Quantum Resistant Ledger (QRL)

peterwaterland@gmail.com

November 2016

1 Introduction

The concept of a peer-to-peer internet ledger of value, recorded as a blockchain and secured by proof of work was first reported in 2008[10]. Bitcoin remains the most widely used cryptocurrency to date. Hundreds of similar cryptocurrency ledgers have been subsequently created but with few exceptions they rely on the same elliptic curve public-key cryptography (ECDSA) to generate digital signatures which allow transactions to be verified securely. The most commonly used signature schemes today such as ECDSA, DSA and RSA are theoretically vulnerable to quantum computing attack. It would be valuable to explore the design and construction of a quantum resistant blockchain ledger to counter the potential advent of a sudden non-linear quantum computing advance.

2 Bitcoin Transaction Security

It is currently only possible to spend (unspent transaction outputs) from a bitcoin address by creating a transaction containing a valid elliptic curve (secp256k1) signature from the private key ($x \in N | x < 2^{256}$) for that specific bitcoin address. If the truly randomly generated private key is kept secret or lost then no funds can reasonably be expected to move from that address ever.
The chance of a specific bitcoin private key collision is 1 in 2^{256}. The probability of any bitcoin address key collision can be estimated using the Birthday Problem. The number of bitcoin addresses which must be generated to result in a 0.1% chance of a collision is $5.4 \text{x} 10^{23}$[13].

However, when a transaction is signed then the ECDSA public key of the sending address is revealed and stored in the blockchain. Best practice is for addresses not to be re-used, but as of November 2016 49.58% of the entire bitcoin ledger balance is held in addresses with exposed public keys[1].

3 Quantum Computing Attack Vectors

RSA, DSA and ECDSA remain secure based upon the computational difficulty of factorisation of large integers, the discrete logarithm problem and the elliptic-curve discrete logarithm problem. Shor's quantum algorithm (1994) solves factorisation of large integers and discrete logarithms in polynomial time. Therefore, a quantum computer could theoretically reconstitute the private key given an ECDSA public key. It is thought ECDSA is more vulnerable to quantum attack than RSA due to the use of smaller keysizes, with a 1300 and 1600 qubit (2^{11}) quantum computer able to solve 228 bit ECDSA.

Public quantum computer development has not passed beyond 2^5 qubits or the factorisation of small numbers (15 or 21). However, in August 2015 the NSA deprecated elliptic curve cryptography ostensibly based upon quantum computing concerns. It is unclear how advanced quantum computing may be presently or that any breakthroughs in this field will be publicised to allow cryptographic protocols in common usage in the internet to be made post-quantum secure. With somewhat anti-establishment origins, bitcoin could find itself the earliest target of an adversary with a quantum computer.

If a significant quantum computing advance were to occur publicly, node developers could implement quantum-resistant cryptographic signature schemes into bitcoin and encourage all users to move their balances from ECDSA-based addresses to new quantum-safe addresses. To mitigate the proportion of effected addresses it would be reasonable to disable public key recycling at the protocol level. Such a planned upgrade would also result in the possible movement of the 1 million coins belonging to Satoshi Nakamoto - with associated price volatility.

A less favourable scenario would be a silent non-linear quantum computing advance followed by a nuanced quantum computing attack on bitcoin addresses with exposed public keys. Such thefts could have a devastating effect upon the bitcoin exchange price due to new heavy sell pressure and a complete loss of confidence in the system as the scale of thefts become known. The role of bitcoin as a store of value ('digital gold') would be very badly damaged with extreme consequences for the world. In this context the authors believe it is reasonable to experiment with quantum-resistant cryptographic signatures in a cryptocurrency ledger and potentially create a backup value store in the event of a black swan.

4 Quantum-resistant signatures

5 Hash-based digital signatures

Quantum resistant hash-based signatures rely upon the security of a one-way cryptographic hash function which takes a message, m and outputs a hash digest, h of fixed length, n. i.e SHA-256, SHA-512. Using a cryptographic hash function it should be computationally infeasible to brute force m from h (pre-image resistance), or brute force h from h_2, where $h_2 = hash(h)$ (second pre-image resistance), whilst it should be very hard to find two messages ($m1 \neq m2$) that produce the same h (collision resistance).

Grover's quantum algorithm may be used to attempt to find a hash collision or perform a pre-image attack to find m, requiring $O(2^{n/2})$ operations. Thus to maintain 128 bit security, a hash digest length, n of at least 256 bit must be selected - assuming a perfect cryptographic hash function.

Hash-based digital signatures require a public key, pk, for verification and a private key, sk for signing a message. Various hash-based one-time signatures (OTS) will be discussed regarding their suitability for inclusion as part of a blockchain ledger.

