An Overview of Distributed Public Key Infrastructures (DPKIs)

Nitesh Kartha

May 6, 2022

Abstract

In recent years, the current public key infrastructure system (known as X.509) has been criticized for its reliance on centralized certificate authorities which creates single points of failure which can be exploited by attackers to allow for forged public key certificates. As a result, researchers have begun experimenting with decentralized PKIs as a way to remove the trusted third parties and expand scalability, typically through the use of blockchain platforms like Ethereum. In this paper, I will discuss in detail, the smart contract based approach discussed by Patsonakis et al. in [PSRK17] and [PSKR19] which is the first DPKI system to have a security proof based off the strong RSA assumption and has been tested on an Ethereum testnet. I will also discuss QChain, the quantum-resilient blockchain DPKI proposed by An and Kim in [AK18] which is based on GLP signatures and the Ring-LWE problem. Although these schemes are not perfect and cannot replace the current X.509 system, they provide a basis for future work that can hopefully lead to a more decentralized and scalable public key system.

1 Introduction

Public key infrastructure (PKI) is defined by NIST as "a set of policies, processes, software, and workstations used for the purpose of administering certificates and public-private key pairs, including the ability to issue, maintain, and revoke public key certificates." [GGF17] PKIs serve as an important aspect of public-key cryptography, as they associate identities with public keys and allow users to validate the accuracy of the public keys they use every day in services like the World Wide Web. However, the predominant form of public key infrastructure used today, X.509, relies on centralized certificate authorities (CAs). This creates single points of failure, which when exploited, have led to attackers forging certificates. Examples of these incidents include Operation Black Tulip which attacked DigiNotar, one of the largest certificate authorities in the Netherlands, and Monpass, a major certificate authority in Mongolia which was attacked by hackers who used a backdoor to spread malware.[Pri11] [CMV21]

In recent years, several researchers have proposed decentralized versions of PKIs (DPKIs), many of which use blockchains and other forms of distributed consensus to provide the same functionality of centralized systems like X.509 while also providing improved scalability and removing singe points of failure for a more secure system. In this overview, I will discuss in detail the Smart Contract DPKIs introduced by Patsonakis et al. which were the first provably secure DPKIs introduced and based on public-state additive universal cryptographic accumulators [PSRK17] and later Merkle hash trees [PSKR19] and Q-chain, a quantum-resistant blockchain-based DPKI by An and Kim which uses lattice-based GLP signatures.[AK18]

2 Preliminaries

Standard notions and definitions are mentioned in Appendix A. For [PSRK17], the standard notion of PRG, 2-Universal Hash Function Family, and the Strong RSA Assumption are required as well as the Random Oracle Model, which are formally defined in Appendix A. For [AK18], Ring-LWE and LWE problems are required which are formally defined in Appendix B which are used as a basis for the GLP signature scheme discussed in this section. The notion of public-state, universal additive cryptographic accumulators is discussed in the next section as well as Appendix C.

2.1 Public Key Infrastructure (X.509)

This paper primarily compares DPKI systems to the current, most popular PKI certificate system, X.509. X.509 is a centralized system first introduced in 1988 and involves centralized certificate authorities (CA) which authenticate and issue certificates for public keys to other entities. CAs themselves have a hierarchy of trust, although that is beyond the scope of this paper. Similarly, registration agents (RAs) are used to register a public key to an entity which is then certified/verified by the CA. As mentioned in the introduction, this results in single points of failure which can be used by an adversary to forge certificates, thus allowing for malicious communication under false identities. For more information about the general structure of X.509 system, refer to this NIST publication [KHPC01].

2.2 Blockchains

Blockchains were first introduced as a method to ensure document timestamp integrity by Bayer, Stornetta, and Haber in 1993 [BSH93]. They consist a growing list of records (known as blocks) that are linked together using a cryptographic hash of the previous block and typically contains transaction information and timestamps (generally represented as a Merkle tree with data nodes represented as leafs). However, the primitive was not popularized until 2008 when it became the foundation of the cryptocurrency Bitcoin. [Nak09] This later inspired more complex platforms like Ethereum which allow for smart contracts, or self-executing protocols, to be run by the decentralized network of miner nodes whenever a certain condition is met. [But13]

In this section, I will discuss the general structure of a blockchain which will helpful to understand QChain [AK18] as well as Ethereum since that is the blockchain platform that [PSRK17] and [PSKR19] used to test the practicality of a DPKI system (which is explained further in the succeeding sections).

2.2.1 General Structure of Blockchain

Blockchains were first popularized by Bitcoin, a decentralized P2P network that runs a virtual currency devised by Nakamoto in 2008 [Nak09]. All users on the blockchain network can create a transaction (whether it is an actual transfer of Bitcoin between accounts or a registration of an id-pk pair, etc.) which are then generated into blocks using a mechanism called Proof-of-Work (note that PoW itself is a rather large topic and beyond the scope of this paper). Just note that nonce is a random value used by PoW to validate blocks. Similarly, blockchains are said to be *immutable*, which means the state of the system (and therefore the history of transactions) cannot be changed once verified (although there can be exceptions, see [Lei17] and the discussion on Ethereum Classic at the end of the paper). Once PoW verifies a block, it is broadcasted to the entire network and registered as part of the "chain".

Each block header has the hash value of the previous block and the transactions of each block are used to generate a Merkle hash tree (to save space and ensure validation of transactions) and the first block is the Genesis block which is hardcoded to ensure the start of the chain. In essence, the P2P nature of the blockchain ensures that there is no need for a trusted third party, which is why it has been proposed as a way to construct a decentralized PKI.

2.2.2 Ethereum and Smart Contracts

Ethereum is a cryptocurrency and open-sourced smart contract platform created and maintained by the Ethereum Foundation. Similar to other cryptocurrencies, Ethereum consists of a peer-to-peer network of miner nodes that validate and execute transactions in exchange for "mined" Ethereum and using networking consensus protocols and cryptographic puzzles known as "proof of work" to ensure that the transactions cannot be tampered with. However, Ethereum is unique in that it has the Ethereum Virtual Machine (EVM) that allows for states to be managed by the Ethereum network and for arbitrary computation to be executed by participant nodes. These arbitrary computations can be constructed as *smart contracts* and programmatically executed by nodes whenever conditions are met. This is the main mechanism in which DPKIs have been proposed to run on blockchain platforms.

In order to pay for the computational effort required to execute a smart contract, a person must pay a "gas" fee to the miner who ends up executing/validating their computation (this has changed with EIP-1559 [Sta21] but for the purposes of the paper, this simplified view of gas fees suffices). As a result, more computationally heavy operations will cost the user more money in the form of gas. For more information about gas fees or Ethereum in general, refer to Section 3.4 as well as the Ethereum docs themselves.

