applied sciences

Article

Post Quantum Cryptographic Keys Generated with Physical Unclonable Functions

Bertrand Cambou *, Michael Gowanlock, Bahattin Yildiz, Dina Ghanaimiandoab, Kaitlyn Lee, Stefan Nelson, Christopher Philabaum, Alyssa Stenberg and Jordan Wright

College of Engineering Informatics and Applied Sciences (CEIAS), Northern Arizona University (NAU), Flagstaff, AZ 86011, USA; michael.gowanlock@nau.edu (M.G.); bahattin.yildiz@nau.edu (B.Y.); dg856@nau.edu (D.G.); kdl222@nau.edu (K.L.); swn34@nau.edu (S.N.); cp723@nau.edu (C.P.); ajs937@nau.edu (A.S.); jaw566@nau.edu (J.W.)

* Correspondence: bertrand.cambou@nau.edu; Tel.: +1-928-523-7824

Featured Application: Using physical unclonable functions (PUFs), in support of networks secured with a public key infrastructure, to generate, on demand, the key pairs needed for lattice and code PQC algorithms.

Abstract: Lattice and code cryptography can replace existing schemes such as elliptic curve cryptography because of their resistance to quantum computers. In support of public key infrastructures, the distribution, validation and storage of the cryptographic keys is then more complex for handling longer keys. This paper describes practical ways to generate keys from physical unclonable functions, for both lattice and code-based cryptography. Handshakes between client devices containing the physical unclonable functions (PUFs) and a server are used to select sets of addressable positions in the PUFs, from which streams of bits called seeds are generated on demand. The public and private cryptographic key pairs are computed from these seeds together with additional streams of random numbers. The method allows the server to independently validate the public key generated by the PUF, and act as a certificate authority in the network. Technologies such as high performance computing, and graphic processing units can further enhance security by preventing attackers from making this independent validation when only equipped with less powerful computers.

Keywords: Lattice cryptography; code cryptography; post quantum cryptography; physical unclonable function; public key infrastructure; high performance computing

Citation: Cambou, B.; Gowanlock, M.; Yildiz, B.; Ghanaimiandoab, D.; Lee, K.; Nelson, S.; Philabaum, C.; Stenberg, A.; Wright, J. Post Quantum Cryptographic Keys Generated with Physical Unclonable Functions. *Appl. Sci.* **2021**, *11*, 2801. https://doi.org/10.3390/app11062801

Academic Editors: Leandros Maglaras, Ioanna Kantzavelou and Mohamed Amine Ferrag

Received: 13 February 2021
Accepted: 18 March 2021
Published: 21 March 2021

1. Introduction

In most public key infrastructure (PKI) schemes for applications such as cryptographic currencies, financial transactions, secure mail and wireless communications, the public keys are generated by private keys with Rivest–Shamir–Adleman (RSA) and elliptic curve cryptography (ECC). These private keys are natural numbers, typically 3000-bit long for RSA and 256-bits long for ECC. For example, in the case of ECC, the primitive element of the elliptic curve cyclic group is multiplied by the private key to find the public key. It is now anticipated that quantum computers (QC) will be able to break both RSA and ECC when the technology to manufacture enough quantum nodes becomes available. The paper entitled "A Riddle Wrapped in an Enigma" by N. Koblitz and A. J. Menezes suggested that the ban of RSA and ECC by the National Security Agency is unavoidable, and that the risk of QC is only one element of the problem [1]. Plans to develop post quantum cryptographic (PQC) schemes have been proposed to secure blockchains by Kiktenko et al. [2], and for cryptocurrency security by Semmouni et al. [3], even if the timeline for the availability of powerful QC is highly speculative. Recently, Campbell et al. [4], and Kampanakisy et al. [5], proposed distributed ledger cryptography and digital signatures with PQC. In 2015, the National Institute of Standards and Technology (NIST) initiated a large-scale program to standardize PQC algorithms. One possible implementation of PQC algorithms for a PKI is the one in which each client device, or designate, generates the key pairs, and sends the public keys to a certificate authority (CA). This assumes a separate authentication process, and that each client device can securely store the key pairs.

The research question that is the subject of this paper is the feasibility of using physical unclonable functions (PUFs), together with a handshake process with the CA that generate new key pairs from the PUF at each transaction, thereby eliminating the need to store the key pair. Attempts to retrieve the secret keys are not relevant anymore as they are only used once. Such a configuration is raising several structural and technical questions. A secure enrollment process of each PUF needs to be established, and the CA has to store the challenges and reference values of each PUF. Such an infrastructure is already known when the security is based on secure hardware elements and tokens and requires special protections against opponents. From a technical standpoint, it is questionable that the long key pairs necessary for PQC algorithms can be generated from physical elements. While a single bit mismatch is not acceptable for PQC algorithms, the natural drifts of PUFs over environmental conditions and aging are real concerns that need to be addressed. This paper is structured in the following way:

[Section 2]: The lattice and code-based cryptographic algorithms under consideration for standardization by NIST are presented. These algorithms are well documented, and the software stack written in C can be downloaded for IoT implementation. The schemes are based on the generation of random numbers, and the computation of public–private key pairs. The digital signature algorithms (DSA), and key encapsulation mechanisms (KEM) are not more complex to implement with PQC than with the existing asymmetrical cryptographic schemes.
[Section 3]: We present some of the challenges associated with the use of PUF technology to secure PKI architectures. The proposed methods are based on existing cryptographic schemes, and commercially available PUFs. We present how the response based cryptographic (RBC) scheme can overcome the bit error rates (BER) that occur when keys are generated from physical elements. Finally, we present some hardware considerations in the implementation of PQC for PKI.
[Section 4]: In this section, we propose schemes that use PUFs to generate the public–private key pairs for lattice and code-based cryptography. We show how the combination of random number generators, combined with the streams generated by the PUF can generate key pairs with relatively low error rates. We show how the error in these streams can be corrected using a search engine.
[Section 5]: Finally, in the implementation and experimental section, we compare cryptographic schemes and algorithms. We analyze the experimental results comparing the efficiency of RBC operating with various PQC schemes, ECC, and advanced encryption standard (AES). As expected, asymmetrical schemes are slower than AES; however, the performance of the selected PQC algorithms is encouraging for the implementation of PUF-based architecture, using the RBC to handle the expected BER.

2. Lattice and Code-Based Post Quantum Cryptography

In 2019, the number of candidates of the NIST standardization effort was narrowed to 26, as part of phase two of the program [6]. In July 2020, NIST announced the selection of seven likely finalists for phase three of the program [7]: CRYSTALS-Kyber, CRYSTALS-Dilithium, SABER, NTRU, and FALCON with lattice cryptography [8–12]; RAINBOW with multivariate cryptography [13], and Classic McEliece with code-based cryptography [14,15]. The software developed is mainly targeting DSA applications, as well as KEM. Lattice cryptography is relatively mature, well documented, and is most likely to become mainstream for cybersecurity. Lattice-based algorithms exploit hardness to resolve problems such as the closest vector problem (CVP), learning with error (LWE), and learning with rounding (LWR) algorithms, and share similarities with the knapsack cryptographic problem.

2.1. Learning with Error Cryptography

The LWE of the CVP problem was first introduced by Regev [16]. The knowledge of integer-based vector $\boldsymbol{t}$, and matrix $\boldsymbol{A}$ with $\boldsymbol{t} = \boldsymbol{A}.s_1$ cannot hide the vector $\mathbf{s}_1$; however, the addition of a "small" vector of error $\mathbf{s}_2$ with $\boldsymbol{t} = \boldsymbol{A}.s_1+s_2$, makes it hard to distinguish the vectors s_1 and s_2 from $\boldsymbol{t}$. The vector s_2 needs to be small enough for the encryption/decryption cycles, but large enough to prevent a third party from uncovering the private key (s_1; s_2) from the public information ($\boldsymbol{t}$; $\boldsymbol{A}$). The public–private cryptographic key pair generation for client device i can be based on polynomial computations in a lattice ring, and is described in Figure 1:

1. The generation of a first data stream called seed $\boldsymbol{a}_{(i)}$ that is used for the key generation; in the case of LWE, the seed $\boldsymbol{a}_{(i)}$ is shared openly in the network.
2. The generation of a second data stream called seed $\boldsymbol{b}_{(i)}$ that is used to compute a second data stream for the private key $\boldsymbol{Sk}_{(i)}$; the seed $\boldsymbol{b}_{(i)}$ is kept secret.
3. The public key $\boldsymbol{Pk}_{(i)}$ is computed from both data streams and is openly shared.
4. The matrix $\boldsymbol{A}_{(i)}$ is generated from seed $\boldsymbol{a}_{(i)}$.
5. The two vectors $s_{1(i)}$ and $s_{2(i)}$ are generated from seed $\boldsymbol{b}_{(i)}$.
6. The vector $\boldsymbol{t}_{(i)}$ is computed: $\boldsymbol{t}_{(i)} \leftarrow \boldsymbol{A}_{(i)}\ s_{1(i)} + s_{2(i)}$.
7. Both seed $\boldsymbol{a}_{(i)}$ and $\boldsymbol{t}_{(i)}$ become the public key $\boldsymbol{Pk}_{(i)}$.
8. Both $s_{1(i)}$ and $s_{2(i)}$ become the private key $\boldsymbol{Sk}_{(i)}$.

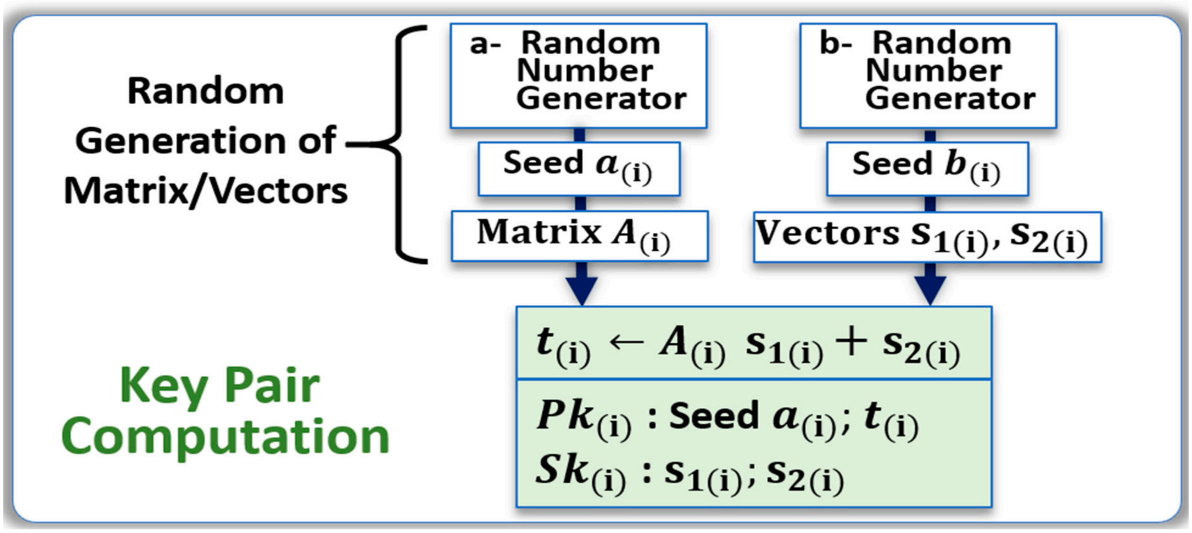
Random Generation of Matrix/Vectors
a- Random Number Generator
b- Random Number Generator
Seed $\boldsymbol{a}_{(i)}$
Seed $\boldsymbol{b}_{(i)}$
Matrix $\boldsymbol{A}_{(i)}$
Vectors $\mathbf{s}_{1(i)}, \mathbf{s}_{2(i)}$
Key Pair Computation
$\boldsymbol{t}_{(i)} \leftarrow \boldsymbol{A}_{(i)}\ \mathbf{s}_{1(i)} + \mathbf{s}_{2(i)}$
$\boldsymbol{Pk}_{(i)}$: Seed $\boldsymbol{a}_{(i)}$; $\boldsymbol{t}_{(i)}$
$\boldsymbol{Sk}_{(i)}$: $\mathbf{s}_{1(i)}$; $\mathbf{s}_{2(i)}$

Figure 1. Example of public–private key generation for LWE based cryptography. The matrix $A_{(i)}$ and vectors $S_{1(i)}$ and $S_{2(i)}$ are generated from random number generators. The vector $t_{(i)}$ is computed from these elements for the generation of the key pair.

