message Length $mlen$, and Secret Key $sk = (\rho, key, tr, s_1, s_2, t_0)$
Result: Signature $sm = (z, h, c)$, Signature Length $smlen$

procedure SIGNATURE GENERATION
$nonce \leftarrow 0$
Extract $\rho, key, tr, s_1, s_2, t_0$ from secret key
Store the input message at the end of sm.
$\mu \leftarrow CRH(tr, m)$
Expand matrix A from ρ
$(s_1, s_2, t_0) \leftarrow NTT(s_1, s_2, t_0)$
while *1* **do**
    Generate intermediate vector y
    $w \leftarrow A * y$
    Decompose $w, (w_1, tmp) \leftarrow w$
    $c \leftarrow Hash(\mu, w_1)$
    $z \leftarrow y + c * s_1$
    if $(z > \gamma_1 - \beta)$ **then**
        Continue
    end
    $w^{'} \leftarrow w - cs_2$
    Decompose $(w^{'}, wcs20, tmp) \leftarrow$w$^{'}$
    Reduce the coefficient of $wcs20$.
    if $(wcs20 > \gamma_2$ - $\beta)$ || $(tmp! = w_1)$ **then**
        Continue
    end
    $ct_0 \leftarrow c * t_0$
    if $(ct_0 > \gamma_2 - \beta)$ **then**
        Continue
    end
    $tmp \leftarrow wcs_2 + ct_0$
    Reduce the coefficient of tmp.
    $n \leftarrow (tmp, -ct_0)$
    if $(n! = w)$ **then**
        Continue
    end
    $sm \leftarrow (z, h, c)$
    $smlen = mlen$ +CRYPTO_BYTES
    return(sm)
end

3.1.4 Signature Verification

Lastly comes the Signature Verification which verifies whether the signature is from an authenticated user having the appropriate signing, secret key. Components of the secret key (sk) and public key (pk) are extracted which are (z, h, c) and (ρ, t_1) respectively. The components $\rho, t_1 and m$ are input to collision resistant hash ($shake256$) to produce μ. Again the matrix A is expanded using seed ρ. From μ w_1, the array cp is created. cp is matched against c to check whether signature needs to be accepted or rejected. The last $mlen$

entries of signature (sm) are copied to message(m) and the signature us "Accepted".

Algorithm 3: CRYSTALS-Dilithium: Signature Verification

Data: Signature $sm = (z, h, c)$, Message = m, Signature length = $smlen$ Public Key $pk = (\rho, t_1)$

Result: "Accept"/"Reject", Message m, Message Length $mlen$

procedure SIGNATURE VERIFICATION
if $smlen < CRYPTO\_BYTES$ **then**
    return "Reject"
end
$(\rho, t_1) \leftarrow pk$
$(z, h, c) \leftarrow sm$
if $z > (\gamma_1 - \beta)$ **then**
    return "Reject"
end
$\mu \leftarrow CRH(CRH(\rho, t_1), m)$
Expand matrix A.
$tmp1 \leftarrow A * z$
$tmp2 \leftarrow c * t_1$
$tmp1 \leftarrow tmp1 - tmp2$
$w_1 \leftarrow (temp1, h)$
$cp \leftarrow H(\mu, w_1)$
if $(cp! = c)$ **then**
    return "Reject"
else
    $m \leftarrow sm[CRYPTO\_BYTES]$
    return "Accept"
end

3.2 SPHINCS+

SPHINCS+ is a post-quantum digital signature submitted to the NIST, which is a modified version of the stateless hash-based digital signature SPHINCS. Its main aim was to generate 2^{64} signatures per key pair in a reasonable amount of time and use the same public key to verify the digital signature to be eligible for the round 2 requirements of the NIST competition. The authors worked on the design of SPHINCS, which could only generate 2^{50}signatures and had many shortcomings. SPHINCS could be modified to generate 2^{64} signatures, but it would significantly hamper the performance of the digital signature. The design of Sphincs+ is modified based on recent cryptography studies and attacks with features such as multi-target attack protection, forests of random sets (FORS), improved HORST, and tree-less WOTS+ along with the use of tweakable hash functions for public-key compression. Given enough time, an attacker can generate a message and signature pair for classical computation, but SPHINCS+ is an Existential forgery secure scheme, and the strength of its security depends on the properties of the used hash functions such as SHA, HARAKA, and SHAKE.[20].