2.3 Distributed Public Key Infrastructure (DPKI)

There is no formally agreed upon definition of what a distributed public key infrastructure (DPKI) system is. In general, it should fulfill the principles of non-repudiation (a user cannot successfully deny that data was signed by their id-pk pair if registered by the system) and revocation (only owners of the id-pk pair should be allowed to deregister/revoke their id-pk registration given a certain condition is met). More specifically, Patsonakis et al. define an "ideal" functionality for what their smart contract DPKI system is in order to formalize what they wish to accomplish. I include a summary of the formalized ideal functionality below which can be used to generalize what DPKI systems should accomplish. Similarly, DPKIs should be able to replicate the same features as the current X.509 infrastructure. Similarly, this model assumes that there is a set of corrupted clients C_{corr} from the initialization and this set is immutable, which can be unrealistic for real-world applications. For more information, see the original paper.

2.3.1 Ideal Naming Service Functionality (F_{ns}) from [PSRK17]

The naming service F_{ns} interacts with n clients, m servers, a trusted party T for setup, and an adversary S called the simulator. This adversary controls a set of corrupted clients C_{cor} which allows it to change the requests sent by those clients. The adversary is also allowed to see all the requests made by any client and the responses by the service, but is not allowed to change the output of the system. Note that relying on a trusted party T for setup does not create a single point of failure since this can be replaced with a distributed protocol which generates the parameters necessary for setup. The system stores (id, public-key) pairs and supports the following operations:

- **Registering pairs**: A client can try to register an (id, public-key) pair if the following conditions are met:

  - The simulator allows the pair to be registered and there is no existing registration associated with the id.
  - The simulator/adversary sends a corrupted request with (id', public-key') when the client is in C_{cor} and there is no existing registration associated with the id'.

  Success/failure messages are sent back to the client via public delayed output.

- **Revoking registrations**: A client can try to revoke an (id, public-key) pair. In order to prevent malicious revocations, there is a revocation relationship R which must be achieved for the request to succeed (i.e. a NIZK proof, etc.). If the simulator allows/corrupts the request and the revocation relationship R is achieved, then the registration is revoked and success is sent back via public delayed output.

- **Retrieving/Verifying pairs**: Assuming that the simulator allows the request to go through, the system will return the public-key, verify that an id has a registration associated with it, and verify that a given mapping of (id, public-key) is valid.

In [PSKR19], this functionality is implemented using smart contracts or a blockchain where miners act as the servers that manage this service. In [AK18], the authors attempt to allow for this functionality through a purpose-built quantum resilient blockchain.

2.4 GLP Signature Scheme (as defined in [AK18] and [GLP12])

One of the major components of the lattice-based blockchain DPKI proposed by [AK18] utilizes a modified version of the GLP signature scheme introduced in [GLP12] by Güneysu et al. based on the Ring-LWE problem. The major difference between the scheme used by [AK18] and the GLP scheme is the use of Number Theoretic Transformations to optimize some calculations. For more information on the Ring-LWE problem (which is similar to the LWE problem) as well as the modified scheme used by An and Kim, see Section 4 and Appendix B and E.

Algorithm 1 GLP Signature Scheme from [GLP12] and [AK18]

```
**Signing Key:** $s_1, s_2 \xleftarrow{R} \mathcal{R}_1^{p^n}$.
**Verification Key:** $a \xleftarrow{R} \mathcal{R}^{p^n}$, $t \leftarrow as_1 + s_2$
**Hash function:** $H : \{0,1\}^* \rightarrow D_{32}^n$
$\mathsf{Sign}(\mu, a, s_1, s_2)$ :
    1. $y_1, y_2 \xleftarrow{R} \mathcal{R}_k^{p^n}$;
    2. $c \leftarrow H(ay_1 + y_2, \mu)$;
    3. $z_1 \leftarrow s_1 c + y_1$;
    4. $z_2 \leftarrow s_2 c + y_2$;
**if** $z_1 \notin \mathcal{R}_{k-32}^{p^n}$ or $z_2 \notin \mathcal{R}_{k-32}^{p^n}$ **then**
    Return to step 1;
**else**
    return $(z_1, z_2, c)$;
**end if**

$\mathsf{Verify}(\mu, z_1, z_2, c, a, t)$ :
**if** $z_1, z_2 \in \mathcal{R}_{k-32}^{p^n}$ **then**
    $c \neq H(az_1 + z_2 - tc, \mu)$;
    return **reject**;
**else**
    return **success**;
**end if**
```

Where $\mathcal{R}^{p^n} = \mathbb{Z}_q[X]/(X^n + 1)$ and $\mathcal{R}_k^{p^n}$ defines the subset of the ring $\mathcal{R}^{p^n}$ where all polynomials have coefficients in the range $[-k, k]$. For more information, refer to [GLP12] and [AK18].

3 Smart Contract DPKIs ([PSRK17], [PSKR19])

In 2017, Patsonakis et al. proposed the first provably secure DPKI based off of public-state additive universal cryptographic accumulators like those in [LLX07] and smart contracts that run on the blockchain. This allows for the size of the smart contract state, which is the most expensive resource to access on the blockchain, to be minimized as well as the scheme to derive its security from the strong RSA assumption under the Random Oracle model.

3.1 Public-State, Additive, Universal Accumulators

The DPKI sketch introduced in [PSRK17] heavily utilizes a public-state additive universal cryptographic accumulators, of which the overall security is based off the strong RSA assumption under the Random Oracle model. Informally, public-state additive universal cryptographic accumulators means that no trap-door knowledge is required for one to perform any operation with the accumulator (public-state), it allows for the addition of elements (additive), and allows for both membership and non-membership witnesses (universal). Crucially, the authors' accumulators require the use of two trusted parties: T runs the key generation algorithm and publishes the accumulator's public-key while T_{acc} maintains the accumulator, which contrasts from a strong accumulator [CHKO12] which makes the key-generation algorithm publicly executable. A succinct version of the formal definition is included in the appendix and refer to the original paper for more details.

This scheme is identical to that proposed by Li et al. [LLX07] with the addition of a Map procedure which maps an arbitrary string (i.e. an identity) to a prime number which can then be used by the accumulator. A brief description of the Map procedure is provided in the appendix. For more details, refer to the original paper [PSRK17] as well [GHR99] which Map is heavily inspired by.

