Hash based Signature schemes

Implementation and application in blockchained applications

Harald Heckmann

March 27, 2019

Contents

1 Introduction

This project focuses on the application of hash based post-quantum secure on time signature schemes. The usage of such signatures schemes in blockchain based applications is the main goal. First, relevant one-time-signature (OTS) schemes will be regarded. This includes Lamport-Diffie OTS (LD-OTS) [Lam79], Winternitz OTS (W-OTS) [Buc+11][Hül17], which is an extension of LOTS, and eXtended Merkle Signature Scheme (XMSS) [JH11], which is a method to convert OTS Schemes to few-time Signature schemes. This Signature schemes serve as a basis for Blockchained Post-Quantum Signatures (BPQS) [Cha+18]. BPQS configures XMSS parameters in a way, that takes advantage of the blockchain to reduce execution costs of OTS schemes in regards to time and space.
The focus changed during the work on this project. The main goal was first to create a Proof-Of-Concept (POC) implementation of BPQS, but after realizing that no public WOTS python package exists, the goal shifted a bit. The creation of a open source winternitz library in a project, which was setup with the intention that a large group of people can potentially collaborate on this project (further detail on this in section 3.1), became the center of attention. Using this library in a BPQS POC implementation became the follow-up project. Section 2 delivers the fundamental knowledge to understand this project. Section 3 describes the design of the Winternitz and BPQS implementations. Section 4 describes the implementation itself. In section 5 the results and an outlook are presented.

2 Fundamentals

This section covers all fundamentals required to understand the design decisions and implementations. First, a brief overview of cryptocurrencies in aspect of the data structures and transactions and groups of transactions (which are finally linked) is delivered. This is required to understand BPQS. This section is followed by the description of all for this project relevant OTS schemes. Finally, the idea behind BPQS is explained.

2.1 Cryptocurrencies

Since late 2008 cryptocurrencies exist. They allow a decentral network to achieve consensus about the next state. This results in an ordered execution of transactions. The following subsection describes what a transaction exactly is and how they are stored after a consensus about their order has been determined.

2.1.1 Transactions, blocks and the blockchain

The transaction is simply a document, which contains the instructions to transfer value from one party to another. To proof the authenticity of a transaction, digital signature schemes are used. Currently public-key cryptography on elliptic curves is usually the primary choice, but it is known that quantum computers will break those algorithms in the near future. This is where post-quantum cryptography comes in.
All transactions are stored in a block. That block is propagated to the network, where each participant updates his local state according to the transactions contained within that block. Each recipient of the block does not simply accept it believing that all contents are valid. Instead, he or she does check if the transactions within and the block itself is valid, which implies that every participant checks every signature in every transaction ever propagated in a block. All those blocks are linked together as a hash-chain, where each $n+1$th block does reference the nth block. The blockchain does specify the order of blocks. To protect the blockchain from being altered, the result of a difficult computation or a solution for a hard problem, which can not be solved with computational power, is used as an requirement for each block. The blockchain is stored on a persistent storage. Figure 1 illustrates the blockchain simplified.