This report compares the hash-based signature scheme SPHINCS+ with the lattice-based signature scheme Crystalls Dilithium on their performance theoretically for smart-card implementation. It is essential to gain a better understanding of these schemes. Therefore,

in the sections below, we will see brief literature on hash-based signatures, SPHINCS, and SPHINCS+. The explanation is provided by looking at the different components that SPHINCS consists of and discussing how key generation, signature formation, and verification work.

The hash-based signature scheme cannot be comprehended as it uses a hash function. These properties are secure even if we use the MD5 algorithm.[21]. Based on the work of other authors, the Collision attacks are not enough to break the security of the hash functions assuming that the Pseudo-Random Function is within the limits of the tweakable hash function. [20]. SPHINCS+ is a non-lattice hash-based digital signature with many instantiations that can support a large signature size and fast generation. We will check the feasibility comparison of SPHINCS+ for quantum secure authentication on smart cards.

SPHINCS: Practical Stateless Hash-Based Signatures

Researchers introduced a practical method to perform stateless hash-based signatures in this paper that can sign messages comparatively faster with a small signature size. The SPHINCS-256 version generated signatures as small as 41KB using a public key size of 1,056 bytes and a secret key size of 1,088 bytes, which can provide up to 2^{128} bit security in the quantum computing era. The authors realized that stateful hash-based digital signatures had a fundamental flaw. The scheme was stateful, which meant that given a secret key along with the message, the scheme generated Hash signatures. However, it also generated an updated secret key it. The compromise of such an updated secret key could directly lead to an attack on the security of the signature scheme. Moreover, The state of the OTS tree is stored, as we use each one-time signature only once, which increases the size of the signature and big messages take forever to sign.

The authors used OTS in a binary certificate tree where the secret keys are generated pseudo-randomly on the go. They chose the index pseudo-randomly to make the scheme stateless and used a few time signatures (FTS) to sign the message, which reduced tree height and allowed a few collisions. It uses the few-time signature about 7-8 times until the security completely depreciates. The security of the few-time signature decreases after every use.

SPHINCS works on a one-time signature binary certificate tree with height h divided into d layers. In SPHINCS-256, the height is h is 60 with 12 layers. Each of these subtree leaves is a WOTS+ key. The leaves sign the root nodes on the underlying layers with the WOTS+ key, and HORST keys are signed using the leaves on the underlying layer. A HORST key finally signs the message.

Although this digital signature scheme decreased the signature size using the few-time signature scheme, it could only generate 2^{50} signatures per key pair, due to which it could not qualify for the requirements for the NIST competition for post-quantum digital signatures.[21]

The SPHINCS+ Signature Framework

This paper introduces SPHINCS+ as a stateless hash-based signature framework for post-quantum digital signatures. It improved the existing stateless hash-based digital signature SPHINCS and has considerable advantages over speed, signature size, and security. The significant contributions of the paper include the new few-time signature scheme called FORS, along with tweakable hash functions. Using such secure parameters

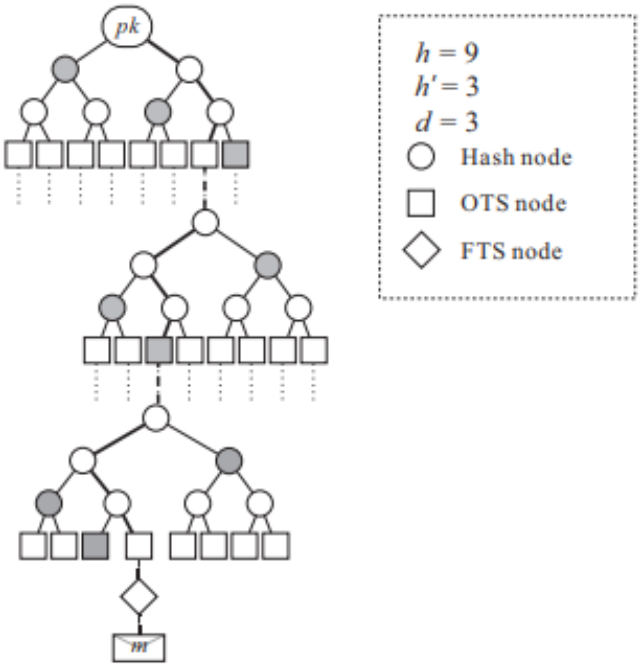

Figure 4: Sphincs+ Structure [22]

makes security reductions possible and allows SPHINCS+ to be an eligible candidate for the NIST competition for post-quantum digital signatures. The SPHINCS+ generates 2^{64} signatures per key pair and allows using the same public key to verify the digital signature.