A digital signature algorithm (DSA) can be realized from the LWE instance by first generating a public–private key pair as in Figure 1. The secret key is then used to sign a message, and the public key is used to verify this signed message. In CRYSTALS-Dilithium [8], the authors use a Fiat-Shamir with Aborts approach [17] for their signing and verification procedure. The outline of the signing procedure is as follows:

1. Generate a masking vector of polynomials $\boldsymbol{y}$.
2. Compute vector $\boldsymbol{A}.\boldsymbol{y}$ and set w_1 to be the high-order bits of the coefficients in this vector.
3. Create the challenge $\boldsymbol{c}$, as the hash of the message and w_1.
4. Compute intermediate signature $\boldsymbol{z} = \boldsymbol{y} + \boldsymbol{c}.s_1$.
5. Set parameter $\boldsymbol{\beta}$ to be the maximum coefficient of $\boldsymbol{c}.s_1$.

6. If any coefficient of z is larger than $\gamma_1 - \beta$, then reject and restart at step 1.
7. If any coefficient of the low-order bits of $A.z - c.t$ is greater than $\gamma_2 - \beta$, then reject and restart at step 1.

Note: γ_1, γ_2, and β are set such that the expected number of repetitions is between 4 and 7.

The general outline of the verification procedure is given by the following:

Compute w_1' to be the high-order bits of $A.z–c.t$ and accept if all coefficients of z are less than $\gamma_1–\beta$ and if c is the hash of the message and w_1'.

Encapsulation allows for two parties to securely share a symmetric key by encapsulating the key in ciphertext. When both parties have the symmetric key, they are then able to use a symmetric-key encryption algorithm to communicate (e.g., AES). These algorithms are known as key encapsulation mechanisms (KEM) and a few examples from NIST are SABER [18], Classic McEliece [14,15], CRYSTALS-Kyber [19], and NTRU [20]. The process of using encapsulation with LWE/LWR is described below:

- The public and private keys of both parties are constructed as described in Figure 1.
- Person A sends Person B their public key.
- Person B randomly generates a symmetric key and encapsulates it in a ciphertext with the public key of person A.
- Person B sends the ciphertext to person A.
- Person A decapsulates the ciphertext with their private key.
- Both parties now have the symmetric key in their possession.

In summary, LWE schemes are now relatively mature, and very well documented. The methods selected by the NIST standardization program, presented here, are straightforward to use. The codes are widely available online for download, and we successfully deployed them in our research environment to study the use of PUFs for key generation.

2.2. Learning with Rounding Cryptography

The learning with rounding problem was first introduced by Banerjee [21]. It is the derandomized version of learning with error, which deterministically generates the noise in the LWE by rounding coefficients. This will eliminate the noise sampling, and significantly reduce the bandwidth [22]. The LWR is proved to be as hard as LWE to solve; hence, it remains secure to be used in cryptographic applications. In schemes such as "Saber", a constant h is added as a constant vector to simulate the rounding operation by bit shifting, therefore playing a similar protecting role than the error vectors of LWE [18]. Saber, which is one of the NIST's finalists in the key encapsulation category, uses LWR for key generation in public key encryption and key encapsulation. Below all three steps of PKE and KEM are described:

Saber PKE Key Generation

1. Similar to LWE, seed $a_{(i)}$ is used to generate matrix $A_{(i)}$.
2. Seed $b_{(i)}$ is used to generate vector $s_{(i)}$.
3. The vector $t_{(i)}$ is computed: $t_{(i)} \leftarrow A_{(i)} . s_{(i)} + h_{(i)}$.
4. Both seed $a_{(i)}$ and $t_{(i)}$ become the public key $Pk_{(i)}$.
5. $s_{(i)}$ becomes the private key $Sk_{(i)}$.

Saber PKE Encryption

1. The seed $a_{(i)}$ and $t_{(i)}$ is extracted from public key to encrypt the message m.
2. Matrix $A_{(i)}$ and vector $s'_{(i)}$ are generated.
3. The vector $t'_{(i)}$ is computed by rounding the product of $A_{(i)} . s'_{(i)}$: $t'_{(i)} \leftarrow A'_{(i)} . s'_{(i)} + h_{(i)}$.
4. Polynomial $v'_{(i)}$ is calculated as: $v'_{(i)} = t_{(i)} . s'_{(i)}$.
5. $v'_{(i)}$ is used to encrypt the message m which denoted as c_m.
6. Ciphertext consists of c_m and $t'_{(i)}$.

Saber PKE Decryption

1. $v_{(i)}$ is calculated as: $v_{(i)} = t'_{(i)} \cdot s_{(i)}$.
2. The message m' is decrypted by reversing computations with $v_{(i)}$ and c_m.

The Saber key encapsulation mechanism has three steps: Saber KEM Key Generation, Saber KEM Encapsulation, and Saber KEM Decapsulation:

Saber KEM Key Generation

1. Saber PKE key generation is used to return seed $\boldsymbol{a}_{(i)}$, $\boldsymbol{t}_{(i)}$ and $\boldsymbol{s}_{(i)}$.
2. Both seed $\boldsymbol{a}_{(i)}$ and $\boldsymbol{t}_{(i)}$ become the Saber KEM public key $\boldsymbol{Pk}_{(i)}$.
3. The hashed public key $\boldsymbol{Pkh}_{(i)}$ is generated using SHA3-256.
4. Parameter $\boldsymbol{z}$ is randomly sampled.
5. $\boldsymbol{z}, \boldsymbol{Pkh}_{(i)}$ and $\boldsymbol{s}_{(i)}$ become the Saber KEM secret key.

Saber KEM Encapsulation

1. Message $\boldsymbol{m}$ and public key $\boldsymbol{Pk}_{(i)}$ are hashed using SHA3-256.
2. Saber PKE encryption is used to generate ciphertext.
3. Hash of the $\boldsymbol{Pk}_{(i)}$ and ciphertext are concatenated, then hashed to encapsulate the key.

Saber KEM Decapsulation

1. Message $\boldsymbol{m'}$ is decrypted by using Saber PKE Decryption.
2. The decrypted message $\boldsymbol{m'}$ and hashed public key $\boldsymbol{Pkh}_{(i)}$ are hashed to generate K′.
3. Ciphertext $\boldsymbol{c'_m}$ is generated from saber PKE Encryption for message $\boldsymbol{m'}$.
4. If $\boldsymbol{c_m} = \boldsymbol{c'_m}$ then the $\boldsymbol{K = Hash(K',c)}$, if not, $\boldsymbol{K = Hash(z,c)}$.

The level of documentation available on LWR is not quite as complete as what is available for LWE. However, the proposed implementation of LWR for NIST’s PQC program is solid. The use of PUFs to secure PKIs based on LWR is not more challenging than the one based on LWE.

2.3. NTRU Cryptography

Cryptographic algorithms such as FALCON, which uses NTRU (***Nth*** degree of TRUncated polynomial ring) arithmetic, are also based on lattice cryptography. The parameters of the scheme include a large prime number N, a large number **q** and a small number **p** that are both used for modulo arithmetic. Two numbers **df** and **dg** are used to truncate the polynomials $\boldsymbol{f}_{(i)}$ and $\boldsymbol{g}_{(i)}$. The key generation cycle for client device (i), as shown in Figure 2, is the following:

1. Generation of the two truncated polynomials $\boldsymbol{f}_{(i)}$ and $\boldsymbol{g}_{(i)}$ from seed $\boldsymbol{a}_{(i)}$ and seed $\boldsymbol{b}_{(i)}$.
2. Computation of $\boldsymbol{Fq}_{(i)}$, which is the inverse of polynomial $\boldsymbol{f}_{(i)}$ modulo **q**.
3. Computation of $\boldsymbol{Fp}_{(i)}$, which is the inverse of polynomial $\boldsymbol{f}_{(i)}$ modulo **p**.
4. Computation of polynomial $\boldsymbol{h}_{(i)}$: $\boldsymbol{h}_{(i)} \leftarrow p \cdot \boldsymbol{Fq}_{(i)} \cdot \boldsymbol{g}_{(i)}$.
5. The private key $\boldsymbol{Sk}_{(i)}$ is $\{\boldsymbol{f}_{(i)}; \boldsymbol{Fp}_{(i)}\}$.
6. The public key $\boldsymbol{Pk}_{(i)}$ is $\boldsymbol{h}_{(i)}$.

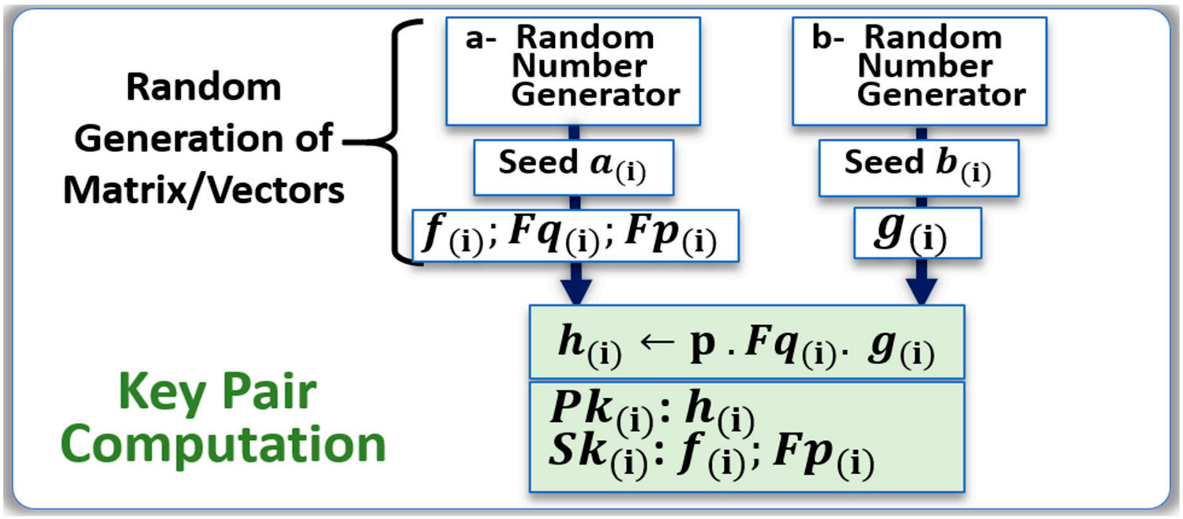
Random Generation of Matrix/Vectors
a- Random Number Generator
b- Random Number Generator
Seed $a_{(i)}$
Seed $b_{(i)}$
$f_{(i)}; Fq_{(i)}; Fp_{(i)}$
$g_{(i)}$
$h_{(i)} \leftarrow p \,.\, Fq_{(i)} .\, g_{(i)}$
Key Pair Computation
$Pk_{(i)}: h_{(i)}$
$Sk_{(i)}: f_{(i)}; Fp_{(i)}$

Figure 2. Example of key generation for NTRU cryptography. The polynomials *f*(i) and *g*(i) and are generated from random number generators, from which the inverses *Fq*(i) and *Fp*(i) are computed. The public key *h*(i) is also computed from these polynomials.