5.1 Lamport-Diffie OTS

In 1979 Lamport described a hash-based one-time signature for a message of length, m bits (usually the output of a collision resistant hash function). Keypair generation creates m pairs of random secret keys, $sk_j^m \in \{0,1\}^n$ where $j \in \{0,1\}$. i.e. the private key is: $sk = ((sk_0^1, sk_1^1), .., .., (sk_0^m, sk_1^m))$. Let f be a one-way hash function $\{0,1\}^n \rightarrow \{0,1\}^n$, with m pairs of public keys generated $pk_j^m = f(sk_j^m)$, i.e. the public key is: $pk = ((pk_0^1, pk_1^1), .., .., (pk_0^m, pk_1^m))$. Signing involves bitwise inspection of the message hash to select sk_j (i.e. if $bit = 0$, $sk_j = sk_0$, $bit = 1$, $sk_j = sk_1$), creating the signature: $s = (sk_j^1, .., sk_j^m)$ which reveals half the private key. To verify a signature, bitwise ($j \in \{0,1\}$) inspection of the message hash checks that $(pk_j = f(sk_j))^m$.

Assuming that after Grover's algorithm 128 bit security is desired, where message length is a fixed hash output from SHA256, $m = 256$ and $n = 256$, resulting in a $pk = sk =$ 16kb, and a signature of 8kb for each OTS used. A Lamport signature should be used only once, may be generated very quickly, but suffers from large key, signature and by extension transaction sizes, making it impractical for a public blockchain ledger.

5.2 Winternitz OTS

For a message digest, M, of length, m bits, with secret and public keys of length, n bits, a one-way function, $f : \{0,1\}^n \rightarrow \{0,1\}^n$ and a Winternitz parameter, $w \in N | w > 1$, the general idea of the Winternitz

one-time signature is to apply an iterating hash function on a list of random secret keys, $sk \in \{0,1\}^n$, $sk = (sk_1, .., sk_{m/w})$, creating chains of hashes of length, $w-1$, ending with public keys, $(pk \in N|\{0,1\}^n)$, $pk_x = f^{2^{w-1}}(sk_x)$, $pk = (pk_1, .., pk_{m/w})$.
Unlike the bitwise inspection of the message digest in in the Lamport signature, instead the message is parsed w bits at a time to extract a number, $i \in N, i < 2^w - 1$, from which the signature is generated, $s_x = f^i(sk_x)$, $s = (s_1, ..s_{m/w})$. With a growing w providing a tradeoff between smaller keys and signatures for increased computational effort.[9].
Verification involves simply generating $pk_x = f^{2^{w-1}-i}(s_x)$ from M, s and confirming the public keys match.

Using SHA-2 (SHA-256) as a one-way cryptographic hash function, f: $m = 256$ and $n = 256$, with $w = 8$ results in a $pk = sk = s$ size of $\frac{(m/w)n}{8}$ bytes = 1kb.
To generate pk requires f^i hash iterations, where $i = \frac{m}{w}2^{w-1} = 8160$ per OTS keypair generation. At $w = 16$ the keys and signatures halve in size, but $i = 1048560$ becoming impractical.

5.3 Winternitz OTS+ (W-OTS+)

Buchmann introduced a variant of the original Winternitz OTS by changing the iterating one-way function to instead be applied to a random number, x, repeatedly but this time parameterised by a key, k, which is generated from the previous iteration of $f_k(x)$. This is strongly unforgeable under adaptive chosen message attacks when using a pseudo random function (PRF) and a security proof can be computed for given parameters[3]. It eliminates the need for a collision resistant hash function family by performing a random walk through the function instead of simple iteration. Huelsing introduced a further variant W-OTS+, enabling creation of smaller signatures for equivalent *bit* security through the addition of a bitmask XOR in the iterative chaining function.[6] Another difference between W-OTS(2011 variant)/ W-OTS+ and W-OTS is that the message is parsed $log_2(w)$ bits at a time rather than w, decreasing hash function iterations but increasing keys and signature sizes.