Finally, the authors proved the entire accumulator and Map was secure under the strong-RSA assumption and Random Oracle Model. I omit these proofs for brevity since the proofs themselves are not important for the overall security proof of the protocol. Merely the fact that the accumulators and Map are secure is sufficient for the security proof described in Section 3.3. For more details, see the original paper. Before discussing the authors' attempt to implement the DPKI in [PSKR19], I will briefly discuss the type of naming service functionality F_{ns} that the authors wished to provide with their DPKI and provide a discussion of the security proof itself.

3.2 Naming Service Smart-Contract Functionality

The authors define an ideal functionality F_{ns} which interacts with n clients, m servers, and trusted party T which is responsible for setup, and adversary S which is called the simulator as discussed in Section 2.2.1. This is the functionality they wish to achieve using their smart contract primitive.

Notably, the authors only allow for static corruptions (the simulator specifies the set of corrupted clients, C_{corr} before setup) in this model and requires the need for a trusted party T, which are both unrealistic/insecure assumptions to make in a practical setting. With smart contracts, the size of the state needs to be minimized since it is the most expensive resource to access. As a result, the authors separated the storage from verification by maintaining two public-state universal accumulators as discussed in Section 3.1. and Appendix B. These accumulators have constant-size and are based on RSA.

The smart contract works by running a program P: it initializes the two accumulators and stores the state of the service as the parameters (pk_1, pk_2, R) where pk_1, pk_2 are the public keys of the accumulators and R is the revocation relation. One accumulator stores (id, pk, i, op) where $\mathsf{op} = a$ marks an element as added and $\mathsf{op} = d$ marks an element as deleted. This allows for deletion to be recorded without needing trapdoor/private-key access to the accumulator. The other stores (id, i, op) which allows clients to infer if an identity is registered in the system. When a server asks for an identity registration, it checks the current state (i.e. the two accumulators) to see if the identity is currently in the state and if not, it adds the identity pair to the accumulator. A similar process is involved with revocation, except it also checks that the revocation relation R. As a result, the smart contract acts as T_{acc} by managing the state of the accumulators. For a more information, see the original paper.

This implementation requires an honest majority of servers (which is also an underlying requirement of most blockchains in general) and allows the adversary to have full knowledge of the computations performed and to be able to interfere through allowing/aborting an execution of P, but it cannot change the output. Servers act as "miners" and the trusted party T and clients interact through posting transactions, with the installation of program P being a special transaction. Similarly, executing P requires it to be run by all miners and recording its state update in the blockchain, resulting in the security of the scheme being derived from the security of the blockchain (the persistence of transactions). Although the trusted setup is assumed to be done by a trusted party, this step can be replaced with distributed protocols which remove the single point of failure.

The authors also introduce F_{UDB} which is an "unreliable database" which stores the identity pairs. It is "unreliable" as the adversary is allowed to tamper with its contents. A client queries F_{UDB} to find out the information necessary to interact with the smart contract (i.e. whether a pair has been registered or not) and following an interaction with the smart contract, the client stores the output in F_{UDB}. The authors then prove that the protocol informally described here (involving the smart contract, program P, and F_{UDB}) is secure using the Strong RSA Assumption in the Random Oracle Model. The proof is discussed in detail in the next subsection.

3.3 Security Proof of the Protocol

The main goal of the security proof is to show that the smart contract proposed by the authors is equivalent to the ideal functionality. The correctness of the model means that it only fails to provide the correct action (i.e. returning the current mapping, registering a new mapping, etc.) if the client is part of C_{corr} as defined by the adversary prior to setup and does not get tricked by the adversary by committing to a false registration or revocation request. For example, the model should not allow for a mapping to be revoked unless the revocation relationship R is met.

More formally, the authors prove the following: *The "hybrid" protocol using the smart contract and F_{UDB} is secure and realizes the ideal functionality F_{ns} under the Strong-RSA assumption in the Random Oracle Model.*

3.3.1 Proof Sketch

There are five main commands of importance: $\mathsf{Init}, \mathsf{Setup}, \mathsf{Register}, \mathsf{Revoke}$ and $\mathsf{Retrieve}$. For the sake of brevity, I will only consider the case with $\mathsf{Revoke}/\mathsf{Register}$. I separated the proof into each of the possible messages/commands that can be sent by the environment to show that the output of the parties is indistinguishable from the hybrid model and the ideal functionality.

The environment $\mathcal{Z}$ sends a $\mathsf{Revoke}/\mathsf{Register}$ **command to a client** C: There are essentially two subcases in this situation: either the adversary A changes the state of F_{UDB} prior to the command by $\mathcal{Z}$ or it doesn't. In the case that A does change the state, either it has an impact on the given command (i.e. changing it so that the identity you want to revoke doesn't exist) or it doesn't. In the case that it does have an impact, an honest client in both situations (ideal and hybrid) will fail when the state is incompatible with the command they want to do. With corrupted clients, the accuracy and security of the protocol is reduced into the security of the accumulators themselves. This is because the corrupted client would not be able to compute the right accumulated values necessary for the protocol to approve the forged request unless the accumulator itself was insecure. Since those cannot be manipulated freely by the adversary, the security of the scheme is dependent on the strong RSA assumption of the accumulators.

In the case that adversary does not change the state of F_{UDB} prior to $\mathcal{Z}$'s command, an honest client will either succeed (if they can) or fail if their command is incompatible with the state. In the case of corrupted clients, they can only convince the smart contract and F_{UDB} of false information if they can corrupt the accumulators (similar to the previous case), thus reducing the security to the strong RSA assumption. This is true regardless of whether I am discussing the hybrid model or the ideal functionality.

- $\mathcal{Z}$ **sends** (sid, Init) **to a server** S_i: The only difference between the hybrid and ideal functionality is that the server sends an initialization command to the smart contract and F_{UDB} so they can be added to the set and the adversary is notified in the hybrid model. In the ideal world, the simulator simulates the smart contract and F_{UDB} so after the ideal functionality F_{ns} send the initialization command, it simulates the initialization. In both situations, the server receives $\mathsf{success}$.

- $\mathcal{Z}$ **sends** (sid, Setup, R) **to the trusted party** T: In the hybrid model, the trusted party T sends an install request to the smart contract which (given the adversary $\mathcal{A}$'s approval) then runs the program P who initializes the keys for the two accumulators, initializes the state, and gives these parameters back to the smart contract which if the adversary approves again, sends back to trusted party T which then sends success to $\mathcal{Z}$. In the ideal functionality model, the simulator simulates T and the smart contract and when the adversary approves, the simulator $\mathcal{S}$ sends a command to F_{ns} which then returns success also. Thus, both in the ideal and hybrid model, $\mathcal{Z}$ is returned success.