As the polynomials $f_{(i)}$ and $g_{(i)}$ are not always usable, they are subject to some preconditions such as invertible modulo p and q. The client device needs to try several possible random numbers, and select the ones giving acceptable private keys. Once sufficient public and private keys are available, the encryption of the plaintext message m, $m \in \{-1, 0, 1\}^N$ is done by finding the random polynomial r, $r \in \{-1, 0, 1\}^N$, which uses a corresponding parameter d_r, and calculating the ciphertext with the equation $e \equiv r.h + m$ (mod q). To retrieve m from e, we first calculate $a \equiv f.e$ (mod q) and lift the coefficients of a to be between $\pm\, q/2$. Then, a (mod p) is equal to m. [23].

NTRU lattices can also be applied to DSA. This was originally introduced in NTRU-Sign, but NIST submissions such as Falcon expand on these algorithms [9]. Falcon utilizes the GPV framework applied to NTRU lattices; that is, the public key is a long basis for an NTRU lattice while the private key is a short basis. From here, the message m is sent a non-lattice point C, utilizing a random value salt and hash function H. Using the short basis, a user signs by finding the closest vector v to c. The signature is (salt, $s = c - v$), verified by checking if s is short and H (msg ‖ salt) − s is a point on the lattice (verified using the long basis [24]).

The cryptography based on NTRU is well known, and extremely well documented. The polynomial arithmetic truncating the N-th element is elegant and effective. Like other lattice algorithms under consideration by NIST, we are considering the NTRU as a strong candidate, both for DSA, and KEM.

2.4. Code-Based Cryptography

Code-based algorithms such as Classic McEliece are implemented with binary Goppa codes, that is, Goppa codes with underlying computations in finite Galois fields $\mathbf{GF(2^m)}$. The parameters are an irreducible polynomial of degree **t**, field exponent **m**, and code length **n**. The resulting code has error-correction capability of **t** errors, the information-containing part of the code word has a size of $\mathbf{k} = \mathbf{n} - \mathbf{m} \times \mathbf{t}$ and has generator matrix **G** with a size of $\mathbf{k} \times \mathbf{n}$ [14,15].

The block diagram of Figure 3 is showing an example of public–private key generation for code-based cryptography, and client device i.

1. Seed $a_{(i)}$ is used to create a random invertible binary $\mathbf{k} \times \mathbf{k}$ scrambling matrix $S_{(i)}$.
2. Seed $b_{(i)}$ is used to create a random $\mathbf{n} \times \mathbf{n}$ permutation matrix $P_{(i)}$.
3. The public key $Pk_{(i)} = \hat{G}_{(i)}$ is computed with the generator matrix G: $\hat{G}_{(i)} \leftarrow S_{(i)} \,.\, G \,.\, P_{(i)}$
4. The private key $Sk_{(i)}$ is $\{G; S_{(i)}^{-1}, P_{(i)}^{-1}\}$.

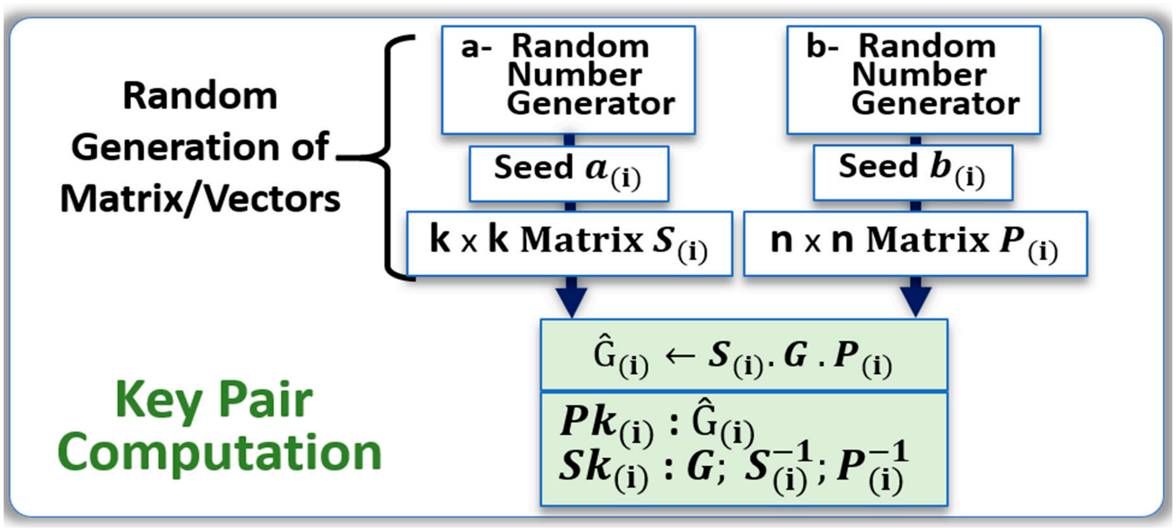
Random Generation of Matrix/Vectors
a- Random Number Generator
Seed $a_{(i)}$
k x k Matrix $S_{(i)}$
b- Random Number Generator
Seed $b_{(i)}$
n x n Matrix $P_{(i)}$
Key Pair Computation
$\hat{G}_{(i)} \leftarrow S_{(i)} . G . P_{(i)}$
$Pk_{(i)} : \hat{G}_{(i)}$
$Sk_{(i)} : G; S^{-1}_{(i)}; P^{-1}_{(i)}$

Figure 3. Example of key generation for code-based cryptography. The two matrixes $S_{(i)}$ and $P_{(i)}$ are generated from random number generators. The key pairs are computed from the matrixes, and the generator matrix G.

Given a generator matrix of a binary Goppa code G, an irreducible polynomial of degree t, the field exponent m, and the code length n, the encryption process involves the following steps:

1. Create the public key, $\hat{G}_{(i)}$ as described above.
2. Multiply the message m by $\hat{G}_{(i)}$, creating the ciphertext message $\hat{m}$.
3. Add a random error vector e of Hamming weight t to $\hat{m}$ to obtain the ciphertext **c.**

Given a ciphertext c, a decoding algorithm, and the private key $\{G; S_{(i)}^{-1}, P_{(i)}^{-1}\}$, decryption involves the following steps:

1. Compute $\hat{c} = c\, P_{(i)}^{-1}$.
2. Use the decoding algorithm to correct the errors to obtain $\hat{m}$.
3. Obtain the original message by computing $m = \hat{m}.S_{(i)}^{-1}$.

One example of a decoding algorithm is Patterson's algorithm. This algorithm calculates the error-locator polynomial which has roots corresponding with the locations of the error bits added to the encrypted message. This algorithm can be implemented as follows [25]:

Input: Syndrome polynomial s, Goppa polynomial g of degree t
Patterson **(s, g):**

```
1. $t = s^{-1} \bmod g$.
2. $t = \sqrt{t + x}$.
3. Find polynomials $a$, $b$ such that $b.t \equiv \bmod g$ with deg(a) ≤ ∣$t$/2∣ and deg(b) ≤ ∣($t$-1)/2∣
   using the extended Euclidean algorithm.
4. Calculate and return the error locator polynomial, $e = a^2 + x.b^2$.
```

Once the error locator polynomial is found, the Berlekamp Trace Algorithm can be used to find the roots of the polynomial via factorization. These correspond to the locations of the error bits added to the message. The Berlekamp Trace Algorithm can be implemented as follows [26]:

Input: Polynomial to factor p, trace polynomial t, basis index i
Berlekamp Trace **(p, t, i)**:

```
1. if deg($p$) $\leq 1$
2. return the root of $p$.
3. $p_0 = gcd(p, t(B_i . x))$.
4. $p_1 = gcd(p, 1+t(B_i . x))$.
   return berlekampTrace($p_0$, i+1), berlekampTrace($p_1$, i+1).
```

Code-based is probably the most mature, and well documented PQC algorithm currently under consideration. The new implementations are highly effective for KEM; the extended output functions allow the quick generation of the two matrixes needed for key generation.

3. Public Key Infrastructure

3.1. Public-private key pairs

As part of a PKI, the public–private key pairs can be used to securely transmit shared secret keys though KEM and to digitally sign messages with DSA (see Figure 4). The public key ***Pk***$_{(2)}$ of Client 2 encapsulates the shared secret key of Client 1, that can only be viewed by the client (2), thanks to their private key ***Sk***$_{(2)}$ that reverses the encapsulation. Client 1 uses their private key ***Sk***$_{(1)}$ to digitally sign a message that is verified with the public key ***Pk***$_{(1)}$, providing non-alteration and non-repudiation in the transaction. The trust and integrity of such an architecture relies on the following:

i. The secure generation and distribution of the public–private key pairs to the client devices that are participating in the PKI.
ii. The identification of the client devices, and trust in their public keys.
iii. The sharing of the public keys among participants.

Most PKIs rely on certificate authorities (CA) and registration authorities (RA) to offer such an environment of trust and integrity. The architecture is vulnerable to several threats, including loss of identity, man-in-the-middle attacks, and side channel attacks in which the private keys are exposed during KEM, and DSA.