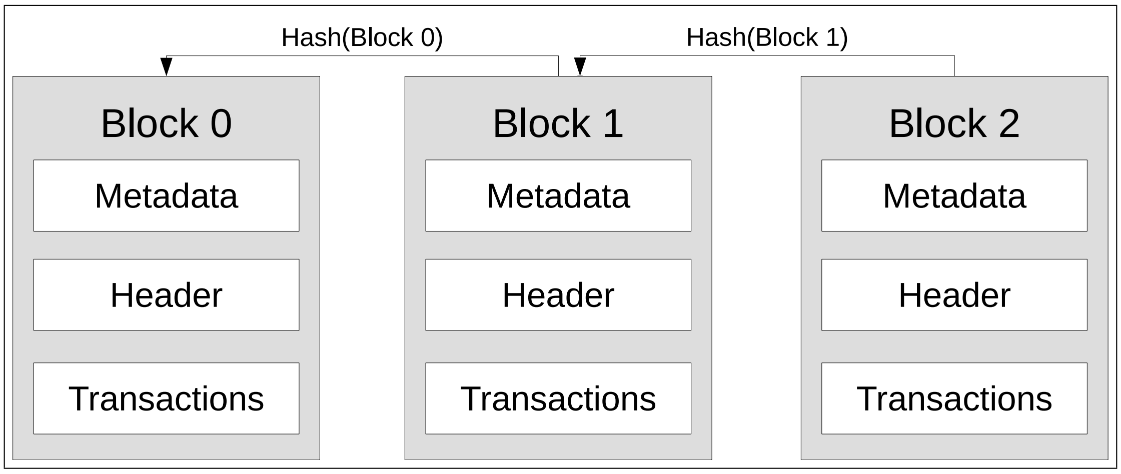

Figure 1: Simplified illustration of a blockchain

2.1.2 Implicit and explicit accounts

Cryptocurrencies like bitcoin don't store accounts explicitly. Instead, one must search all the balances bound to a public key that he or she can generate a signature for. In contracst, cryptocurrencies like ethereum use a state database, where each account is stored explicitly with a mapping between the public address (used to receive funds) and the data associated to it (e.g. the balance).

2.2 Lamport-Diffie OTS

In the year 1979 Leslie Lamport and Whitfield Diffie developed a OTS scheme which is based on cryptographically secure hash functions. The security assumptions are based on the properties of the cryptographically secure hash function.
The basic idea is the following: For each bit in the message you create two hash values. All the hash values form the verification key, which has to published before signing a message. For each bit in the message the inverse image of one of the hash values, depending on the value of the bit, is published. All those inverse images form the signature. An attacker is not able to modify one bit, because he or she does not know the inverse image of the hash that serves as the verification key for the value of that specific bit.
Since the verifier has to hash each part of the signature once, which consists of as many values as the message has bits, this hash algorithm is quite efficient in regards to time-complexity. On the other hand, for each bit the signature contains value, which is inefficient in regards to space-complexity. For example, having a fingerprint of a message which has 256 bit, and further using a hash function with a digest size of 512 bit, results in a total signature size of $256 \cdot 512 = 131,072$ bit ≈ 16.4 kB, assuming that the size of the input for the hash function is 512 bit as well. The verification key is twice as big, a fingerprint of it can be published instead of the full key though.

2.3 Winternitz OTS

After the publication of the paper describing the LD-OTS algorithm, another algorithm called the "Winteritz OTS" algorithm was discovered. The main feature is that the user of this algorithm can adjust the time- to space-complexity tradeoff. This is achieved by grouping bits (or more generally, groups of one digit base-w numbers) together, where w is the Winternitz parameter. The number of keys is determined by the number of one digit base-w numbers a message can have. The bigger the grouping, the bigger the reduction of the space-complexity. This is due to the fact, that longer parts of the message are hashed together, which ultimately reduces the total number of hashes required for the verification key and the signature. On the other hand, a bigger Winternitz parameter also increases time-complexity. The signature is created by hashing the corresponding secret key m times, where m is the value of the part of the message which is to be signed. By doing so, each base-w part of the message can have its own signature. As you can see, the bigger the parts of the message that are signed separately, the higher the time effort to do so. The verification key is created by hashing each secret key w-1 times. Each hash that is produced from a secret key to a verification key is called "hash chain", if regarded together and in the correct order. The verification key can be derived from the signature by hashing each part of the signature $w - m - 1$ times. Additionally, an (inverse sum) checksum is appended to the message before signing it, which prevents man-in-the-middle attacks.

2.3.1 Winternitz OTS+ extension

The Winternitz OTS+ extension allows to calculate the security level of a given parameter set, does decrease the signature size significantly without lowering the security of the scheme and reduces the requirements of the hash function to be only second-preimage resistant. This achieved by introducing different bit masks, which are XORed to the input of the hash function. A pseudo random function and a seed can be used to produce such bitmasks.

2.4 XMSS

XMSS opens the possibility to group OTS signatures and to use a different key of this set for each signature. It does so by inserting the OTS verification keys (or just a fingerprint of them) into one merkle tree, which allows to sign as many times as unused OTS keys are available. The merkle root represents a verification key. The validity of a signature can be proven by deriving the verification key (or a fingerprint of it) from the signature and the message for one OTS and finally, given a verification path, by executing the merkle proof.

2.5 BPQS

Remember that every transaction is included in blocks and every of those blocks is included in the blockchain. This means that every copy of the blockchain also contains all digital signatures provided to proof the authenticity of a transaction. The bigger the signatures, the shorter the time until the physical medium storing the blockchain is out of capacity. This is especially a problem, because the blockchain is duplicated and spread to every participant in the network willing to deliver the full service to the network. It is wanted that everybody can participate without having vast physical storage available. So the main goal of a digital signature scheme is, of course besides providing the functionality it should, to produce a minimal signature size. The next goal is to reduce the verification times, since every participant verifies each signature in each transaction in each block. If this takes to long, the transaction output such a network can achieve is reduced. What is not that important is, how long it takes to generate an OTS key pair. Surely it should not take seconds, but in contrast to the signature storage and the verification, this procedure is executed independently from the network and only on one machine. BPQS uses these observation to design a signature scheme, which is optimized for the usage in such networks.
There is another eventuality that has to be considered by BPQS. It is in fact possible, that transactions can be rejected. This is for example the case when an invalid value in regards to the available balance is ordered to be transferred.