WOTS+ is a result of extensive modification of WOTS, and it has several versions with slight differences. WOTS+ used in SPHINCS+ is very different from the original version. It introduces two additional parameters which turn the hash function into a tweakable hash function. The two parameters are n and w. n is the message length and the length of a private key element, a public key element, and a signature element in bits, which acts as a security parameter. w is the Winternitz parameter, which can trade off signature generation time and signature size. The WOTS+ secret key is derived from the SPHINCS+ private key, and the WOTS+ key air address is derived from the PRF of the hypertree.

SPHINCS+ includes protections against such multi-target attacks by use of tweakable hash functions. It can manipulate the length of the message to process arbitrary lengths of a message. It uses a new few-time signature known as forests of random subsets (FORS). It allows choosing a keypair independently from different pools, thereby reducing the attack target and providing multi-target protection. SPHINCS+ signature scheme has signatures that are 25 percent shorter than SPHINCS-256 signatures and has a 1.7 times faster-signing routine than SPHINCS-256.

Sphincs+ has many instances using the SHA-256, SHAKE-256 and HARAKE-256 that involve simple and fast versions which basically trade off signature size with the signing speed. The below table shows the parameters and results of various instances of sphincs+.[22]

Parameters	LEVEL 1 SPHINCS+-128f	LEVEL 1 SPHINCS+-128s	LEVEL 3 SPHINCS+-192f	LEVEL 3 SPHINCS+-192s	LEVEL 5 SPHINCS+-256f	LEVEL 5 SPHINCS+-256s
NIST LEVEL	1	1	3	3	5	5
n	16	16	24	24	32	32
h	60	64	66	64	68	64
d	20	8	22	8	17	8
Log t	9	15	8	16	10	14
k	30	10	33	14	20	22
W	16	16	16	16	16	16
Signature size (bytes)	16976	8080	35664	17064	49216	29792
Public Key size (bytes)	64	64	96	96	128	128
Private Key Size (bytes)	32	32	48	48	64	64

Figure 5: Sphincs+ instantiations Size

3.2.1 Algorithm Description

Treehash Algorithm

The Treehash function generates the authentication path and the root node of the sub-tree. It is the building block of WOTS+ and FORS. The Tree Hash Algorithm generates the internal node using a secret seed, a public seed, an index of the leaf, an index offset for FORS, the target node height, and an address of the tree. The loop iterates twice the tree height. A leaf node generates for each loop in the tree, and it sets the height of the leaf node to 0 and increases the offset. Suppose two leaves get added, The While loop generates nodes for the tree. Each pair of two leaves requires a parent node in a binary tree. We need to set an address to create the node. It generates the hash of the top node of the stack by taking two nodes as input. The height of the new node is higher than the leaf node. The node is stored if it is in the authentication route. This iteration occurs until we reach the top root node. [20]

Algorithm 4: Sphincs+: Treehash

Data: SK_SEED, PUB_SEED, leaf_idx, idx_offset, tree_height, tree_addr
Result: SK = (root, auth_path)

procedure TREEHASH
t $\leftarrow$ *tree_height*
for *(idx = 0;idx* $< 2^t$*; idx++)* **do**
    Add the leaf node to the root
    offset $\leftarrow offset + 1; heights[offset - 1] \leftarrow 0;$
    Copy the root node as authenticating path if(leaf_idx *xor* 1 == idx)
    while *offset* $\geq 2heights[offset - 2]$==$heights[offset - 1]$ **do**
        Set the address of the new node
        Hash the top node
        offset $\leftarrow offset - 1; heights[offset - 1] \leftarrow heights[offset - 1] + 1;$
        Store for authenticating path if required
    end
end

Key Generation
The Key Generation algorithm creates the public and secret keys. The public key comprises an n-byte public seed value and the root of the tree of the top layer. The secret key includes a seed to generate WOTS+ and FORS private key elements, a pseudo-random function (PRF) key, and a public key. AES generates the random number for the secret, public, and pseudo-random keys. The sub-tree is created based on the secret and public keys. Once the tree layer is set, the hash function is initialized based on the public seed and the secret seed. Then the root node of the sub-tree is computed. [20]

Algorithm 5: Sphincs+: Key Generation

Data: No Input Required
Result: SK = (SK_SEED, SK_PRF, PUB_SEED, root) and PK = (PUB_SEED, root)
1 **procedure** KEY GENERATION
2 Generation of SK_SEED, SK_PRF, and PUB_SEED using AES
3 Set layer initialize
4 Hash function initialization
5 Compute the root node of the top layer of tree.
6 **return**(SK, PK)