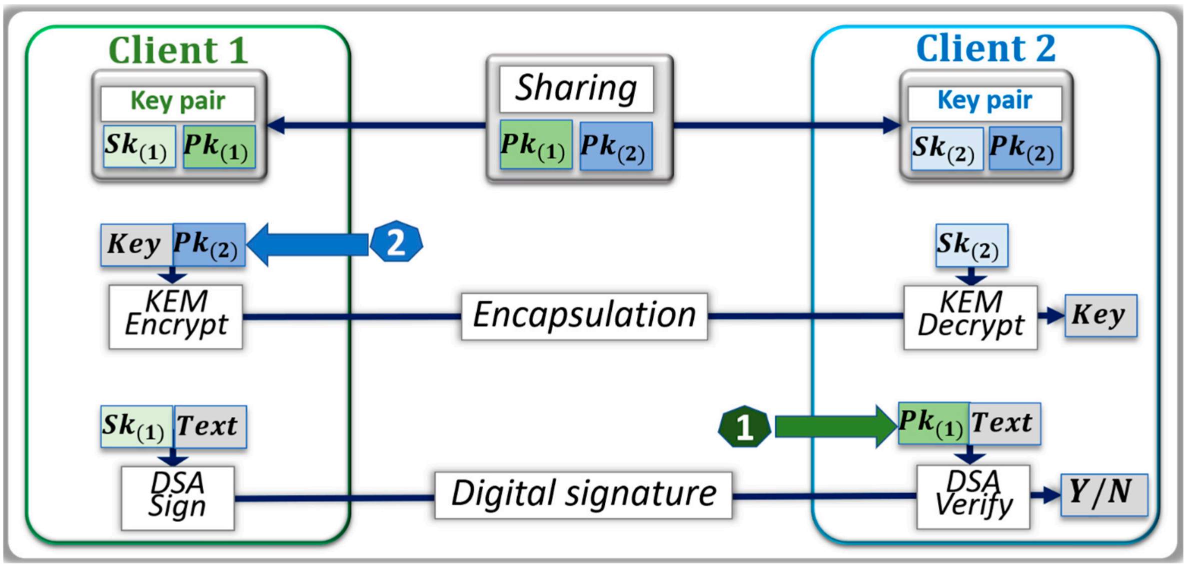

Figure 4. Communication protocol between two client devices with shared public keys. Each client device is set with its key pair. For KEM, Client 1 uses the public key of Client 2 to encrypt a shared key; Client 2 retrieves the key with their private key. For DSA, Client 1 signs a message with their private key, Client 2 verifies the message with the public key of Client 1.

3.2. PKI with Network of PUFs

The use of networks of PUFs can mitigate the vulnerabilities of PKIs. PUF technology exploits the variations created during fabrication to differentiate each device from all other devices, acting as a hardware "fingerprint" [27–29]. Solutions based on PUFs embedded in the hardware of each node can mitigate the risk of an opponent reading the keys stored in non-volatile memories. The keys for the PKI can be generated on demand with a one-time use; stealing a key becomes useless as new keys are needed at each transaction. During enrollment cycles, the images of the PUFs are stored in look-up tables in the CA (see Figure 5); enrollment has to be done only once in a secure environment. Handshake protocols [30] can select a portion of the PUFs—and their image is stored in the CA—to extract a data stream that generates the key pairs. The PUFs can be erratic, therefore the generation of cryptographic keys, the focus of this work, is challenging. A single-bit mismatch in a cryptographic key is not acceptable for most encryption protocols. Therefore, the use of error correcting code (also use the acronym ECC, not to be confused with "elliptic curve cryptography) methods, helper data, and fuzzy extractors can minimize the levels of errors [31–33]. The alternate method is one where the CA has search engines, such as response-based-cryptography (RBC), that can handle the validation of erratic keys [34–37].

The RBC engine that validates public keys, shown in Figure 5, generates public/private key pairs until the public key matches the client's provided key. The server searches over a seed (e.g., a 256 bit seed), and uses that seed for key generation. If the generated public key matches the client's public key, then the client is authenticated. If the public keys do not match, then the server flips one bit of the seed at a time (increasing the Hamming distance) until the public keys match. Thus, the search is carried out by generating the public/private key pairs by iterating over the seed and increasing the Hamming distance until the seed is found that matches the client's public key. The search space for a 256 bit key is 2^{256} and it would be nearly impossible to authenticate a user in a fixed time without the use of parallel computing. High-performance computing HPC and graphics processing unit (GPU) technologies are valuable to enhance the ability of the CA to validate the public key generated by the client devices. For instance, graphics processing units (GPUs) can be employed to parallelize and accelerate the authentication process. By using a GPU, the server can search over multiple keys in parallel.

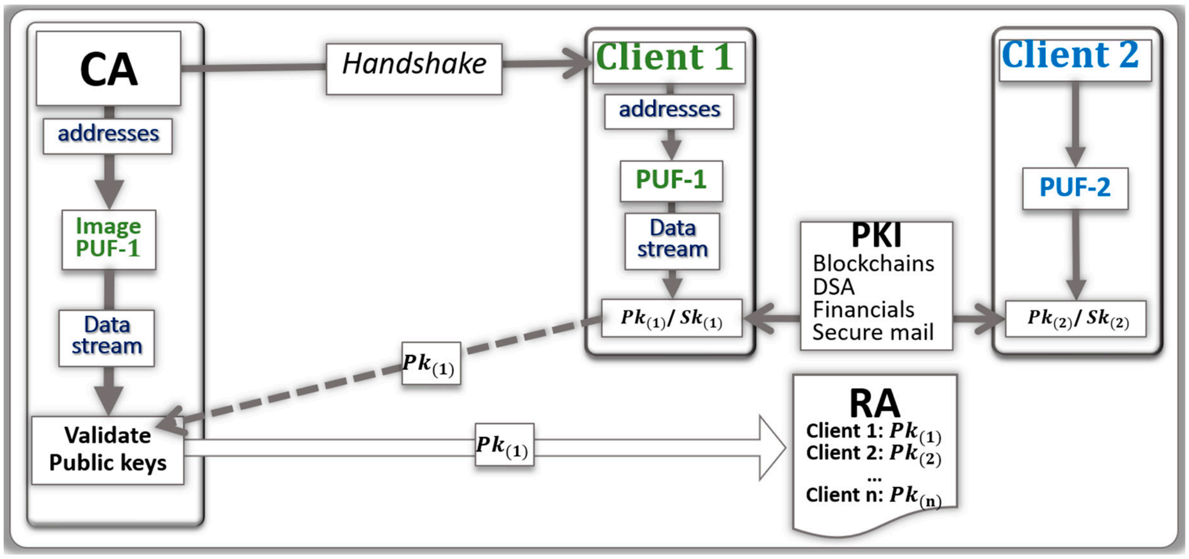

Figure 5. Physical unclonable function (PUF)-based public key infrastructure (PKI). The Certificate Authority (CA) verify the validity of the public keys and transfer them to the Register Authority (RA). The key pairs are generated from the PUFs embedded in each client device. The architecture enables cloud-based peer-to-peer secure transactions protected by asymmetrical cryptography.

3.3. Implementation of PQC Algorithms for PKI

The CRYSTALS-Dilithium digital signature algorithm consists of the following procedures: key generation, signing, and verification. These procedures are computationally bound by two operations: multiplication in the polynomial ring noted $\mathbb{Z}_q[X]/(X_n+1)$, and matrix/vector expansion via an extendable output function (XOF). Therefore, any attempt to optimize Dilithium should target these operations. We describe below the literature that focuses on such optimizations.

The operation of polynomial multiplication has a quasi-linear time complexity bound by the Number Theoretic Transform (NTT) implementation, and the operation of expansion via XOF is bound by the SHAKE-128 implementation. Using the AVX2 instruction set, the matrix and vector expansion is optimized by using a vectorized SHAKE-128 implementation that operates on four sponges that can absorb and squeeze blocks in parallel. Additionally, Ducas et al. [8] use the AVX2 instruction set to optimize the NTT thus speeding up the polynomial ring multiplication by about a factor of two. This optimization is achieved by interleaving the vector multiplications and Montgomery reductions so that parts of the multiplication latencies are hidden.

Nejatollahi et al. [38] outline two different works that optimize the NTT using an Nvidia GPU. The first reports higher throughput polynomial multiplication [39] and the second is a performance evaluation between several versions of the NTT, including iterative NTT, parallel NTT, and CUDA-based FFT (cuFFT) for different polynomial sizes [40]. Strictly algorithmic optimizations of the NTT are presented in other works [41-42]. Longa et al. [41] show that limiting the coefficient length in polynomials to 32 bits yields an efficient modular reduction technique. By employing this new technique in NTT, reduction is only required after multiplication, and significant performance gains are achieved when compared to a baseline implementation. Additionally, the authors use signed integer arithmetic which decreases the number of add operations necessary in both sampling and polynomial multiplication. Greconici et al. [42] use signed integer arithmetic to decrease the number of add operations, which leads to performance gains in several functions including NTT and SHAKE-128. The authors also employ a merging layers technique in NTT that reduces the number of loads and stores by about a factor of two.

The SABER KEM algorithm is similarly computationally bound by polynomial multiplication and hashing functions. As mentioned by D'Anvers et al. [18], since SABER uses power-of-2 moduli, this eliminates the need for rejection sampling and makes modular reduction fast by using bit shift operations. However, one drawback of using power-of-2 moduli is the inability to take advantage of faster NTT multiplication since the moduli are not prime. As described above, Akleylek et al. [40] examines the performance of different multiplication techniques. By implementing a version of cuFFT in a similar fashion for SABER, we may observe a speedup in polynomial multiplication. In addition, SABER is computationally bound by hashing and extendible functions. SABER uses SHA3_256 and SHA3_512 functions for hashing and SHAKE128 as an XOF. Roy et al. [43] demonstrate parallelizing SHAKE128 using AVX2 and batching four operations, thus achieving a 38% increase in throughput for SABER's key generation. Additionally, optimizing the hashing functions and SHAKE128 in a different way, the SABER technical documentation describes replacing the SHA3 functions with SHA2 and replacing SHAKE128 with AES in counter mode [18].

Focusing on three PQC algorithms, SABER, CRYSTALS-Dilithium, and NTRU, a breakdown of the fraction of time spent (as a percentage) in the hashing/XOR and polynomial multiplication components of the algorithms is reported in Table 1. NTRU spends the majority of its time doing polynomial multiplication first, then hashing second [44], but no benchmarks have been calculated thus far. The times spent for the hashing and polynomial multiplication components of CRYSTALS-Dilithium, and SABER are reported as percentages of the total execution time for the key pair generation procedure where the percentages are an average of 10 time trials.

Table 1. Breakdown of the fraction of the time hashing and extendable output function XOF compared with the time performing polynomial multiplication. Both SABER and CRYSTALS are constrained by the polynomial multiplication. The light versions considered in the post quantum cryptographic (PQC) implementations of hashing and XOF functions such as SHA3 with SHAKE are extremely effective.