W-OTS+ will now be briefly described. With security parameter, $n \in N$, corresponding to the length of message (m), keys and signature in bits, being determined by the cryptographic hash function chosen and the Winternitz parameter, $w \in N|\ w > 1$ (usually $\{4,\ 16\}$), l is computed. l is the number of n *bit* string elements in a WOTS+ key or signature, where $l = l_1 + l_2$:

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

A keyed hash function is used, $f_k : \{0,1\}^n \rightarrow \{0,1\}^n \mid k \in \{0,1\}^n$.
The chaining function, $c^i_k(x, r)$: on input of $x \in \{0,1\}^n$, iteration counter i, key, $k \in K$, and randomisation elements, $r = (r_1, .., r_j) \in \{0,1\}^{n*j}$, with $j \geq i$, is used as follows:
Where $i = 0$, $c^0_k(x, r)$ returns x.
For $i > 0$, $c^i_k = f_k(c^{i-1}_k(x, r) \oplus r_i)$ - that is a bitwise xor of the previous iteration of c_k and the randomisation element followed by f_k on the result, which is then fed into the next iteration of c_k.
To create the secret key, sk, $l + w - 1$ n *bit* strings are chosen uniformly at random (with PRF), of which the first l make up the secret key, $sk = (sk_1, ..sk_l)$ and the remaining $w-1$ n *bit* strings become $r = (r_1, .., r_{w-1})$.
A function key, k is chosen uniformly at random.
The public key is:

$$pk = (pk_0, pk_1, .., pk_l) = \{(r, k), c^{w-1}_k(sk_1, r), c^{w-1}_k(sk_2, r), .., c^{w-1}_k(sk_l, r)\}$$

To perform a signature, message M, of length m, is parsed such that $M = (M_1, ..M_{l_1}), M_i \in \{0, w-1\}$.
Next the checksum is calculated:

$$C = \sum_{i=1}^{l_1}(w - 1 - M_i)$$

INCOMPLETE

W-OTS provides a security level of at least $n - w - 1 - 2log(lw)$, the signature size is roughly mn/w bits, with $2^w m/w$ hash operations required[3].

the Winternitz one-time signature scheme (W-OTS) is existentially unforgeable under adaptive chosen message attacks

6 Merkle tree signature schemes

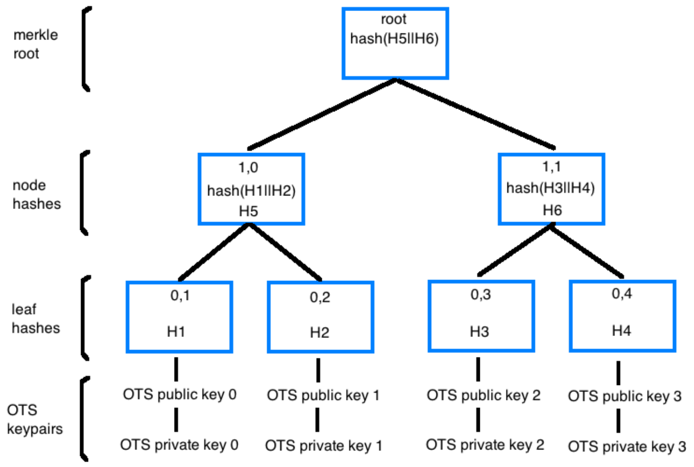
Whilst one-time signatures provide satisfactory cryptographic security for signing and verifying transactions they have a major drawback that they may only be used once safely. If a ledger address is based upon some transformation of the public key of a single OTS keypair then this would lead to an extremely restrictive blockchain ledger where all funds from a sending address would need to move with every single transaction performed - or those funds would be at risk of theft.
A solution is to extend the signature scheme to incorporate more than one valid OTS signature for each ledger address allowing as many signatures as OTS keypairs are pre-generated. A binary hash tree known as a merkle tree is a logical way to achieve this.

6.1 Binary hash tree

The general idea behind a merkle tree is an inverted tree composed of parent nodes computed by hashing the concatenation of child sibling nodes upwards in layers to the root. The existence of any node or leaf can be cryptographically proven by computing the root.

A merkle tree is formed from n base leaves and has height to merkle root, h ($n = 2^h$) - starting from the leaf hashes (layer 0) and counting upwards with each layer of nodes. Each leaf node is created in our hypothetical ledger use-case by hashing a randomly pre-generated OTS public key. From the tree below it can be seen that the node above each pair of leaf hashes is itself formed by hashing a concatenation of the child hashes.

Figure 1. Merkle tree signature scheme example



This continues upwards through layers of the tree until confluence into the root hash of the tree, known as the merkle root.

From the example tree in the diagram, taking the merkle root as the public key, four pre-computed OTS keypairs can be used to generate four cryptographically secure valid one-time signatures. The merkle root of the binary hash tree can be transformed into a ledger address (possibly by iterative hashing with an appended checksum). A full signature, S, of a message, M, for a given OTS keypair includes: the signature,

s, the ots key number, n, and the merkle authentication path. i.e. for OTS keypair 0 (thus $n = 0$):

$$S = s,\ n,\ \mathit{OTS\ public\ key}\ 0,\ H1,\ H2,\ H5,\ H6,\ \mathit{root}$$

Given the OTS public key and leaf hash can be deduced from s, and parent nodes can be computed from their children in fact this may be shortened to:

$$S = s,\ n,\ H2,\ H6,\ \mathit{root}$$

Where S is valid by verifying the OTS public key from s and M, then checking the hashes from the merkle authentication path recreate a matching merkle root (public key).