Signature Generation
The Signature Generation Algorithm generates the digital signature using the secret key. Different hash functions such as SHA, HARAKA, and SHAKE can be implemented for SPHINCS+ variants, which are initialized. Then a random number is generated, which generates random hash values for the message along with the message and secret key pseudo-random function. The message digest, leaf index, and tree values are generated based on this random hash value. The FORS+ public key and signature are generated and stored in the SPHINCS+ signature. The WOTS tree parameter is set up for each layer, and we generate the WOTS signature. Using the WOTS signature, the algorithm computes the root and the authentication paths for the WOTS leaf. The index leaf and tree are updated for the next iterations. The signature is completely generated after the authentication path for all sub-trees is calculated. [20]

Signature Verification

Signature verification algorithm verifies the signature using the public key. The hash function is initialized after taking the public seed from public key. The signature is invalid and rejected if the signature length is not size of message hash + signature size + all WOTS sub-tree signature size + FORS. If the signature is valid the WOTS tree is created along with the message digest, tree, and leaf index, which are generated using the message from the signature, signature random number, and public seed. Then the FORS public key is calculated using this message digest, public seed, and signature. For every WOTS sub-tree, a root value is computed. Then the WOTS public key is generated. The hash value of the leaf node is calculated using the public key. The leaf node and the authentication path from the key are used to compute the root node of the sub-tree. The value of the next sub-tree is then updated. After all the sub-trees are calculated, the public key root value should match this root node of the final sub-tree for successful signature verification. In case the signatures do not match, the verification fails to verify. [20]

There have been many efforts recently to compress the size of SPHINCS+ with minimum cost to the execution time for signature. SPHINCS+ remains a viable alternative to lattice based digital signatures. [23]

Algorithm 6: Sphincs+: Signature Generation

Data: Message m, Message Length mlen, and Secret Key sk = (SK_SSED, SK_PRF,PUB_SEED, root)
Result: Signature sm = (sign, m), Signature Length smle

procedure SIGNATURE GENERATION
Initialize the hash function with PUB_SEED and SK_SEED
Generate a random seed number
Compute the digest from random number, SK$_P RF, and message.$
Compute message hash, tree and leaf index
Setup for FORS+ and WOTS
Sign the message hash using FORS+
for *(i=0; i < SPX_D; i++)* **do**
    Setup the WOTS layer
    Compute WOTS signature
    Compute hash root node tree and authentication path (part of signature)
end
Concatenate the message with sign
return Signature

Algorithm 7: Sphincs+: Signature verification

Data: Message m, Message Length mlen, and Secret Key sk = (SK_SSED, SK_PRF,PUB_SEED, root)
Result: Signature sm = (sign, m), Signature Length smle

procedure SIGNATURE GENERATION
Initialize the hash function with PUB_SEED and SK_SEED
Generate a random seed number
Compute the digest from random number, SK_PRF, and message.
Compute message hash, tree and leaf index
Setup for FORS+ and WOTS
Sign the message hash using FORS+
for *(i=0; i < SPX_D; i++)* **do**
    Setup the WOTS layer
    Compute WOTS signature
    Compute hash root node tree and authentication path (part of signature)
end
Concatenate the message with sign
return Signature

4 Proposed Smart Card operation environment design

A smart card is basically a plastic/metallic card which comprises of a integrated circuit chips which may or may not have any computational ability such as the functions of a microprocessor, memory etc. Credit cards which have magnetic stripes and optical laser storage can also be termed as smart cards. Through the course of time, smart cards have evolved and have been used for authentication for a long time. Smart Cards are widely used as a secondary form of authentication in addition to a primary mode, basically a form of multi-factor authentication [24].

Smart Cards are usually used in high security perimeters where entry to unauthorised personnel is prohibited. Few of the industries which widely use smart cards are enterprises, financial institutions, government Institutions, healthcare Organisations or Hospitals, telecommunications Industry and transportation Industry. Smart Cards are basically allotted to authorised personnel to have access to places of high security which are otherwise off bounds to the general public or employees of an organisation.

Most of the smart cards today still use magnetic strips to authenticate users and industries that have adopted authentication technologies which employ the use of smart cards have cryptography algorithms which aren't quantum-safe and have major disadvantages of getting forged and used in a malicious way.

With the advent of research in applying quantum-secure cryptography algorithms in the smart card technology industry these disadvantages can be curbed. We intend to study both of the Quantum-Safe Algorithms and check their advantages and disadvantages when implemented on a Smart Card for authentication.