	Hashing and XOF	**Polynomial Multiplication**	**Reference**
SABER	~30%	~60%	Our benchmarks
CRYSTALS-Dilithium	~42%	~33%	Our benchmarks
NTRU	second bottleneck	first bottleneck	[40,41]

4. PUF-Based Key Distribution for PQC

4.1. PUF-Based Key Distribution for LWE Lattice Cryptography

The proposed generic protocol to generate public–private key pairs with PUFs for LWE lattice cryptography is shown in Figure 6. The random number generator (a) is used for the generation of seed $\boldsymbol{a}_{(i)}$, which is public information. However, Seed k that is needed for the generation of the private key $\boldsymbol{Sk}_{(i)}$ is generated from the PUF. The outline of a protocol generating a key pair for LWE cryptography is the following:

1. The CA uses a random numbers generator and hash function to be able to point at a set of addresses in the image of the PUF-i.
2. From these addresses, a stream of bits called Seed K′ is generated by the CA.
3. The CA communicates to the Client (i), through a handshake, the instructions needed to find the same set of addresses in the PUF.
4. Client (i) uses the PUF to generate the stream of bits called Seed K. The two data streams Seed K and Seed K′ are similar, however slightly differ from each other due to natural physical variations and drifts occurring in the PUFs.
5. [If needed, Client (i) applies error correcting codes to reduce the difference between Seed K and Seed K′; the corrected, or partially corrected, data stream is used to generate the vectors $\boldsymbol{s}_{1(i)}$ and $\boldsymbol{s}_{2(i)}$]
6. Client (i) independently uses a random numbers generator (a) to generate a second data stream Seed $\boldsymbol{a}_{(i)}$, which is used for the computation of the matrix $\boldsymbol{A}_{(i)}$.
7. The vector $t_{(i)}$ is computed: $\boldsymbol{t}_{(i)} \leftarrow \boldsymbol{A}_{(i)}\ \boldsymbol{s}_{1(i)} + \boldsymbol{s}_{2(i)}$.
8. The private key $\boldsymbol{Sk}_{(i)}$ is $\{\boldsymbol{s}_{(1(i)}; \boldsymbol{s}_{2(i)}\}$.
9. The public key $\boldsymbol{Pk}_{(i)}$ is $\{\boldsymbol{a}_{(i)}; \boldsymbol{t}_{(i)}\}$.
10. Client (i) communicates to the CA, through the network, the public key $\boldsymbol{Pk}_{(i)}$;
11. The CA uses a search engine to verify that $\boldsymbol{Pk}_{(i)}$ is correct. The search engine initiates the validation by generating a public key from Seed $a_{(i)}$ and Seed K′ with lattice cryptography codes. If the resulting public key is not $\boldsymbol{Pk}_{(i)}$, an iteration process gradually injects errors into Seed K′ and computes the corresponding public keys. The search converges when a match in the resulting public key is found, or when the CA concludes that the public key should be bad.
12. If the validation is positive, the public key $\boldsymbol{Pk}_{(i)}$ is posted online by the RA.

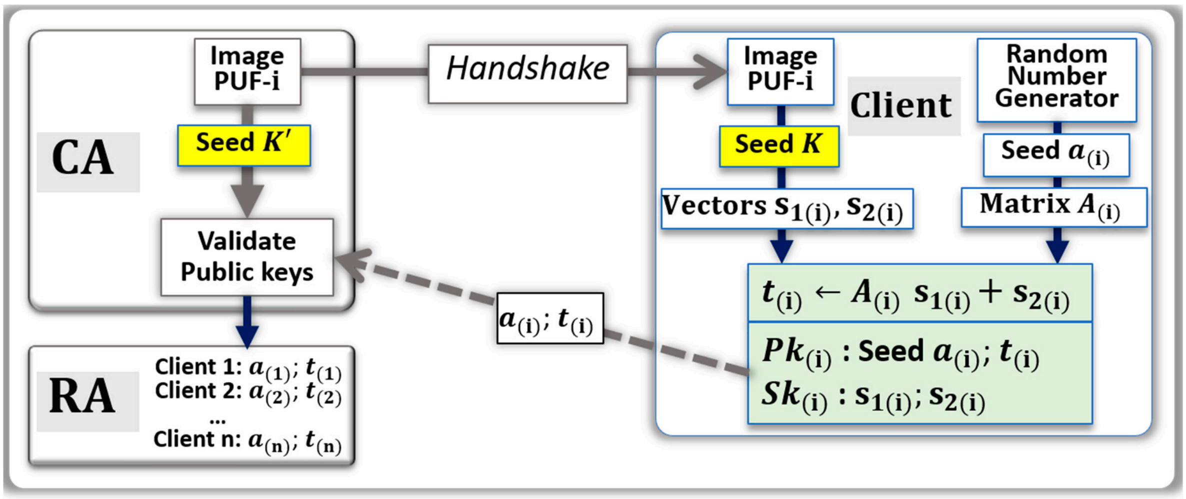
CA
Image PUF-i
Seed K'
Validate Public keys
Handshake
$a_{(i)}$; $t_{(i)}$
Image PUF-i
Client
Random Number Generator
Seed K
Seed $a_{(i)}$
Vectors $s_{1(i)}, s_{2(i)}$
Matrix $A_{(i)}$
$t_{(i)} \leftarrow A_{(i)}\ s_{1(i)} + s_{2(i)}$
$Pk_{(i)}$: Seed $a_{(i)}$; $t_{(i)}$
$Sk_{(i)}$: $s_{1(i)}$; $s_{2(i)}$
RA
Client 1: $a_{(1)}$; $t_{(1)}$
Client 2: $a_{(2)}$; $t_{(2)}$
...
Client n: $a_{(n)}$; $t_{(n)}$

Figure 6. PUF-based key pair generation for learning with error (LWE). The private keys, i.e., vectors $S_{1(i)}$ and $S_{2(i)}$, are generated from the seed K that is extracted from the PUF. The matrix $A_{(i)}$ continues to be generated from a random number. The search engine of the CA has access to an image of the PUF, and can independently validate the validity of public key, which is posted by the RA for cloud-based transactions. A new key pair can be generated, and validated by the CA, at each handshake cycle.

This protocol is applicable for single use key pairs that are generated for each transaction. The random number generators of the first step of the protocol can generate new data streams, which point at different portions of the PUFs, thereby triggering the generation of new key pairs. The search engine described above can benefit from noise injection and high-performance computing. The injection of noise in Seed K will make the search too difficult for CA, unless equipped with HPCs, or GPUs. This can preclude hostile CAs from participating.

4.2. PKI Architecture with PUF-Based Key Distribution and LWE

The PUF-based key pair generation scheme with LWE cryptography, as presented in the previous section, can be integrated in a PKI securing a network of i clients. Figure 7 shows two client devices communicating directly, either by exchanging secret keys through KEM or DSA. The client devices independently generate the seed $\boldsymbol{a}_{(i)}$, while the PUFs and their images are used for the independent generation of the vectors $\boldsymbol{s}_{1(i)}$ and $\boldsymbol{s}_{2(i)}$. The role of the CA is to check the validity of the vectors $\boldsymbol{t}_{(i)}$, and to transmit both the seeds $\boldsymbol{a}_{(i)}$ the vectors $\boldsymbol{t}_{(i)}$ to the RA, which maintain a ledger with valid public keys. Such an architecture is secured assuming the following conditions:

i. The enrollment process in which the PUFs are characterized to generate their image is accurate and not compromised by the opponent.
ii. The database stored in the CA that contains the image of the PUFs for the i client devices is protected from external and internal attacks.
iii. The PUFs embedded in each client device are reliable, unclonable, and tamper resistant.
iv. The key generation process, KEM, and DSA are protected from side channel analysis.

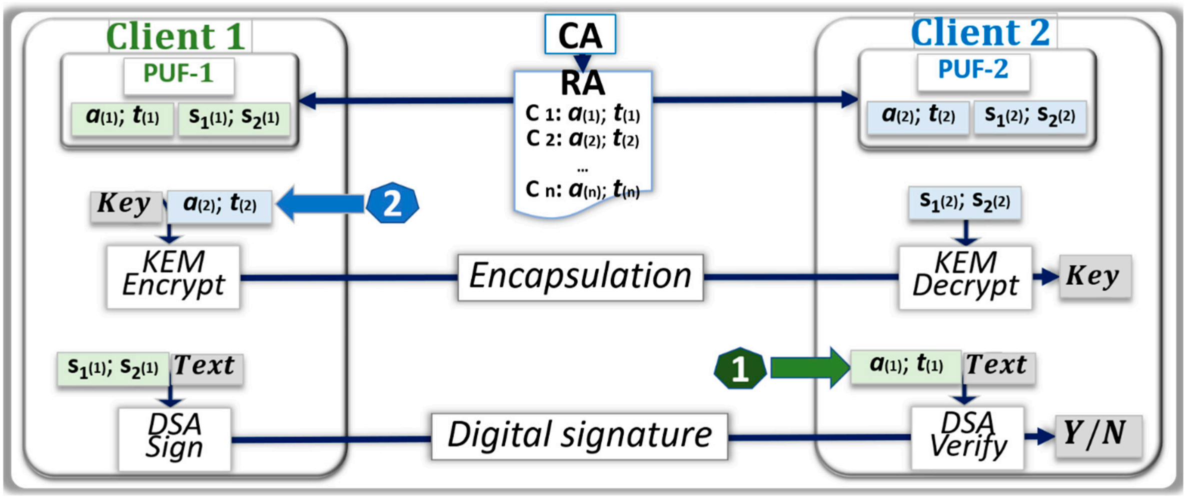

Figure 7. PUF-based public key infrastructure (PKI) with LWE cryptography. The use of PUFs in the PKI network does not impact the user experience during peer-to-peer secure communication. The ability to generate one time use key pairs, and authenticate each client device at each transaction, enhances the root of trust. Such PUF based architectures are only valid if the latencies are kept below a few seconds.

As we experimentally verified that the latencies of the key generation process from the PUFs are low enough, such a protocol can be used to change the key pairs after each encryption cycle. Therefore, the potential loss of the secret keys during an encryption/decryption cycle has minimum impact as different keys will be used during the subsequent cycles.

4.3. PUF-Based Key Distribution for LWR Lattice Cryptography

There are some similarities between LWE and LWR implementations. The seed k of the PUF is only used to generate one vectors $s_{1(i)}$, while a constant vector $h_{(i)}$ can be generated independently. The public vector $t_{(i)}$ is computed in a similar way: $t_{(i)} \leftarrow A_{(i)} . s_{1(i)} + h_{(i)}$.

4.4. PUF-Based Key Distribution for NTRU Lattice Cryptography

The protocol to generate key pairs from PUFs for NTRU cryptography is similar than the one presented above in Section 4.1 for LWE, see Figure 8. We are suggesting a method where the only source of randomness is the PUF, Seed **K**, to compute both the public key $\boldsymbol{Pk_{(i)}}$, and the private key $\boldsymbol{Sk_{(i)}}$. In our implementation, Seed **K** feeds the hash functions SHA-3, and SHAKE, to generate a long stream of bits, then compute the two polynomials $\boldsymbol{f_{(i)}}$ and $\boldsymbol{g_{(i)}}$.

As previously discussed in Section 2.3, the polynomials $\boldsymbol{f_{(i)}}$ and $\boldsymbol{g_{(i)}}$ are not always usable due to pre-conditions, therefore a scheme to try several possible ways of addressing the PUF has to be developed. One way is to implement a deterministic method that is known by both the client device and the CA, which can have a negative impact on the latencies. We preferred the solution driven by the client device that asks the CA to initiate new handshakes. The summary of the method used to generate the key pairs for NTRU cryptography is the following:

1. The CA uses random numbers to point at a set of addresses in the image of the **PUF-i**.
2. From these addresses, a stream of bits called Seed **K′** is generated by the CA.
3. The CA sends the handshake to the client (i) to find the same addresses.
4. Client (i) uses the PUF to generate Seed **K**.
5. Client (i) applies error correction to Seed **K** and generates the truncated polynomials $f_{(i)}$ and $g_{(i)}$.
6. Computation of $\boldsymbol{Fp_{(i)}}$ and $\boldsymbol{Fq_{(i)}}$ and verify that the pre-conditions are fulfilled.
7. If needed, ask for a new handshake and iterate.