For such cases, OTS schemes alone are not sufficient. BPQS does also address this. The signature size problem is solved by introducing a XMSS tree, which does only grow in height (figure 2). It is relatively rare that a transaction gets rejected, so a few-time signature scheme containing enough signatures to reduce the probability that all keys are required due to rejected transactions, should be sufficient. This can also be worked around by adding one branch to the tree directly under the root, which contains the verification key of a many-time signature scheme (figure 3). Many-time signature schemes are expansive, so that is just an emergency backup. The XMSS tree does grow in height, because the hashes required for the proof are minimized for the first (couple of) signature(s). Imagine a 32 leaf nodes XMSS tree.

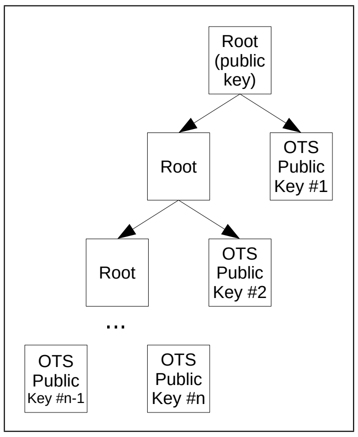

Figure 2: BPQS-few

If it is balanced, you always have to publish $log_2(32) = 5$ additional hashes to be able to proof the integrity of the data in question. If it grows in height, you always have to publish h additional hashes, where h is the height (starting from 1). Letting the tree grow in height in combination with the blockchain has another potential advantage: If you're willing to use multiple keys of your tree, you can just point to a block where a part of the verification of the hash tree has already been done. By doing so, you don't have to publish the verification path again and you don't have to verify that path, since it was already verified in the past. This feature further reduces space and time usage. Some blockchain designs even require this type of verification, if a user should be able to use the same address for as long as he or she wants.

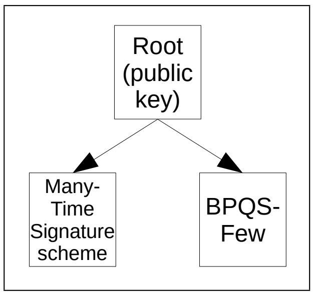

Figure 3: BPQS-GEN-B

3 Design

This section covers the design and the explanation for this design choices.

3.1 Winternitz python package

The requirements for this project are:

1. Possibility for collaborative work with many people
2. Well readable code
3. Extendable

4. Support debugging
5. Completely tested
6. Completely documented

Speed is secondary, since python does not fulfill this requirement by design. Nevertheless, it was still a concern to implement these algorithms as efficient as possible without impacting the other requirements negatively. The first requirement was fulfilled by setting up a project on github and an integration pipeline using travis. The github project contains a git-prehooks file, which commands be activated to check git commits for certain requirements. The integration pipeline run every time changed are pushed to the repository. It automatically invokes tox with different parameters. Tox is then executed with python 3.4 and python 3.7 to run the tests, the linter and an import order and format check, and finally build the documentation. If procedure succeeds, the commit is marked as accepted in github and can be included into stable branches.
To maintain readability, the code is following a strict style, which enforces different notations for variables, functions and classes, self-explanatory names, enough space so that the code does not appear compressed. Confusing parts are commented and in general, the code is kept quite simple. Every function and class is explained in detail in docstring comments.
To be extendable, the project contains a simple package structure:

- winternitz (package)
  - signatures (module)
    - AbstractOTS (abstract class)
    - WOTS (W-OTS, inherits AbstractOTS)
    - WOTS+ (W-OTS+, inherits WOTS)

If anybody wants to add another Winternitz signature scheme, they have to inherit from AbstractOTS, which forces them to implement sign(...), verify(...), getPubkeyFromSignature(...) and equality check functions. If the extension is even similiar to WOTS, they only have to implement the chaining function _chain(...), override sign(...) to return additional values required for the verification, override the equality checks, object representation function and string representation function.
The code does support debugging in three ways. First, many cases where invalid parameters are used in functions are detected and result in an detailed exception. Second, the internal state of an object can be printed in an human-readable format. Third, the repr(...) function can be used on the objects, which returns a line of code that, when executed, produces an equal object to that on which repr(...) was executed on. This code can also be automatically interpreted by using the eval(...) function.
The code is completely tested. Every possible usage (I could think of) of the library and every line of code is tested (the code coverage matches 100%).
The code is completely documented. Some parts like an introduction or usage examples

are written in separate files using reStructuredText. Apart from that, a whole api specification is generated from docstring which describes the classes and functions completely and elaborately.