Post-quantum cryptography algorithms cannot function on the current memory and processing power standards for Smart Cards. High end smart cards that are available today provide 48kB of RAM and CPU processing power of 32-bit, 1 core @100 MHz which is not sufficient to execute PQC Algorithms in acceptable time [25]. Therefore to make the smart cards work, we need to shift most of the processing power to the card reader or the central system within which the card will operate. Therefore we propose an environment in which we can implement these algorithms.

In a scenario where a user wants to access a secure premise (ideally an employee of an organization), let us call the user Alice. Alice is a part of an organization. The organization responsible for issuing keys for Smart Cards issues a Smart Card to Alice. The Smart Card only stores a secret key that is encrypted with the master key of the Key Issuing Authority.

Alice scans her smart card along with her bio-metric information. The bio-metric information is used as the message which can be used to generate the signature. The bio-metric information also serves as a way of checking whether the user is actually the user and not someone who is misusing another employee's card or an adversary. The secret key is extracted from the smart card and a signature is generated using the signature generation algorithm on the smart card reader. The signature is then sent to the signature verification server where the signature verification takes place and determined whether Alice gets access or not. Figure 6 depicts the scenario we discussed pictorially

5 Space and Execution time comparison of Dilithium Vs SPHINCS+

Our study was based on the comparison of two algorithms which are Dilithium (Lattice-based) and SPHINCS+ (Stateless-hash based). We fixed a device for testing these algorithms whose specifications were RAM: 16GB CPU: AMD Ryzen 3 5300U @ 2.6 GHz. We didn't have access to real smart cards and therefore had to constrict our scope to computers and because majority of the smart cards are Java-based we managed to get hands on the Java source codes of these algorithms.

These are the results after we executed the code to generate signatures for similar messages for both the PQC Algorithms. Figure 7 represents the Security parameters, signature size,

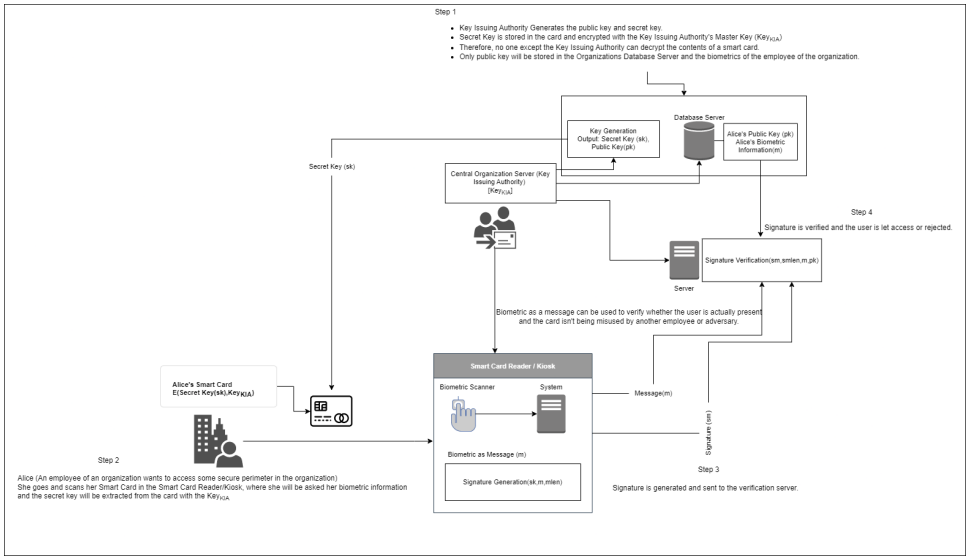

Figure 6: This is the proposed design of an environment which doesn't require any computational resources on a smart card.

public key size, and secret key size of SPHINCS+ NIST security variants for all levels[19].

Parameters	LEVEL 1 SPHINCS+-128f	LEVEL 1 SPHINCS+-128s	LEVEL 3 SPHINCS+-192f	LEVEL 3 SPHINCS+-192s	LEVEL 5 SPHINCS+-256f	LEVEL 5 SPHINCS+-256s
NIST LEVEL	1	1	3	3	5	5
n	16	16	24	24	32	32
h	60	64	66	64	68	64
d	20	8	22	8	17	8
Log t	9	15	8	16	10	14
k	30	10	33	14	20	22
W	16	16	16	16	16	16
Signature size (bytes)	16976	8080	35664	17064	49216	29792
Public Key size (bytes)	64	64	96	96	128	128
Private Key Size (bytes)	32	32	48	48	64	64

Figure 7: Security parameters, signature size, public key size, and secret key size of SPHINCS+ variants

Figure 8 shows the sizes of different security parameters of CRYSTALS - Dilithium algorithm [19].