- $\mathcal{Z}$ **sends a** Retrieve **command to a client** C**.**: This case parallels the prior cases. At most, a corrupted client would be able to get information about the state itself, but this does not influence the output of the functionalities in either model and as a result does not matter for our analysis. As a result, the security follows from the accumulator security.

For more information (such as a formal proof), see the original paper.

3.4 Improvements to the Original Design ([PSKR19])

In 2019, the authors revisited the Smart Contract PKI [PSKR19] and implemented the design in Ethereum, one of the largest blockchain systems that support smart contract capabilities. They found that the design was impracticable to implement for large-scale projects and instead created a Merkle hash-tree based smart contract system that could leverage the tree hashes of the blockchain to provide performance benefits.

In this subsection, I will discuss the performance issues of the implementation of the previous RSA-based design as well as the newer hash tree-based design that was found to be more efficient when implemented. Note that price estimates are for illustrative purposes only and are based off the price of Ethereum on 4/25/2022 at the specific time I was writing this paper and that the conversion of gwei to dollar amount may be inaccurate but serve as a relative cost per operation.

3.4.1 Performance issues of RSA-based ([PSRK17]) Smart Contract DPKI)

The authors implemented the RSA-based design by creating a testnet of Ethereum servers running the smart contract program. The bit lengths of the exponents, moduli, and exponentiation bases were set to 195, 3072, and 3072 bits long respectively.

They ran four sets of experiments: evaluating the Big Number library operations (primality tests, modulo multiplications, and exponentiation), evaluating the RSA accumulator (accumulations of 195-bit prime numbers, (non) membership witness verifications), the cost of mapping strings to 195-bit primes (Map), and the cost of registering/revoking (id,pk) pairs in the DPKI itself.

The experiments found that modulo exponentiation and primality testing were expensive operations that had a cascading impact on the overall cost of running the DPKI itself. In particular, cases where the exponent length was larger than the length of an EVM word (32 bytes) led to large access gas costs. Thus, operations like witness verifications (755,537.23 gwei for membership verification/1,519,279.96 gwei for non-membership, $2.26 per membershp operation/$4.45 per non-membership operation) and adding (810,153.32 gwei, roughly $2.43 per operation as of 4/25/2022) to the accumulator are extremely expensive since they require multiple modulo exponentations.

In short, the RSA-based design sacrifices computational overhead for the sake of keeping the smart contract state and witnesses constant size. However, this results in operations becoming extremely expensive when translated and executed by the blockchain. This is particularly true of the Map procedure which has a computational complextity of $O(k^2)$ where k is the length of the identifier. This procedure costs on average 2,141,780,036 gwei to execute once which is approximately $6,425 per operation! As a result, (de)registration operations on the DPKI are extremely expensive and unrealistic as a replacement for the current X.509 system.

The main reasons why Map is so expensive to run on Ethereum is because it is cheaper to access one EVM word (32 bytes) than one byte and at the time of the proposal, EVM bit operations were extremely expensive. Improvements to Ethereum's cost model to make accessing one byte cheaper as well as native bitwise support (which has been implemented in EIP-145 [BB17]) would make the RSA-based system cheaper to implement and execute. In the absence of those changes, the authors introduced a Merkle-tree hash based DPKI construction that attempts to mitigate the inefficiencies of the RSA accumulator.

3.4.2 Merkle-hash based DPKI

The primary difference between the newer construction and the older, inefficient constructor, is the universal accumulator. In the prior design, an accumulator based off the Strong RSA assumption was used which led to expensive modulo exponentiation and primality tests. As a result, the authors opted to use the collision-resistant hash tree based accumulators devised by Camacho et al. [CHKO12] The security of the protocol, rather than being based on the strong RSA assumption directly, is dependent on the provable security of the collision-resistant hash function used for the accumulators. In addition to not requiring the more expensive operations, the accumulator is also *strong* which means it does not require trusted setup like the RSA-based implementation and allows for public verification of additions/deletions through CheckUpdate which returns 1 on a witness honestly generated by Add or Delete and the accumulator values before and after the change. In general, the accumulator's underlying data structure is a balanced, binary tree, with the accumulator's value represented as the hash of the root node and witnesses are hash paths starting from a node leading up to the root node. This leads to constant size accumulator values, but witnesses have $O(\lambda \log(n))$ bit size where n is the number of accumulated elements. For more information, refer to the original paper by Camacho et al.

The authors ran two sets of experiments: the first experiment tested the cost of verifying (non) membership and CheckUpdate once the accumulator has 100,000 elements. The second experiment tested the gas cost of registering/revoking 100,000 (id, pk) pairs. As expected, since the size of each proof grew logarithmically as the accumulated values grew, the gas costs also grew with a similar trend. However, since witnesses are hash paths, sometimes the gas cost was more or less than "expected" depending on the individual element. In terms of the DPKI itself, revoking keys were the most expensive as they required verifying signatures, but even the most expensive revocation (2,999,214 gwei/$8.99) was considerably cheaper than the RSA-based DPKI. Gas costs could be cheaper if the authors used KECCAK-256 which is available as EVM opcode, but this is not NIST compliant.

Despite this, the authors found that this implementation could easily and securely be used on Ethereum's mainchain, particularly if developers of the project were willing to make changes (such as having more sophisticated precompiled contracts, support for NIST compliant hash-functions, etc.) that would allow for a DPKI to be faster and more efficient. However, one challenge that remains is witnesses being dependent on the number of id-key pairs registered, which can become verbose over time unless steps are taken to "reset" the accumulators from time-to-time.

4 Lattice-based blockchain DPKI ([AK18])

In 2018, Hyeongcheol An and Kwangjo Kim from the Korea Advanced Institute of Science and Technology (KAIST) proposed Qchain, a quantum-resistant blockchain-based DPKI. This expanded on prior work of using lattice-based cryptography on blockchains such as [BS16] while also attempting to fix the issues with X.509 that were discussed earlier in this paper.

4.1 How QChain Works

The public key encryption scheme supported by QChain is the scheme proposed by Lyubashevsky et al. [LPR10] which is secure against quantum computing attacks and in order to allow for concrete efficiency, the authors used Number Theoretic Transformation (NTT) operations [GOPS13]. Rather than store transactions in each block like cryptocurrencies or require a smart contract to manage the system, QChain is its own blockchain system where each block stores the id-pk pairs and uses their hashes for the Merkle hash tree structure.