3.2 POC of BPQS

The idea is that transactions are created randomly and propagated in a network (of processes), whereupon a transaction can be invalid and is therefore rejected. In this case, the sender of the transaction has to use a fallback key.
The POC implementation simulates a cryptocurrency, that uses explicit accounts stored in a state database, just like in Ethereum [Woo14]. The storage database contains a mapping of an address to data associated with that address. The storage database is depicted in figure 4.

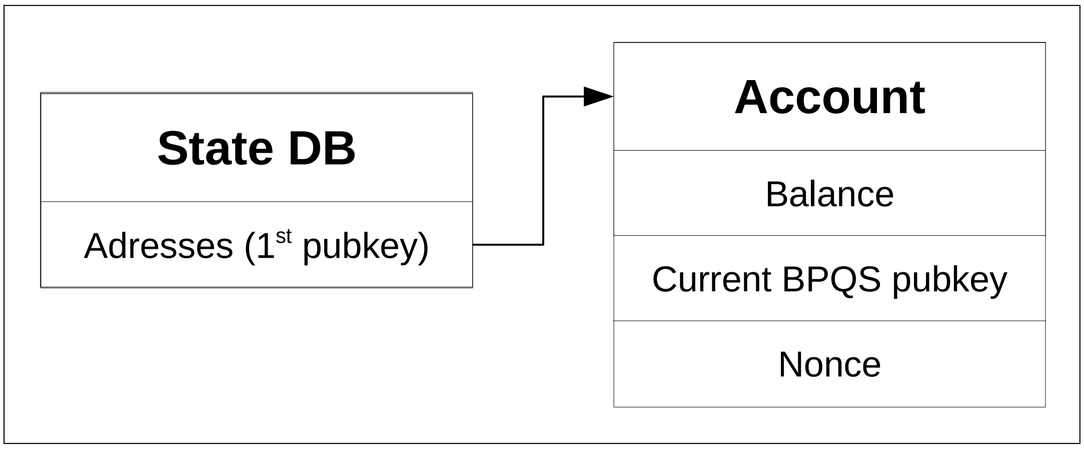

Figure 4: Storage database

The nonce is a protection against replay attacks on transactions, the public key contains the public key of the current BPQS tree. It has to be updated when a transaction succeeds. Therefore, the transaction must include the next public key. Further, it contains a verifcation path in addition to the signature to execute the merkle proof. The transaction is depicted in figure 5. The next public key and the key management in general, are handled by a wallet library. This library is able to generate the BPQS-FEW trees, return a public key and build and sign a transaction with the current key. Optimally, it should be able to derive the secret keys that are required required for the OTS schemes used in the BPQS tree, for example by using the nonce which is stored in the

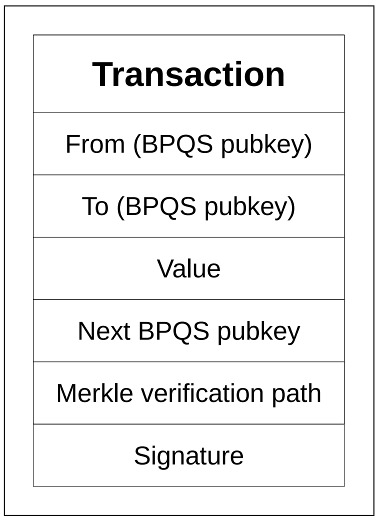

Figure 5: Transaction

state database as well as an secret seed. Every executed transaction is summarized in a block. A miner (authority) constructs those blocks when the transaction count reaches a threshold.
First, a network of processes is initialized, where one process is the miner and three other processes are nodes, which are like a miner without the power to create and publish blocks. The processes communicate through full-duplex pipes. This is depicted in figure 6.