8. The polynomial $\boldsymbol{h}_{(i)}$ is computed: $\boldsymbol{h}_{(i)} \leftarrow \mathbf{p} \cdot \boldsymbol{Fq}_{(i)} \cdot \boldsymbol{g}_{(i)}$.
9. The private key $\boldsymbol{Sk}_{(i)}$ is $\{f_{(i)}; \boldsymbol{Fp}_{(i)}\}$.
10. The public key $\boldsymbol{Pk}_{(i)}$ is $\boldsymbol{h}_{(i)}$.
11. Client (i) communicates to the CA, through the network, the public key $\boldsymbol{h}_{(i)}$.
12. The CA uses a search engine to verify that $\boldsymbol{h}_{(i)}$ is correct.
13. If the validation is positive, the public key $\boldsymbol{h}_{(i)}$ is posted online by the RA.

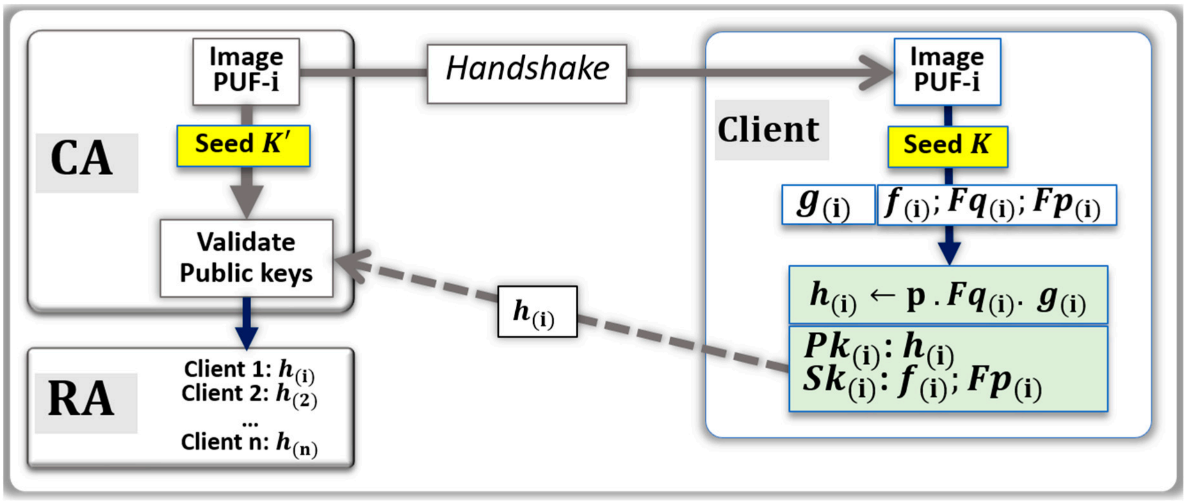

Figure 8. PUF-based key pair generation for NTRU. After each handshake, the polynomials $g_{(i)}$ and $f_{(i)}$ are generated from the Seed K that is extracted from the PUF. The public and private key pairs are computed from these two polynomials. The search engine of the CA can independently validate the validity of public key with the image of the PUF.

It is important to notice that steps six and seven of the proposed method, "Computation of $\boldsymbol{Fp}$(i) and $\boldsymbol{Fq}$(i) and verify that the pre-conditions are fulfilled; if needed ask for a new handshake and iterate", could be handled differently to minimize backward and forward communications cycles between the CA and the client device. One example of implementation is to have a pre-arranged way to modify the seed generated by the PUF and its image. When the CA fails to validate the public key, several pre-arranged modifications of Seed K' will be tested.

4.5. PUF-Based Key Distribution for Code-Based Cryptography

An example of a protocol to generate the key pairs with PUFs for code-based cryptography is shown in Figure 9. The overall protocol is similar to the one presented above for lattice cryptography. Much like NTRU, the only source of randomness is Seed **K** that is generated from the PUF to compute the two matrixes $\boldsymbol{S}_{(i)}$ and $\boldsymbol{P}_{(i)}$. The brief outline of the protocol for generating key pairs for code-based cryptography is the following:

1. The CA uses random numbers to point at a set of addresses in the image of the **PUF-i**.
2. From these addresses, a stream of bits called Seed **K'** is generated by the CA.
3. The CA sent the handshake to the client (i) to find the same addresses.
4. Client (i) uses the PUF to generate Seed **K**.
5. Client (i) applies ECC) on Seed **K** and generates the matrixes $\boldsymbol{S}_{(i)}$ and $\boldsymbol{P}_{(i)}$.
6. Computation of $\boldsymbol{S}_{(i)}^{-1}$ and $\boldsymbol{P}_{(i)}^{-1}$.
7. The public key $\boldsymbol{Pk}_{(i)} = \hat{\boldsymbol{G}}_{(i)}$ is computed with the generator matrix $\boldsymbol{G}$: $\hat{\boldsymbol{G}}_{(i)} \leftarrow \boldsymbol{S}_{(i)} \cdot \boldsymbol{G} \cdot \boldsymbol{P}_{(i)}$.
8. The private key $\boldsymbol{Sk}_{(i)}$ is $\{\boldsymbol{G}; \boldsymbol{S}_{(i)}^{-1}, \boldsymbol{P}_{(i)}^{-1}\}$.
9. Client (i) communicates to the CA, through the network, the public key $\hat{\boldsymbol{G}}_{(i)}$.
10. The CA uses a search engine to verify that $\hat{\boldsymbol{G}}_{(i)}$ is correct.
11. If the validation is positive, the public key $\hat{\boldsymbol{G}}_{(i)}$ is posted online by the RA.

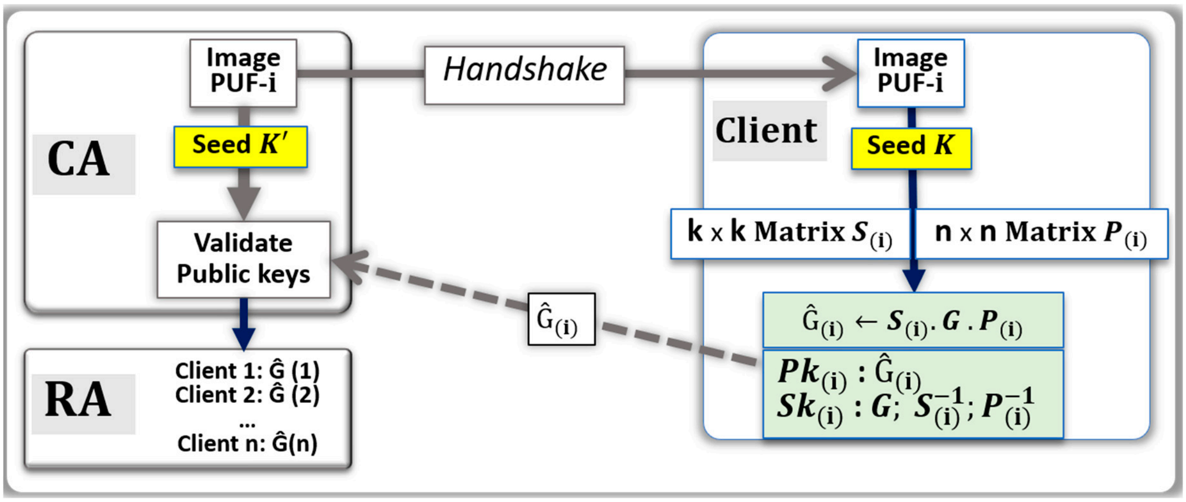

Figure 9. PUF-based key pair generation for code-based cryptography. The two matrixes $S_{(i)}$ and $P_{(i)}$ are generated from the Seed K that is extracted from the PUF. As was done in the previously presented PQC schemes, the search engine of the CA has access to an image of the PUF and can independently verify the validity of the public key.

Step six, the computing of inverses, does not work for all matrixes: request for new handshakes, or shared ways to find invertible matrixes and iterate.

5. Experimental Evaluation

The purpose of this evaluation is to demonstrate the practicality of the proposed protocols and understand the potential limitations. The replacement of the random number generators by the PUFs follow a similar path for various PQC algorithms; therefore we reduced the scope of this evaluation to LWE (qTESLA, CRYSTALS-Dilithium), and LWR (LightSABER). The generation of the Seed K from the PUF is done using known methods, and the computation of the key pairs is based on the PQC codes made available by NIST, which are also considered known. The unknown in the practicality of the protocol is the sensitivity to bit error rates (BER) of the search engine of the CA for verifying the public keys. The RBC method [34–37] uses the Seed K′ as a starting point to generate an initial public key, then iterates by incrementally adding errors, eventually finding the public key computed from Seed K by the client device. At high BER, it is desirable to use cryptographic algorithms that have the ability to generate the key pairs at high throughput.

The RBC itself is an interesting simulation platform for this evaluation, because of the possibility to directly measure the throughput in term of the number of key pairs generated by second. We selected the RBC to experimentally demonstrate that the PQC protocols are fast enough. We designed an experiment to benchmark three algorithms, (qTESLA, CRYSTALS-Dilithium, and LightSABER) with two known cryptosystems (AES and ECC). Each of these cryptosystems have a list of parameter sets as a part of their specifications. We chose parameter sets that were inherently compatible with a 256-bit output from a hypothetical PUF as well as these that were best optimized between performance, size, and security for IoT devices. For these reasons, the parameter sets AES256, ECC Secp256r1, qTESLA-p-I, CRYSTALS-Dilithium 2, and LightSABER were used for the performance comparison. In this analysis, the comparison with ECC is the most relevant one, because the PQC codes under consideration and ECC are similar in their objective to generate public–private keys pairs for PKI. Therefore, ECC and the three PQC algorithms are tested here with their software versions. The comparison with AES was included as a benchmark of excellence; we used the hardware implementation of AES, natively available in Intel processors. One of the objectives of the PQC standardization program driven by the NIST is to encourage private industry to eventually design hardware implementations of the selected codes.

We summarize the parameter sets for each algorithm and our motivation for their selection in Table 2, as shown below.

Table 2. Selected cryptosystems, parameter sets, whether the PQC algorithms are NIST round 3 candidates, specialized instructions employed in the implementations, and our motivation for selecting the algorithm and its configuration.