In Figure 9 we present a graphical comparison of signature sizes between present day cryptosystems such as RSA and ECC with all variants of SPHINCS+ and Dilithium. It is clearly visible that signature sizes of SPHINCS+ is way greater than any other algorithm's signature size which makes it the least suitable for use in real time authentication, on the other hand, Dilithium gives us promising results which can be used in real time authentication with lower resources as compared to SPHINCS+.

When comparing execution times of Key Generation, Signature Generation and Signature Verification of both the algorithms we get the following results Dilithium 10 Vs SPHINCS+

	Weak	Medium	Recommended	Very high
NIST security level	–	1	2	3
Parameter q	8380417	8380417	8380417	8380417
Parameter d	14	14	14	14
Parameter weight of c	60	60	60	60
Parameter $\gamma_1 = (q-1)/16$	523776	523776	523776	523776
Parameter $\gamma_2 = \gamma_1/2$	261888	261888	261888	261888
Parameter (k, l)	(3,2)	(4,3)	(5,4)	(6,5)
Parameter η	7	6	5	3
Parameter β	375	325	275	175
Parameter ω	64	80	96	120
Signature size (bytes)	1387	2044	2701	3366
Public key size (bytes)	896	1184	1472	1760
Secret key size (bytes)	2081	2733	3348	3916

Figure 8: Security parameters, signature size, public key size, and secret key size of Dilithium variants

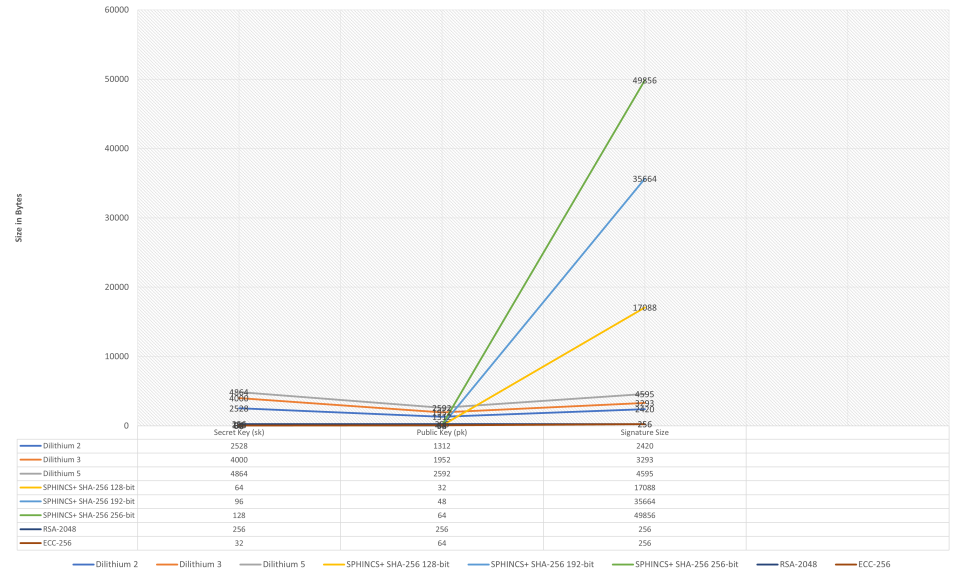

Figure 9: Comparison between signature sizes for all security levels of Dilithium, SPHINCS+, RSA and ECC

11. Clearly SPHINCS+ takes way more time than Dilithium for signature generation (Sign) which makes SPHINCS+ not a suitable choice when looking forward to efficiency.

As we used computational resources of a computer instead of smart cards, we obtained results from several sources which implemented these algorithms on different hardware architectures and processors. We found similar trends in results on most platforms and concluded that SPHINCS+ is definitely not a choice to consider when looking for efficiency.