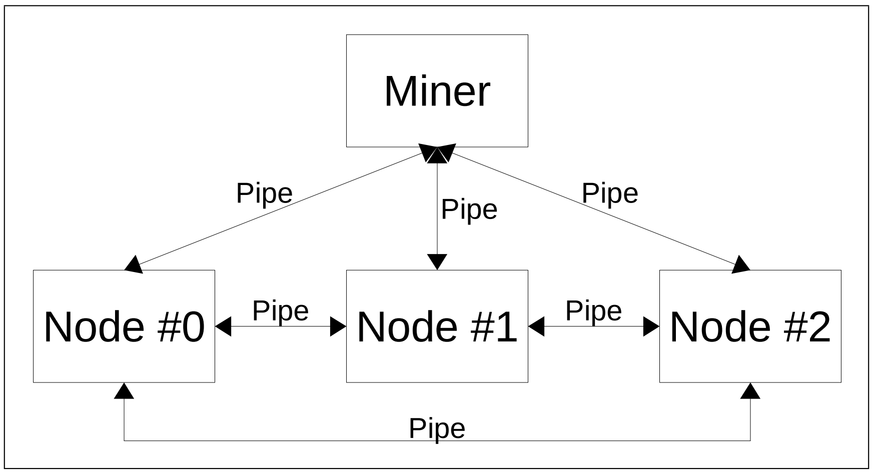

Figure 6: Peer-to-peer network

All those nodes get a blockchain with one block inside (the genesis block), which contains four transactions, one for each node and one for the miner. The transactions within the genesis block are executed and the results are stored in the state db of each participant. By doing so, every participant has an account after the initialization with some funds on it. Every participant then initializes a wallet with pre defined private keys, so that he has access to one of the four accounts which are by now stored in the state database. The miner can send blocks to the nodes, which signals them to update their state database according to the transactions contained in the block. This also synchronizes the states of the nodes. This procedure is depicted in figure 7.

Both, the miner and the nodes, randomly send transactions with random values to their peers. The nodes can additionally send invalid transactions, where the value exceeds their balance. After sending a transaction, the node waits for approval to know whether it can use the next BPQS-Tree (figure 8) or whether they should use a fallback key in the current BPQS-Tree (figure 9). The messages contain a type and data. Possible types are TX (0) for an incoming transaction, BLK (1) for an incoming block, ACP (2) for accepting a transaction and DNY (3) for denying a transaction.

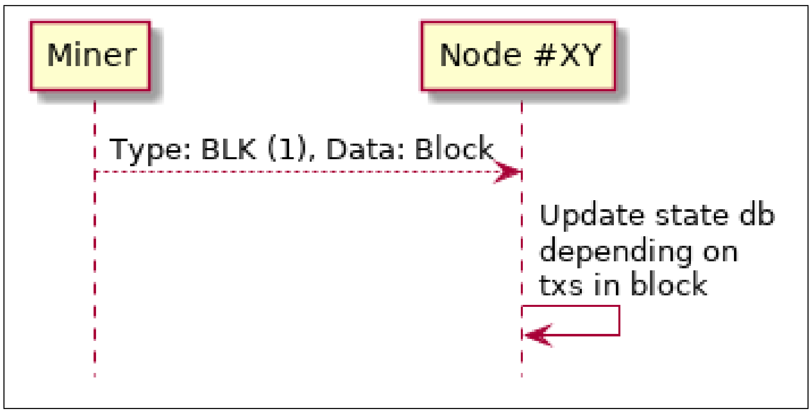
Miner
Node #XY
Type: BLK (1), Data: Block
Update state db depending on txs in block

Figure 7: Propagate block

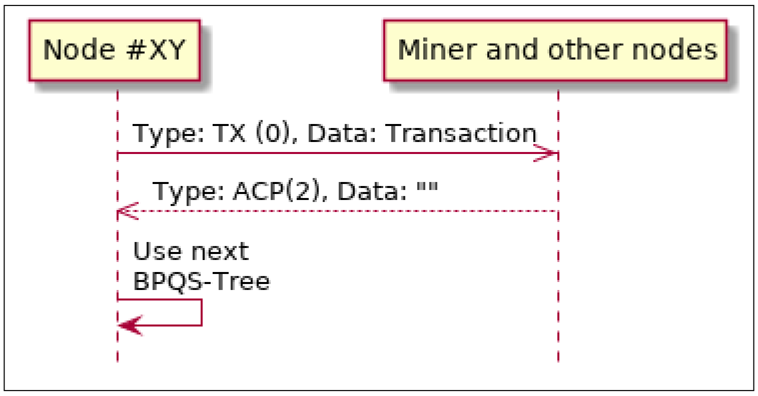

Figure 8: Send valid transaction, update state db

4 Implementation

This section describes the implementation details of the winternitz python package and the POC implementation of BPQS.