Algorithm	Parameter Set	NIST Candidate	Instructions	Selection Reason
AES	AES256	N/A	AES-NI, SSE	Benchmark: HW implementation Lacks DSA and KEM capabilities
ECC	Secp256r1	N/A	AVX, SSE	Benchmark: mainstream for PKI Uses 256-bit long keys for DSA
qTESLA	p-I	Dropped	AVX, SSE	PQC dropped by NIST: too slow DSA uses relatively small keys
CRYSTALS-Dilithium	2	Phase 3	AVX, SSE	Active LWE PQC algorithm One of preferred DSA scheme
SABER	LightSABER	Phase 3	AVX, SSE	Active LWR PQC algorithm One of the preferred KEM scheme

5.1. Experimental Methodology

As of the time of writing, there are few implementations of RBC engines proposed. In this paper, we focus on executing the RBC protocol on a single machine equipped with multi-core CPUs. Our implementations are parallelized using OpenMP. To terminate the search when a thread finds the correct key, we use a flag in shared memory that is atomically updated. All implementations utilized the same overall structure and key iteration mechanism. We also use AVX instructions in all cryptosystems where applicable; however, further optimizations can be made by taking advantage of AVX2 or other wide vector technologies. The AES256 implementation takes advantage of the AES-NI instruction set, whereas all other cryptosystems tested do not use any additional vectorized instructions except AVX and SSE.

RBC engines targeting purely CPU platforms were only considered for demonstrative purposes. The purpose of this experiment is to compare the relativistic performance between all five chosen cryptosystems. The ease of porting one cryptosystem to another all on the CPU influenced the scope of experiments. Future experimental evaluations exploring GPU focus will require more dedicated, specialized programming for each cryptosystem. The CPU used for the experiments was a 2× Xeon Gold 6132 (Skylake) CPU with 28 total physical cores. Experiments were executed on a dedicated platform. All codes were written in C/C++, compiled using O3 optimization flag.

The experiments were executed by randomly selecting a target thread and using the 256-bit permutation that is the middle of that given thread's workload. This guarantees that each run accurately reflects the average case where execution stops halfway through the 256 choose k search space, for any Hamming distance k. We decided to use this approach to reduce the need for a high number of iterations to reach a statistical central point. Thus, 10 iterations were performed for each cryptosystem, and the median response time was selected from the set of 10 time trials.

The major key performance index (KPI) of the RBC search is key search throughput. Therefore, our performance evaluation uses the "effective key throughput" performance metric. Since the search increases exponentially with Hamming distance, the fraction of time spent initializing/dismantling our procedure will dominate the response time on small workloads (small Hamming distances). Consequently, to measure the effective key throughput, we use a sufficiently large Hamming distance in each algorithm such that we observe peak throughput, indicating that the initialization and dismantling procedures (freeing memory, deconstructing objects, etc.) constitute a negligible fraction of the total response time.

For AES256, the minimum Hamming distance is 4, while for ECC Secp256r1, qTESLA-p-I, CRYSTALS-Dilithium 2, and LightSABER the minimum Hamming distance is 3. Unfortunately, due to the intractable nature of the problem, the single bit error jump from a Hamming distance of 3 to 4 makes it impractical to run a statistically sufficient number of runs for ECC and qTESLA-p-I. For this reason, the AES256 benchmarks ran at a Hamming distance of 4, and the remaining cryptosystems ran at a distance of 3.

5.2. Evaluation of the Effective Key Throughput

In this section, we evaluate the effective (peak) key throughput. Figure 10 plots the median of each RBC cryptosystem's effective key throughput on a logarithmic scale. The AES256 implementation, aided by AES-NI, runs several orders of magnitude more efficiently than the public key cryptography variants at 2.17 10^8 keys per second. ECC Secp256r1 performed the second slowest at 4.77 10^4 keys per second. The post-quantum algorithms largely performed better than ECC with 1.97 10^5 and 6.83 10^5 keys per second for CRYSTALS-Dilithium 2 and LightSABER respectively. qTESLA-p-I was the worst performing PQC and overall cryptosystem out of all five at 2.24 10^4 keys per second.

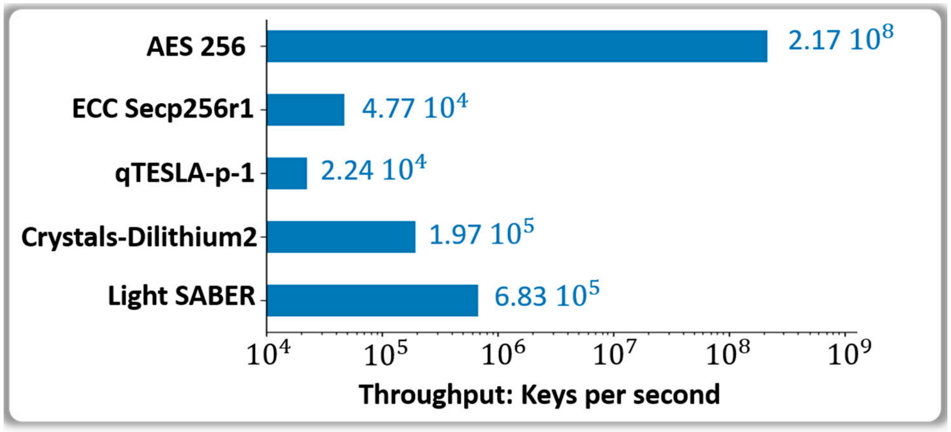

Figure 10. Key performance index (KPI) and the effective throughput in keys per second achieved by the response based cryptographic (RBC) search engine powered with AMD Ryzen 9 3900X. Benchmark of the post quantum cryptographic (PQC) algorithms are compared to the reference codes AES 256 and ECC.

To get a better sense of the relativistic scaling, we set ECC Secp256r1's effective key throughput results as the reference point since we are interested in how the PQC algorithms perform when replacing it in future PKI cryptosystems. This is plotted in Figure 11, where now the response variable is displayed in a percentage of the throughout relative to ECC Secp256r1's. Shown here, AES256 is roughly 4550 times more performant than ECC Secp256r1. CRYSTALS-Dilithium 2 is over 4.14 times more efficient than ECC Secp256r1. The most efficient PQC was LightSABER at 14.3 times faster, and the worst overall cryptosystem was qTESLA-p-I at 0.469 times slower.

From these results, we confirm NIST's position that qTESLA is slower than the algorithms selected in round three. Out of what was tested, this leaves CRYSTALS-Dilithium as the strongest candidate for DSA in a PQC environment. For key encapsulation, our results show that SABER is a strong candidate for its relatively fast key generation. Future testing might consider comparing FALCON against CRYSTALS-Dilithium for DSA, and CRYSTALS-Kyber, NTRU, and Classic McEliece against SABER for KEM.

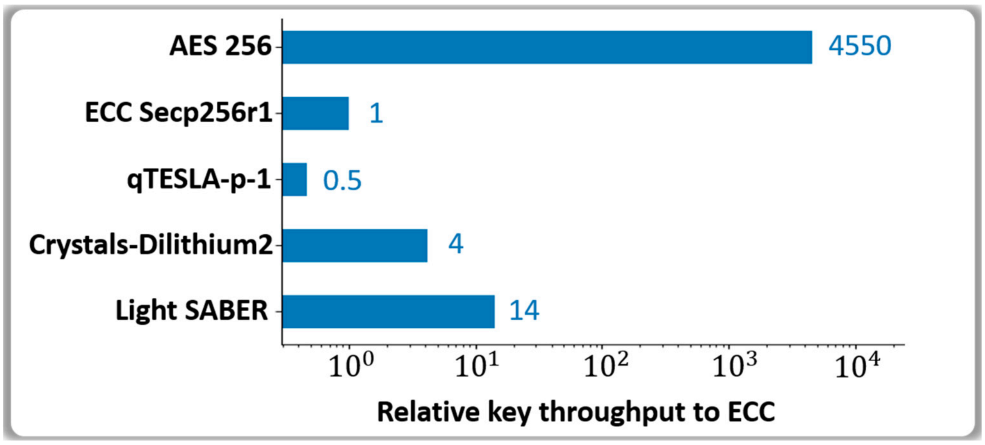

Figure 11. Maximum effective throughput relative to the performance of ECC Secp256r1 achieved for each response based cryptographic (RBC) cryptosystem implementation powered by AMD Ryzen 9 3900X. Light SABER performance is approximately 14 times faster than elliptic curve cryptography (ECC).

6. Conclusion and Future Work

The PQC algorithms under standardization are encouraging and the latencies are reasonable, making the protocols suitable for PKIs securing networks of client devices and IoTs. The generation, distribution, and storage of the public–private key pairs for PQC can be complex because the keys are usually very long. This paper proposes to generate the public–private key pairs by replacing the random number generators with data streams generated from addressable PUFs to get the seeds needed in the PQC algorithms. Unlike the key pairs computed by PQC algorithms, the seeds are relatively short, typically 256-bits long. The use of PUFs as a source of randomness is applicable to all five lattice-based codes under consideration in the phase III investigation of NIST, and to the code-based Classic McEliece scheme. In order to simultaneously generate key pairs from a server acting as the certificate authority, and the client device with access to its PUF, it is critical to handle the bit error rates (BERs) that are frequent with physical elements. We verified in the experimental section that the RBC can find the erratic seeds by testing an excess of 10^5 seeds per second with CRYSTALS-Dilithium 2 and LightSABER, which is faster than what we measured with mature algorithms such as the ones with elliptic curves. The experimental evaluation conducted in this work, with the RBC, also lets us conclude that the pre-selection by NIST of CRYSTALS-Dilithium for DSA and SABER for KEM are promising from a performance standpoint. Our results show that the key generation performance is at least comparable to that of ECC. The PQC algorithms under consideration are excellent in an environment targeting PUF-based key exchange. The AES hardware-accelerated AES-NI implementation yields roughly 220 million keys per second throughput on a single machine, which serves a practical real world upper bound for future hardware-accelerated PQC implementations.

In this work we have not yet studied the multivariate-based RAINBOW code, which is also an important scheme under consideration for standardization; we are currently studying ways to use PUFs for key generation. The task needed to deploy PUF-based PQC solutions is not underestimated by the authors of this paper. This will include the use of highly reliable PUFs, and the optimization of the cryptographic protocol pointing simultaneously at the same set of addresses in the PUF, and in the look up table capturing the challenge–response pairs stored in the server. Further optimizing the PUF's protocols and the RBC for PQC algorithms is seen as an opportunity. The use of noises, nonces, errors, and rounding vectors can exploit the stochasticity of PUFs, and the ability to handle erratic streams of the RBC. The PQC algorithms analyzed in this paper can also benefit from the use of distributed memories, high performance computing, and parallel computing, which have the potential to further reduce the latencies of the RBC.

Author Contributions: Conceptualization, B.C.; methodology, B.C., M.G., B.Y.; software, M.G., D.G., K.L., S.N., C.P., A.S., J.W.; validation, C.P.; formal analysis, M.G., B.Y., D.G., K.L., S.N., C.P., A.S., B.C.; investigation, M.G., B.Y., D.G., K.L., S.N., C.P., A.S., B.C.; resources, B.C., M.G., B.Y.; data curation, M.G., B.Y., D.G., K.L., S.N., C.P., A.S., B.C.; writing—original draft preparation, B.C., D.G., K.L., S.N., C.P., J.W.; writing—review and editing, B.C., C.P., M.G.; visualization, M.G., B.Y., D.G., K.L., S.N., C.P., A.S., B.C.; supervision, B.C., M.G., B.Y.; project administration, B.C..; funding acquisition, B.C., M.G. All authors have read and agreed to the published version of the manuscript.