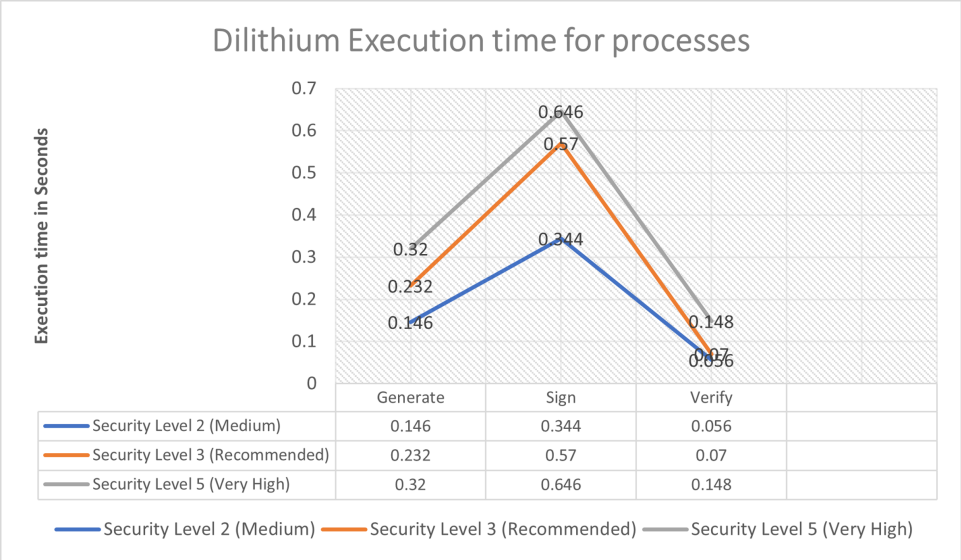

Figure 10: Comparison between signature sizes for all security levels of Dilithium, SPHINCS+, RSA and ECC

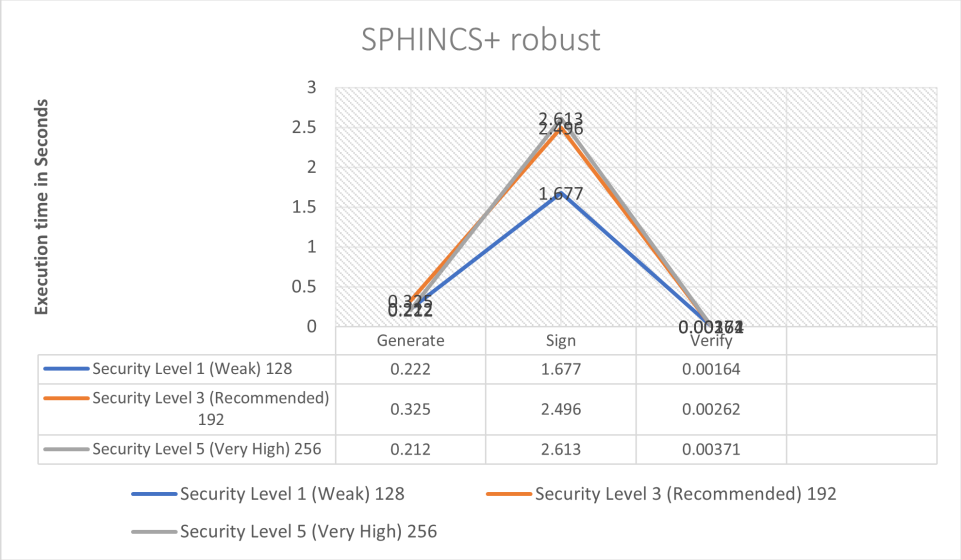

Figure 11: Comparison between signature sizes for all security levels of Dilithium, SPHINCS+, RSA and ECC

6 Results and Conclusion

Currently, off-card computation seems the most feasible as the current Smart cards need more processing power to generate, store and process digital signatures for time-sensitive applications such as authentication and the payment industry. CRYSTALS - Dilithium is the perfect candidate for post-quantum lattice-based digital signature implementation on smart cards. It is significantly faster than the other candidates that qualified for NIST round 3. The symmetric cryptography-based signature schemes that use a hash-based signature framework like SPHINCS+ have significant advantages like speed, signature size, and overall security. However, stateless signature schemes based on their symmetric

primitive do not perform as well as lattice-based signature schemes like CRYSTALS - Dilithium. It is recommended to use such hash-based digital signatures in applications that involve the transmission of the signature or in applications that do not require the rapid generation of digital signatures. Considering the small private and public key sizes, SPHINCS+ can be used in applications such as code or certificate signing or for TLS protocol. Research on increasing the computation power of traditional smart cards for implementing post-quantum digital signatures and comparing the feasibility of the application of smart cards can bring a breakthrough in the relatively new domain of post-quantum secure authentication.

References

[1] S. Kemp, "Digital around the world," 2022.

[2] Wikipedia contributors, "Pkcs 1 — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=PKCS_1&oldid=1120337930, 2022, [Online; accessed 20-November-2022].