4.1 Winternitz python package

For detailed informations of the winternitz python package implementation, please view the code at `https://github.com/sea212/winternitz-one-time-signature` and the documentation at `https://winternitz-one-time-signature.readthedocs.io/en/1.0.2/`
Alternatively, a copy of the code was supplied with this documentation at "Code/winternitz-one-time-signature" and a copy of the documentation was supplied at "Documentation/winternitz-one-time-signature/user_guide.pdf"

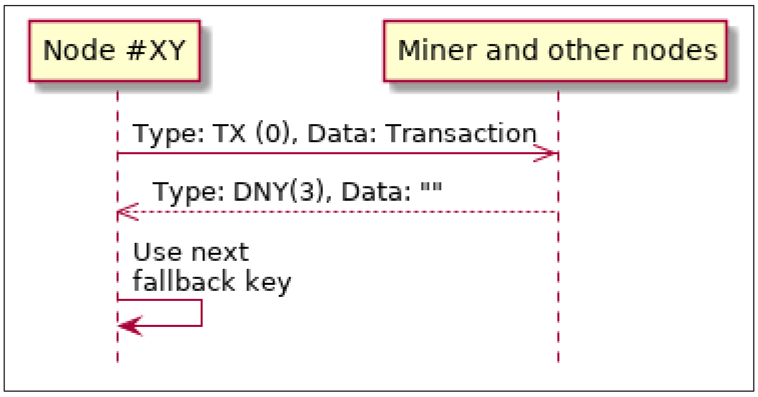

Figure 9: Send invalid transaction, use fallback key

In this context, especially the sections "Usage" and "winternitz" are relevant in the documentation.

4.2 POC of BPQS

The code to the POC implementation of BPQS was supplied with this documentation and can be found at "Code/bpqs-poc". To execute the implementation, change to the root directory of the implementation and run the following commands in your shell (tested with python 3.7.2):

```
pip install winternitz
python3 main.py
```

The first command installs the winternitz package, which was created during that project. The second command should execute the main.py file, which spawns four processes, one miner and three nodes, which then simulate a BPQS-FEW based cryptocurrency. The implementation is structured in the following way:

- main.py - main routine, execute this
- miner.py - miner process
- node.py - node process
- privkeys.py - private keys for each node and the miner
- transaction.py - used to verify a transaction, including the signature
- blockchain.py - creates initial blockchain with one genesis block
- wallet.py - Wallet that creates BPQS-Trees and manages the key. It also used to generate transactions

5 Conclusion

The winternitz implementation can now serve the community as a python standard package. It is setup in a way that everybody with the knowledge to do so can contribute. All together, it was an exiting experience to focus on creating a package as clean as possible with everything required for collaborative work.
The POC implementation of the BPQS scheme has proven that the idea can indeed be used in cryptocurrencies. It has shown it's advantages by reducing the additional hash outputs will providing the possibility to sign multiple times in case of error. Further additions to the POC implementation could be to implemented BPQS-GEN-B, which offers a many-time signature scheme in emergency situations when all few-time signatures are used. Furthermore, another addition could be to use the blockchain structure and the fact that every block within was already verified to reduce the additional hash outputs for the verification path and the verification time, since a partial verification path is already stored and verified.

References

[Buc+11] Johannes Buchmann et al. *On the Security of the Winternitz One-Time Signature Scheme*. Cryptology ePrint Archive, Report 2011/191. `https://eprint.iacr.org/2011/191`. 2011.

[Cha+18] Konstantinos Chalkias et al. *Blockchained Post-Quantum Signatures*. Cryptology ePrint Archive, Report 2018/658. `https://eprint.iacr.org/2018/658`. 2018.

[Hül17] Andreas Hülsing. *WOTS+ – Shorter Signatures for Hash-Based Signature Schemes*. Cryptology ePrint Archive, Report 2017/965. `https://eprint.iacr.org/2017/965`. 2017.

[JH11] Erik Dahmen Johannes Buchmann and Andreas Hülsing. *XMSS - A Practical Forward Secure Signature Scheme based on Minimal Security Assumptions*. Cryptology ePrint Archive, Report 2011/484. `https://eprint.iacr.org/2011/484`. 2011.

[Lam79] Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. Oct. 1979. URL: `https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/`.

[Woo14] Gawin Wood. *Ethereum: A secure Decentralised generalised transaction ledger*. 2014. URL: `https://ethereum.github.io/yellowpaper/paper.pdf`.