4.1.1 Modified GLP Signature Scheme

The major modifications of the signature scheme used for QChain compared to the original GLP scheme (as discussed in Section 2.4) is the use of Number Theoretic Transformation (NTT), a form of Fast Fourier Transformation, to reduce the computation necessary for lattice-based operations.

Number Theoretic Transformation (NTT): For a polynomial $\mathbf{g}= \sum_{i=0}^{1023} g_i X_i \in \mathcal{R}_q$:

$$NTT(g) = \hat{g} = \sum_{i=0}^{1023} \hat{g}_i X_i \;\; where \;\; \hat{g}_i = \sum_{j=0}^{1023} \gamma^j g_j \sigma^{i,j}$$

where $\sigma = 49$, $\gamma = \sqrt{\sigma} = 7$. The inverse function NTT^{-1} is defined below:

$$NTT^{-1}(\hat{g}) = g = \sum_{i=0}^{1023} \hat{g}_i X_i \;\; where \;\; g_i = n^{-1}\gamma^{-i} \sum_{j=0}^{1023} \hat{g}_j \sigma_{ij}$$

where $n^{-1} \bmod q = 12277$, $\gamma^{-1} \bmod q = 8778$, $\sigma^{-1} \bmod q = 1254$. This simplifies the matrix multiplications that are originally done in GLP signatures. The scheme, otherwise, remains unchanged. For more information, refer to the original paper and Appendix E. [AK18]

4.1.2 QChain Algorithms

Below is an abridged list of the functions that manage QChain:

- $\mathsf{Qchain.Setup}(1^\lambda)$: Choose security parameter λ and output parameters $n, q, \sigma = \sqrt{16/2} \approx 2.828$ based off [BLP+13]
- $\mathsf{Qchain.KeyGen}(n, \sigma)$: Polynomial r_1 and r_2 are sampled from Gaussian distribution using NTT operation in polynomial multiplication and addition.

$$r_{1,i}, r_{2,i} \leftarrow \chi_\sigma$$

$$y_{1,i}, y_{2,i} \xleftarrow{R} \mathcal{R}_q^k$$

$$a_i \xleftarrow{R} \mathcal{R}_q;\ \ \hat{a}_i \leftarrow NTT(a_i)$$

$$\hat{r}_{1,i} \leftarrow NTT(r_{1,i});\ \ \hat{r}_{2,i} \leftarrow NTT(r_{2,i})$$

$$\hat{y}_{1,i} \leftarrow NTT(y_{1,i});\ \ \hat{y}_{2,i} \leftarrow NTT(y_{2,i})$$

$$\hat{p}_i \leftarrow \hat{r}_{1,i} - \hat{a}_i * \hat{r}_{2,i}$$

$$\hat{t}_i \leftarrow \hat{a}_i * \hat{r}_{1,i} + \hat{r}_{2,i}$$

where $\mathcal{R}_q$ is the polynomial ring $\mathbb{Z}_q[X]/(X^n+1)$ where polynomial f is irreducible of degree $n-1$. The public key is $(\hat{a}_i, \hat{p}_i, \hat{t}_i)$ and the secret key is $(\hat{r}_{1,i}, \hat{r}_{2,i}, \hat{y}_{1,i}, \hat{y}_{2,i})$ for user i based off [LPR10].

- QChain.User.Setup(pk_i, H_{root}) : This algorithm generates the information necessary to add to the blockchain for id-pk registration.

  Previous hash $\leftarrow H_{Block0}$
  $nonce \xleftarrow{R} \{0,1\}^n$
  $pk_i \leftarrow$ **User public key** $\in \{0,1\}^n$ where $0 \le i \le 2^{10}$
  $timestamp_i \leftarrow$ **current time**

- QChain.User.Add$(ID_i, Username_i, sk_i, pk_i)$: This algorithm adds the user information about their public keys onto a block in the blockchain. The algorithm works as follows:

$$H(ID_i),\ \ ID \leftarrow \text{User ID}$$

$$H(Username_i),\ \ Username_i \leftarrow \text{Username}$$

$$(\hat{r}_{1,i}, \hat{r}_{2,i}, \hat{y}_{1,i}, \hat{y}_{2,i}) \leftarrow sk_i$$

$$y_{1,i} \leftarrow NTT^{-1}(\hat{y}_{1,i});\ \ y_{2,i} \leftarrow NTT^{-1}(\hat{y}_{2,i})$$

$$c_i \leftarrow H(a_i, y_{1,i} + y_{2,i} ID_i);\ \ \hat{c}_i \leftarrow NTT(c)$$

$$r_{1,i} \leftarrow NTT^{-1}(\hat{r}_{1,i});\ \ r_{2,i} \leftarrow NTT^{-1}(\hat{r}_{2,i})$$

$$Sign(ID_i, a_i, r_{1,i}, r_{2,i})$$

The output signature value is $(z_{1,i}, z_{2,i}, \hat{c}_i)$ which is based on the modified-GLP scheme discussed in Appendix E. They define **pkInfo** to be the information used for the Merkle hash tree:

$$\textbf{pkInfo} = pk_i || H(ID_i) || H(Username_i) || timestamp_i || Sign(ID_i, a_i, r_{1,i}, r_{2,i})$$

Then they construct the Merkle hash tree using **pkInfo** as follows:

$$H_{\frac{i-1}{2},\ldots,j} = \begin{cases} H_{\frac{i-1}{2},\ldots,0} = H(\textbf{pkInfo}) & \text{if } i = odd \\ H_{\frac{i-1}{2},\ldots,1} = H(\textbf{pkInfo}) & \text{if } i = even \end{cases}$$

And then the top hash value H_{root} is calculated using the hash value of the leaf nodes. The maximum depth of the Merkle hash tree (and therefore the number of users per block) is $2^1 0$ in order to restrict complexity growth.

- $\mathsf{QChain.User.Verify}(ID_i, pk_i, Sign(ID_i))$: To verify the public key pk_i and $Sign(ID_i)$ of the user, the verification algorithm works as follows:

$$\hat{a}_i, \hat{t}_i \leftarrow pk_i$$

$$a_i \leftarrow NTT^{-1}(\hat{a}_i);\;\; t_i \leftarrow NTT^{-1}(\hat{t}_i)$$

$$z_{1,i}, z_{2,i}, \hat{c}_i \leftarrow Sign(ID_i)$$

$$c_i \leftarrow NTT^{-1}(\hat{c}_i)$$

$$Verify(ID_i, z_{1,i}, z_{2,i}, c_i, a_i, t_i)$$

  and pk_i and $Sign(ID_i)$ are public parameters. The verification algorithm is based on the modified-GLP scheme (but is almost identical to the one in Section 2.4) described in more detail in the Appendix E.