[3] ——, "Rsa (cryptosystem) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&oldid=1121810982, 2022, [Online; accessed 20-November-2022].

[4] ——, "Rsa numbers — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=RSA_numbers&oldid=1122295571, 2022, [Online; accessed 20-November-2022].

[5] L. Malina, S. Ricci, P. Dzurenda, D. Smekal, J. Hajny, and T. Gerlich, "Towards practical deployment of post-quantum cryptography on constrained platforms and hardware-accelerated platforms," in *International Conference on Information Technology and Communications Security*. Springer, 2019, pp. 109–124.

[6] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 212â219. [Online]. Available: https://doi.org/10.1145/237814.237866

[7] V. Mavroeidis, K. Vishi, M. D. Zych, and A. Jøsang, "The impact of quantum computing on present cryptography," *arXiv preprint arXiv:1804.00200*, 2018.

[8] T. Moses, "The quantum computer and its implications for public-key crypto systems," https://www.entrust.com/-/media/documentation/whitepapers/sl20-1026-001-ssl_quantumcomputers-wp.pdf, 2019.

[9] I. Inc, "The quantum computer and its implications for public-key crypto systems," https://ionq.com/news/february-23-2022-ionq-aria-furthers-lead, 2022.

[10] E. Martín-López, A. Laing, T. Lawson, R. Alvarez, X.-Q. Zhou, and J. L. O'Brien, "Experimental realization of shor's quantum factoring algorithm using qubit recycling," *Nature Photonics*, vol. 6, no. 11, pp. 773–776, oct 2012. [Online]. Available: https://doi.org/10.1038%2Fnphoton.2012.259

[11] D. Ott, C. Peikert *et al.*, "Identifying research challenges in post quantum cryptography migration and cryptographic agility," *arXiv preprint arXiv:1909.07353*, 2019.

[12] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, C. Miller, D. Moody, R. Peralta *et al.*, "Status report on the third round of the nist post-quantum cryptography standardization process," *NIST, Tech. Rep. NISTIR*, vol. 8413, 2022.

[13] L. Malina, S. Ricci, P. Dzurenda, D. Smekal, J. Hajny, and T. Gerlich, "Towards practical deployment of post-quantum cryptography on constrained platforms and hardware-accelerated platforms," in *Innovative Security Solutions for Information Technology and Communications*, E. Simion and R. Géraud-Stewart, Eds. Cham: Springer International Publishing, 2020, pp. 109–124.

[14] V. Lyubashevsky, "Fiat-shamir with aborts: Applications to lattice and factoring-based signatures," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009, pp. 598–616.

[15] P. Schwabe, "Dilithium resources," https://pq-crystals.org/dilithium/index.shtml, 2021.

[16] E. Kiltz, V. Lyubashevsky, and C. Schaffner, "A concrete treatment of fiat-shamir signatures in the quantum random-oracle model," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 552–586.

[17] Q. Liu and M. Zhandry, "Revisiting post-quantum fiat-shamir," in *Annual International Cryptology Conference*. Springer, 2019, pp. 326–355.

[18] J. Don, S. Fehr, and C. Majenz, "The measure-and-reprogram technique 2.0: multi-round fiat-shamir and more," in *Annual International Cryptology Conference*. Springer, 2020, pp. 602–631.

[19] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, and R. Karri, *CRYSTALS-Dilithium*. Cham: Springer International Publishing, 2021, pp. 13–30. [Online]. Available: https://doi.org/10.1007/978-3-030-57682-0_2

[20] ——, "Sphincs+," in *Hardware Architectures for Post-Quantum Digital Signature Schemes*. Springer, 2021, pp. 141–162.

[21] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn, "Sphincs: Practical stateless hash-based signatures," in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 368–397.

[22] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The sphincs+ signature framework," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2129â2146. [Online]. Available: https://doi.org/10.1145/3319535.3363229

[23] M. Kudinov, A. Hülsing, E. Ronen, and E. Yogev, "Sphincs+ c: Compressing sphincs+ with (almost) no cost," *Cryptology ePrint Archive*, 2022.

[24] M. Haykin and R. Warnar, "Smart card technology: New methods for computer access control," 1988-09-01 1988.

[25] I. C. . S. Labs, "Smartcard and post-quantum cryptography," https:

//csrc.nist.gov/CSRC/media/Presentations/smartcard-and-post-quantum-crypto/images-media/session-5-greuet-smartcard-pqc.pdf, 2021.