Funding: This material is based upon the work funded by the Information Directorate under AFRL award number FA8750-19-2-0503.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors thank the staff, students, and faculty from Northern Arizona University (NAU) in particular, Brandon Salter who is a software engineer in NAU's cybersecurity lab. We also thank the professionals of the Information Directorate of the Air Force Research laboratory (AFRL) of Rome, New York (US), who supported this effort. **disclaimer:** (a) Contractor acknowledges Government's support in the publication of this paper. This material is partially based upon the work funded by the Information Directorate, under the Air Force Research Laboratory (AFRL); (b) any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study, collection, analyses, interpretation of data, writing of the manuscript, or decision to publish the results.

References

1. Koblitz, N.; Menezes, A. A Riddle Wrapped in an Enigma. 2015. Available online: http://eprint.iacr.org/2015/1018 (accessed on 18 May 2015).
2. Kiktenko, E.; Pozhar, N.; Anufriev, M.; Trushechkin, A.; Yunusov, R.; Kurochkin, Y.; Lvovsky, A.; Fedorov, A. Quantum Secured Blockchains. Open Source. *arXiv* **2018**, arXiv:1705.09258v3.
3. Semmouni, M.; Nitaj, A.; Belkasmi, M. Bitcoin Security with Post Quantum Cryptography. 2019. Available online: https://hal-normandie-univ.archives-ouvertes.fr/hal-02320898 (accessed on 19 October 2019).
4. Campbell, R. Evaluation of Post-Quantum Distributed Ledger Cryptography. Open Access, JBBA. 2019; Volume 2. Available online: https://doi.org/10.31585/jbba-2-1-(4)2019 (accessed on 16 March 2019).
5. Kampanakisy, P.; Sikeridisz, D. Two Post-Quantum Signature Use-cases: Non-issues, Challenges and Potential Solutions. In Proceedings of the 7th ETSI/IQC Quantum Safe Cryptography Workshop, Seattle, WA, USA, 5–7 November 2019.
6. Ding, J.; Chen, M.-S.; Petzoldt, A.; Schmidt, D.; Yang; B-Y. *Rainbow*; NIST PQC project round 2, documentation. In Proceedings of the 2nd NIST Standardization Conference for Post-Quantum Cryptosystems, Santa Barbara, CA, USA, 22–24 August 2019.
7. NIST status report of phase 3 of PQC program, NISTIR.8309, Available online: https://www.nist.gov/publications/status-report-second-round-nist-post-quantum-cryptography-standardization-process (accessed on 22 July 2020).
8. Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schwabe, P.; Seiler, G.; Stehlé, D. CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation. Part of the Round 3 Submission Package to NIST. Available online: https://pq-crystals.org/dilithium (accessed on 19 February 2021).
9. Fouque, P.-A.; Hoffstein, J.; Kirchner, P.; Lyubashevsky, V.; Pornin, T.; Prest, T.; Ricosset, T.; Seiler, G.; Whyte, W.; Zhang, Z. Falcon: Fast-Fourier Lattice-Based Compact Signatures over NTRU, Specification v1.2. Available online: https://falcon-sign.info/falcon.pdf (accessed on 1 October 2020).
10. Peikert, C.; Pepin, Z. *Algebraically Structured LWE Revisited*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 1–23.
11. IEEE Computing Society. *IEEE Standard 1363.1-2008—Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices*; IEEE: Piscataway, NJ, USA, 2009.
12. Regev, O. New lattice-based cryptographic constructions. *J. ACM* **2004**, *51*, 899–942.
13. Casanova, A.; Faugere, J.-C.; Macario-Rat, G.; Patarin, J.; Perret, L.; Ryckeghem, J. *GeMSS: A Great Multivariate Short Signature*; NIST PQC project round 2, documentation. Available online: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions (accessed on 3 January 2017).
14. McEliece, R.J. *A Public-Key Cryptosystem Based on Algebraic Coding Theory*; California Inst. Technol.: Pasadena, CA, USA, 1978; pp. 114–116.

15. Biswas, B., Sendrier, N. McEliece Cryptosystem Implementation: Theory and Practice. In *Post-Quantum Cryptography. PQCrypto. Lecture Notes in Computer Science*; Buchmann, J., Ding, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; volume 5299.
16. Regev, O. On lattices, learning with errors, random linear codes, and cryptography. In Proceedings of the thirty-seventh annual ACM symposium on Theory of computing—STOC '05; ACM: New York, NY, USA, 2005; pp. 84–93.
17. Lyubashevsky, V. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. Available online: https://www.iacr.org/archive/asiacrypt2009/59120596/59120596.pdf, (accessed on 31 December 2009).
18. D´Anvers, J.-P.; Karmakar, A.; Roy, S.; Vercauteren, F. Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM. Cryptology ePrint Archive, Report 2018/230. Available online: https://eprint.iacr.org/2018/230 (accessed on 7 May 2018).
19. Bos, J.; Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schanck, J.; Schwabe, P.; Seiler, G.; Stehle, D. CRYSTALS—Kyber: A CCA-Secure Module-Lattice-Based KEM. In Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P), London, UK, 24–26 April 2018; pp. 353–367.
20. Hülsing, A.; Rijneveld, J.; Schanck, J.; Schwabe, P. High-speed key encapsulation from NTRU. IACR Cryptol. Available online: https://www.iacr.org/archive/ches2017/10529225/10529225.pdf (accessed on 28 August 2017).
21. Banerjee, A.; Peikert, C.; Rosen, A. Pseudorandom functions and lattices. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 719–737.
22. Alwen, J.; Stephan, K.; Krzysztof, P.; Daniel, W. Learning with rounding, revisited. In *Proceedings of the Annual Cryptology Conference*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 57–74.
23. Nurshamimi, S.; Kamarulhaili. H. NTRU Public-Key Cryptosystem and Its Variants. *Int. J. Cryptol. Res.* 2020, *10*, 21.
24. Gentry, C.; Peikert, C.; Vaikuntanathan. Trapdoors for Hard Lattices and New Cryptographic Constructions. Proc. of the 14th annual ACM symposium on Theory of computing. Pages 197–206. Available online: https://doi.org/10.1145/1374376.1374407 (accessed on 25 August 2008).
25. Heyse, S. *Post-Quantum Cryptography: Implementing Alternative Public Key Schemes on Embedded Devices*; For the degree of Doktor-Ingenieur of the Faculty of Electr. Eng. and Information Technology at the Ruhr-University Bochum: Bochum, Germany, 2013.
26. Menezes, A.; van Oorschot, P.; Vanstone, S. *Some Computational Aspects of Root Finding in GF(q^m)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1989; Volume 358.
27. Papakonstantinou, I.; Sklavos, N. *Physical Unclonable Function Design Technologies: Advantages & Tradeoffs*; Computer and Network Security; Daimi, K., Ed.; Spinger: Berlin/Heidelberg, Germany, 2018; ISBN: 978-3-319-58423-2.
28. Herder, C.; Yu, M.; Koushanfar, F. Physical Unclonable Functions and Applications: A Tutorial. *IEEE* 2014, *102*, 1126–1141.
29. Cambou, B.; Orlowski, M. Design of Physical Unclonable Functions with ReRAM and ternary states. In Proceedings of the Cyber and Information Security Research Conference, CISR-2016, Oak Ridge, TN, USA, 5–7 April 2016.
30. Cambou, B.; Telesca, D. Ternary Computing to Strengthen Information Assurance, Development of Ternary State based public key exchange. In Proceedings of the SAI-2018, Computing Conference, London, UK, 10–12 July 2018.
31. Taniguchi; M; Shiozaki, M.; Kubo, H.; Fujino, T. A stable key generation from PUF responses with a Fuzzy Extractor for cryptographic authentications. In Proc. of the IEEE 2nd Global Conf. on Cons Electronics (GCCE), Tokyo, Japan, 1-4 Oct. 2013.
32. Kang, H.; Hori, Y.; Katashita, T.; Hagiwara, M.; Iwamura, K. Cryptography key generation from PUF data using efficient fuzzy extractors. In Proceedings of the 16th Int. Conf. on Adv. Communication Technology, Pyeongchang, Korea, 16–19 Feb. 2014.
33. Delvaux, J.; Gu, D.; Schellekens, D.; Verbauwhede, I. Helper Data Algorithms for PUF-Based Key Generation: Overview and Analysis. *IEEE Trans. Comput. Des. Integr. Circuits Syst.* 2014, *34*, 889–902.
34. Cambou, C.; Philabaum, D.; Booher, D. Telesca; Response-Based Cryptographic Methods with Ternary Physical Unclonable Functions. In Proceedings of 2019 SAI FICC conference, San Francisco, CA, USA. 14–15 March 2019.
35. Cambou, B. Unequally powered Cryptograpgy with PUFs for networks of IoTs. In Proceedings of the IEEE Spring Simulation Conference, Tucson, AZ, USA, 29 April–2 May 2019.
36. Cambou, B.; Philabaum, C.; Booher, D. Replacing error correction by key fragmentation and search engines to generate error-free cryptographic keys from PUFs. *CryptArchi, Prague, Czech Republic*, Jun. 24th, 2019.
37. Cambou, B.; Mohammadi, M.; Philabaum, C.; Booher, D. Statistical Analysis to Optimize the Generation of Cryptographic Keys from Physical Unclonable Functions. Available online: https://link.springer.com/chapter/10.1007/978-3-030-52243-8_22 (accessed on 16 July 2020).
38. Nejatollahi, H.; Dutt, N.; Ray, S.; Regazzoni, F.; Banerjee, I.; Cammarota, R. Post-Quantum Lattice-Based Cryptography Implementations. *ACM Comput. Surv.* **2019**, *51*, 1–41.
39. Emeliyanenko, P. Efficient multiplication of polynomials on graphics hardware. In Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies, Rapperswil, Switzerland, 24–25 August 2009; pp. 134–149.
40. Akleylek, S.; Dağdelen, Ö.; Tok, Y. On the efficiency of polynomial multiplication for lattice-based cryptography on gpus using cuda. In *Cryptography and Information Security in the Balkans*; Springer Int. Publishing: Berlin/Heidelberg, Germany, 2016.
41. Longa, P.; Naehrig, M. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. *Comp. Sci. Math. IACR* **2016**, 124–139, doi:10.1007/978-3-319-48965-0_8.
42. Greconici, D.; Kannwischer, M.; Sprenkels, D. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Cryptol. ePrint Arch.* **2020**, *1278*, 2020.

43. Roy, S. SaberX4: High-Throughput Software Implementation of Saber Key Encapsulation Mechanism. In Proceedings of the 37th IEEE International Conference on Computer Design, ICCD 2019, Abu Dhabi, UAE, 17–20 November 2019; pp. 321–324.
44. Farahmand, F.; Sharif, M.; Briggs, K.; Gaj, K. A High-Speed Constant-Time Hardware Implementation of NTRUEncrypt SVES. In Proceedings of the International Conference on Field-Programmable Technology (FPT), Naha, Okinawa, Japan, 10–14 December 2018; pp. 190–197, doi:10.1109/FPT.2018.00036.