- $\mathsf{QChain.Enc}(pk_i, m)$: To encrypt message $m \in R_2$, the encryption algorithm works as follows:

$$(\hat{a}_i, \hat{p}_i, \hat{t}_i) \leftarrow pk_i;$$

$$(a_i, p_i, t_i) \leftarrow (NTT^{-1}(\hat{a}_i), NTT^{-1}(\hat{p}_i), NTT^{-1}(\hat{t}_i));$$

$$e_1, e_2, e_3 \leftarrow \chi_\sigma;$$

$$\hat{e}_1 \leftarrow NTT(e_1);\;\; \hat{e}_2 \leftarrow NTT(e_2);$$

$$\hat{m} \leftarrow m * \left\lfloor \frac{q}{2} \right\rfloor;$$

$$(\hat{c}_1, \hat{c}_2) \leftarrow (\hat{a}_i * \hat{e}_1 + \hat{e}_2, \hat{p}_i * \hat{e}_1 + NTT(e_3 + \hat{m}));$$

  And the ciphertext is $c = (\hat{c_1}, \hat{c_2})$.

- $\mathsf{QChain.Dec}(sk_i, c)$: To decrypt a ciphertext $c = (\hat{c}_1, \hat{c}_2)$, the algorithm do the following:

$$\hat{r}_{2,i} \leftarrow sk_i;$$

$$(\hat{c}_1, \hat{c}_2) \leftarrow c$$

$$m; \leftarrow NTT^{-1}(\hat{c}_1 * \hat{r}_2 + \hat{c}_2);$$

$$m \leftarrow \mathsf{Decode}(m');$$

  Where $\mathsf{Decode}()$ is an error reconciliation function that simply rounds off the error of the message:

$$\mathsf{Decode}(m') := \left\lfloor \frac{2}{q} * m * \lfloor q/2 \rfloor \right\rceil * \left\lfloor \frac{q}{2} \right\rfloor$$

An example exchange between QChain operators is the following: the first operator initiates the genesis block (which is the first block, a process that is similar to other blocks except a string of all 0s is used for the previous hash and a random number for pk_i) as well as setting up the parameters for the overall chain. Then, QChain creates another block and users run QChain.User.Setup() and QChain.User.Add() algorithms in order to register a public key. These keys can be verified using QChain.User.Verify() which allows users to determine if a key they are provided is authenticated or not.

The security of this protocol is based on the security of the hash scheme (which the authors stated could be SHA-3 which has some resilience to quantum attack) and the modified GLP scheme based on the ring-LWE problem. However, notably, there is no security proof for the protocol provided by the authors unlike the smart contract model.

In spite of this, QChain provides many features similar to X.509 v3 while also being quantum resistant as I will discuss in the next section.

4.2 How it Compares with X.509 v3

A primary advantage that QChain has to X.509 v3 is that it can keep offline states (except for the initiation of the genesis block) while X.509 v3 must keep online states (particularly by a trusted third party like CA) in order for users to verify public keys. QChain also provides non-repudiation since each block consists of a user's public keys and their signatures, so they cannot deny them, particularly with the immutable nature of the blockchain.

This bears the question about how users can revoke their registrations like they could in Patsonakis et al.'s model [PSRK17]. Although the authors stated that revocation is $O(\log(n))$, the same as the complexity of QChain.User.Add(), I am not fully sure how this scheme supports revocation natively. The authors mentioned that timestamps can be used to allow id-pk pairs to "expire" but this requires additional communications between operators or have a blockchain for a Certificate Revocation List (CRL) but these options are not expanded upon. Perhaps if the verification algorithm simply looks for the most recent association, this could mitigate some concerns, but this requires users to update their associations as soon as possible to risk forgery attacks. In my opinion, this is the biggest flaw in the construction: without an algorithm that allows for native revocation, it's possible for an adversary to simply use a prior association to claim they are someone else and the verification algorithm will validate that association.

QChain's decentralized model however does prevent single points of failure, which is an inherent problem with X.509. In particular, as long as a majority of operators remain honest, it is impossible for malicious blocks to be approved and therefore malicious associations to registered. However, because of the lack of revocation, much remains necessary before this can be utilized in a real-world system, unlike the previous smart-contract based model.

5 Conclusions and Future Work

In this paper, I discussed two major DPKI systems: an Ethereum smart-contract based approach by Patsonakis et al. ([PSRK17], [PSKR19]) and a GLP-signature based approach known as QChain by An and Kim. While [PSRK17] provides a security proof by deriving the security of the system to the security of the cryptographic accumulators (which in turn is based on the strong RSA assumption), this system is not quantum resilient as the strong RSA assumption can be broken through quantum algorithms. Similarly, given Ethereum's history of hard forks [eth22] as a method of "upgrading" the platform, it seems unclear how the DPKI proposed in [PSRK17] would adapt (i.e. would all associations, including those revoked, have to be reintroduced back inito the new blockchain state, etc.).

On the other hand, QChain itself seems little more than a proof of concept for a GLP-signature based blockchain that is purpose-built for DPKIs. In general, it does not seem clear how the immutable property of QChain allows for users to revoke associations and since there is no security model for the blockchain itself, there is a possibility for non-quantum based adversaries to trick the system into accepting malicious registrations or for an adversary to deceive users by using a non-current association of a public key. However, in general, it seems that the DPKI systems highlighted in this paper, although suffering from faults on their own, seem to solve the trusted third party and single points of failure issues that X.509 currently has and are more scalable as a result.

These approaches are just a stepping stone given that Ethereum, the most likely platform for a complex DPKI system, was only founded in 2015 and is constantly improving. As a result, I have proposed some open questions below to help spur the development of decentralized public key infrastructures:

5.1 Open Questions

- **Does the immutable property of blockchains make it impossible to create a purpose-built blockchain for systems, like DPKIs, where revocability is aa integral part?** As seen in [PSRK17] and by the muddled nature of QChain, it seems hard to create blockchains solely for DPKIs and instead researchers have to rely on smart contracts that run primitives like cryptographic accumulators which allow for revocation properties. However, future work could investigate this further to see if there are optimizations that can be made so that revocations run "on chain" rather than through a smart contract medium.

- **How will hard forks and splits in blockchains affect DPKIs?** In 2016, Ethereum Classic was split from Ethereum as a result of a vote to delete the theft of $50 million worth of Ethereum from DAO, a decentralized venture capital fund. [Lei17] This resulted in Ethereum Classic retaining the blockchain of Ethereum up to that point, including the theft, whereas Ethereum itself essentially erased the theft. If a DPKI system was implemented prior to a situation like this, it would result in duplicated DPKIs which could be used maliciously if the community for one of the new blockchains didn't care to maintain it. In general, a split situation like this leads to greater consideration about how to maintain authentication systems built on decentralized systems which are constantly being developed. Future work could explore these situations, especially for a system as important as public key infrastructure.

6 Acknowledgements

This paper would not have been possible without Ansh Shah, who I discussed cryptocurrencies with in great detail during my self-exile adventures in Oklahoma City and San Antonio during the start of the pandemic which inspired this paper, as well my friends and family.

References

[AK18] Hyeongcheol An and Kwangjo Kim. Qchain: Quantum-resistant and decentralized pki using blockchain. In *2018 Symposium on Cryptography and Information Security (SCIS 2018)*. IEICE Technical Committee on Information Security, 2018.

[BB17] Alex Beregszaszi and Paweł Bylica. Eip-145: Bitwise shifting instructions in evm, Feb 2017.

[BLP$^+$13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 575–584, 2013.

[BS16] Rachid El Bansarkhani and Jan Sturm. An efficient lattice-based multisignature scheme with applications to bitcoins. In *International Conference on Cryptology and Network Security*, pages 140–155. Springer, 2016.

[BSH93] Dave Bayer, W. Scott Stornetta, and Stuart Haber. Improving the efficiency and reliability of digital time-stamping. In *Sequences II: Methods in Communication, Security and Computer Science*, pages 329–334. Springer-Verlag, 1993.

[But13] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[CHKO12] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. *International Journal of Information Security*, 11(5):349–363, 2012.

[CMV21] Luigino Camastra, Igor Morgenstern, and Jan Vojtěšek. Backdoored client from mongolian ca monpass, July 2021.

[eth22] History and forks of ethereum, Apr 2022.

[GGF17] Paul A. Grassi, Michael E. Garcia, and James L. Fenton. Digital identity guidelines. Technical Report NIST Special Publication (SP) 800-63-3, Includes updates as of March 2, 2020, National Institute of Standards and Technology, Gaithersburg, MD, 2017.

[GHR99] Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 123–139. Springer, 1999.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 530–547. Springer, 2012.

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In *International Workshop on Post-Quantum Cryptography*, pages 67–82. Springer, 2013.

[KHPC01] D. Richard Kuhn, Vincent C. Hu, W. Timothy Polk, and Shu-Jen Chang. Technical Report NIST Special Publication (SP) 800-32, National Institute of Standards and Technology, Gaithersburg, MD, 2001.

[Lei17] Matthew Leising. The ether theif, Jun 2017.

[LLX07] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In *International Conference on Applied Cryptography and Network Security*, pages 253–269. Springer, 2007.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 1–23. Springer, 2010.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 35–54. Springer, 2013.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[P$^+$16] Chris Peikert et al. A decade of lattice cryptography. *Foundations and Trends® in Theoretical Computer Science*, 10(4):283–424, 2016.

[Pri11] J. R. Prins. Sep 2011.

[PSKR19] Christos Patsonakis, Katerina Samari, Aggelos Kiayiasy, and Mema Roussopoulos. On the practicality of a smart contract pki. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 109–118. IEEE, 2019.

[PSRK17] Christos Patsonakis, Katerina Samari, Mema Roussopoulos, and Aggelos Kiayias. Towards a smart contract-based, decentralized, public-key infrastructure. In *International Conference on Cryptology and Network Security*, pages 299–321. Springer, 2017.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[Sta21] ETH Gas Station. All you need to know about eip-1559, Aug 2021.

A Standard Definitions

Notation: Unless otherwise stated, λ denotes the security parameter and $\mathsf{negl}(\cdot)$ refers to a function that is negligible by some parameter.

Definition A.1 (Strong RSA Assumption from [PSRK17]). *For any p.p.t. adversary $\mathcal{A}$,*

$$\Pr[n \leftarrow \mathsf{KeyGen}(1^\lambda); x \leftarrow \mathbb{Z}_n^*; (y, e) \leftarrow \mathcal{A}(n, x) : y^e = x \bmod n] = \mathsf{negl}(\lambda)$$

where $n = pq$ and p, q are safe primes.

Definition A.2 (2-Universal Hash Function Family from [PSRK17]). *Let $U = \{f | f : X \to Y\}$ be a family of functions. U is a 2-Universal Hash Function Family if, for all $x_1, x_2 \in X$ with $x_1 \neq x_2$ and for all $y_1, y_2 \in Y$,* $\Pr_{f \in U}[f(x_1) = y_1 \wedge f(x_2) = y_2] = (\frac{1}{|Y|})^2$.

Definition A.3 (Pseudorandom Generator). *Let $G : \{0,1\}^k \to \{0,1\}^{p(k)}$ be a deterministic polynomial time algorithm and $p(\cdot)$ be a polynomial in some parameter k. G is a pseudorandom generator if, for any p.p.t algorithm $\mathcal{D}$*

$$|\Pr[\mathcal{D}(r) = 1] - \Pr[\mathcal{D}(G(s)) = 1]| \leq \mathsf{negl}(\mathsf{k})$$

where r is a random string chosen uniformly at random from $\{0,1\}^{p(k)}$ and s, the seed, being chosen uniformly at random from $\{0,1\}^k$.

Definition A.4 (Collision-resistant Hash Function Family) *A family of functions $h_k : \{0,1\}^{m(k)} \to \{0,1\}^{l(k)}$ generated by some algorithm G is a family of collision resistant hash functions if $|m(k)| > |l(k)|$ for any k (compressing the input string), and every h_k can be computed within polynomial time given k but for any p.p.t algorithm A:*

$$Pr[k \leftarrow G(1^n), (x_1, x_2) \leftarrow A(k, 1^n) \;\; s.t. \;\; x_1 \neq x_2 \text{ but } h_k(x_1) = h_k(x_2)] < \mathsf{negl}(n)$$

B Lattice-based Problems

B.1 Summary of LWE problem from [P+16]

LWE is defined by positive integers n and q, and an error distribution χ over $\mathbb{Z}$. χ is usually considered a discrete Gaussian of width αq where $\alpha < 1$.

For a vector $s \in \mathbb{Z}_q^n$, known as the LWE secret, the LWE distribution $A_{s,\chi}$ over $\mathbb{Z}_q^n \times Z_q$ is sampled by choosing an $a \in \mathbb{Z}_q^n$ uniformally at random, choosing $e \leftarrow \chi$ and outputting $(a, b = \langle s, a\rangle + e \bmod q)$.

For cryptography, the **Decision-LWE**$_{n,q,\chi,m}$ is commonly used. The problem is defined as follows:

Decision-LWE$_{n,q,\chi,m}$- Given m independent samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ where every sample is distributed according to one of the following: (1) $A_{s,\chi}$ for a uniformly random $s \in \mathbb{Z}_q^n$ (fixed for all samples) or (2) the uniform distribution, and the goal is to distinguish which is the case with non-negligible advantage.

For more information, refer to the original paper by Regev [Reg09] or the survey conducted by Peikert et al. [P+16]

B.2 Summary of Ring-LWE problem from [P+16]

Ring-LWE is defined by a ring R of degree n over $\mathbb{Z}$, a positive integer modulus q defining the quotient ring $R_q = R/qR$, and an error distribution χ over R. Typically, R is a *cyclotomic* ring and χ is a discretized Gaussian in the canonical embedding of R.

For an $s \in R_q$, known as the Ring-LWE secrete, the ring-LWE distribution $A_{s,\chi}$ over $R_q \times R_q$ is sampled by choosing $a \in R_q$ uniformally at random, choosing $e \leftarrow \chi$, and outputting $(a, b = s \cdot a + e \bmod q)$.

The **Decision-R-LWE**$_{q,\chi,m}$ problem is described below:

Decision-R-LWE$_{q,\chi,m}$- Given m independent samples $(a_i, b_i) \in R_q \times R_q$ where every sample is distributed according to one of the following: (1) $A_{s,\chi}$ for a uniformly random $s \in R_q$ (fixed for all samples) or (2) the uniform distribution and the goal is to distinguish which is the case with non-negligible advantage.

For more infromation, see the original papers by Lyubaskevsky et al. [LPR10] and [LPR13] as well as the survey done by Peikert et al. [P+16]

C Definition of Public-state, Universal Additive Cryptographic Accumulators from [PSRK17]

Definition 1. (Definition 4.1 from [PSRK17]) Public-state, additive, universal accumulators. Let D be the domain of the accumulator's elements and X, the current accumulator set. The accumulator consists of the following algorithms (which aside from KeyGen, require only the public key to use):

- **KeyGen**: generates the public-private key pair (pk, sk) and outputs the public key pk. **Run by trusted party** T.
- **InitAcc**: outputs the initial accumulator value c_0. **Run by trusted party** T_{acc}.
- **Add**: Given x to be added to the accumulator set and the accumulator value c, it outputs the new accumulator value c' and the membership witness for x.
- **MemWitGen**: outputs a membership witness for x given the accumulator value c.
- **NonMemWitGen**: outputs a non-membership witness for x given the accumulator value c.
- **UpdMemWit**: After **Add**$(\cdot)$ is run for an element y, this will provide an updated membership witness for a pre-existing element x.
- **UpdMemNonWit**: Similar to UpdMemWit but with non-membership witnesses.
- **VerifyMem**: Verifies whether a given membership witness W for a given element x is honestly produced.
- **VerifyNonMem**: Verifies whether a given non-membership witness W for a given element x is honestly produced.

The accumulator is considered correct **VerifyMem** and **VerifyNonMem** only return 1 if given an honestly produced witness and 0 otherwise. The accumulator is also considered secure if no PPT adversary can forge a valid membership/non-membership witness.

D Map Procedure from [PSRK17]

Unlike [GHR99], the authors employ a deterministic mapping algorithm. They use a pseudorandom generator $G : \{0,1\}^k \to \{0,1\}^{p(k)}$ where $p(k)$ is polynomial in k and a labeled collision-resistant hash function $h : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^k$ which is modeled as a Random Oracle.

The procedure works as follows: first pick two labels $l_1 0, l_1$ from $\{0,1\}^*$ which remain the same for every run of the algorithm. Then given input $z \in \{0,1\}^*$, first compute $h(l_0, z)$ and then $G(h(l_1, z))$. Then, it samples elements from the set $\{x \in \{0,1\}^{3k} : f(x) = h(l_0, z)\}$ using randomness $G(h(l_1, z))$ and stops when a prime number is found. The authors proved that Map outputs a prime number, except with negligible probability, and assuming the existence of psuedorandom generators and random oracles. Similarly, they proved that Map is collision-resistant as long as h was collision-resistant.

E Modified-GLP Signature Scheme from [AK18]

Below is the signature scheme used by [AK18] for QChain. Note that this is essentially the GLP signature scheme ([GLP12]) but uses Number Theoretic Transformations (NTT) to speed up calculations. For more information on NTT, refer to the section on QChain, the original QChain paper ([AK18]), and the original paper on NTT ([GOPS13]).

Algorithm 2 Modified-GLP Signature Scheme from [AK18]

```
**Signing Key:** $r_1, r_2 \xleftarrow{R} \chi_\sigma$.
**Verification Key:** $a \xleftarrow{R} \mathcal{R}_q,\ \hat{a} \leftarrow NTT(a),\ \hat{r}_1 \leftarrow NTT(r_1),\ \hat{r}_2 \leftarrow NTT(r_2),\ \hat{t} \leftarrow \hat{a}\hat{r}_1 + \hat{r}_2$
**Hash function:** $H : \{0,1\}^* \rightarrow D_{32}^n$
Sign$(\mu, a, r_1, r_2)$ :
    1. $y_1, y_2 \xleftarrow{R} \mathcal{R}_q^k$;
    2. $c \leftarrow H(ay_1 + y_2, \mu)$;
    3. $\hat{c} \leftarrow NTT(c)$;
    3. $\hat{z}_1 \leftarrow \hat{r}_1 * \hat{c} + NTT(y_1)$;
    4. $\hat{z}_2 \leftarrow \hat{r}_2 * \hat{c} + NTT(y_2)$;
    5. $z_1 \leftarrow NTT^{-1}(\hat{z}_1)$;
    6. $z_2 \leftarrow NTT^{-1}(\hat{z}_2)$;
**if** $z_1 \notin \mathcal{R}_q^{k-32}$ or $z_2 \notin \mathcal{R}_q^{k-32}$ **then**
    Return to step 1;
**else**
    return $(z_1, z_2, c)$;
**end if**

Verify$(\mu, z_1, z_2, c, a, t)$ :
**if** $z_1, z_2 \in \mathcal{R}_q^{k-32}$ **then**
    $c \neq H(az_1 + z_2 - tc, \mu)$;
    return **reject**;
**else**
    return **success**;
**end if**
```

For more information, see the QChain paper ([AK18]).