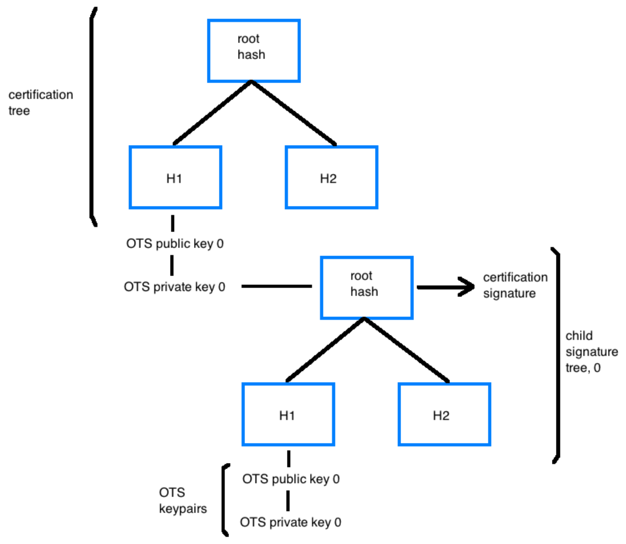
6.2 State

Using the above merkle signature scheme (MSS) securely relies upon not re-using the OTS keys. Therefore, it is dependent upon the state of signatures or signed transactions being recorded. Ordinarily in real world usage this would potentially be a problem, but an immutable public blockchain ledger is the ideal storage medium for a stateful cryptographic signature scheme. A newer hash-based cryptographic signature scheme called SPHINCS which offers practical stateless signatures with 2^{128} bit security was reported in 2015[2].

6.3 Hypertrees

A problem with the basic MSS is that the number of signatures possible is limited and all the OTS keypairs must be pre-generated prior to calculation of the merkle tree. Key generation and signature time grow exponentially with the height of the tree, h, meaning trees larger than 256 OTS keypairs becomes temporally and computationally expensive to generate.
A strategy to defer computation during key and tree generation and also extend the number of OTS keypairs available is to use a tree which is itself composed of merkle trees called a hypertree. The general idea is to sign the merkle root of a child tree with an OTS key from the leaf hash of a parent merkle tree known as a certification tree.

Figure 2. linking merkle trees



In the most simple form (height, $h = 2$) a certification tree is precomputed with 2^1 OTS keypairs and when the first signature is required a new signature merkle tree (signature tree 0) is computed and signed by one of the certification tree OTS keypairs. The signature tree is composed of n leaf hashes with corresponding

OTS keypairs and these serve to sign messages as required. When each OTS keypair in the signature tree has been used then the next signature tree (signature tree 1) is signed by the second OTS keypair from the certification tree and the next batch of signatures is possible.

A signature, S, of such a hypertree construction becomes slightly more complicated and would include:
1. from the signature tree: s, n, *merkle path*, *root*
2. from each certification tree: s (of child tree merkle root), n, *merkle path*, *root*

It is theoretically possible to nest layers of trees down from the certification tree to extend the original MSS infinitely. Signature size grows linearly for each additional tree that is signed, whilst the hypertree signature capacity increases exponentially.

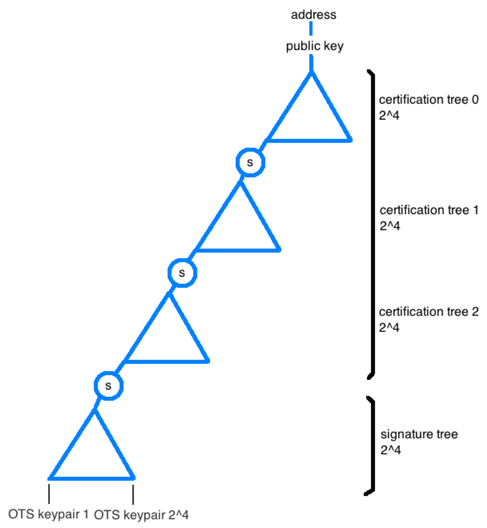
6.3.1 Hypertree examples

To demonstrate how easily the MSS may be extended with a hypertree construction consider an initial certification tree of height, $h_1 = 5$, with 2^5 leaf hashes and associated OTS keypairs. The merkle root of this tree is transformed to generate a ledger address. Another merkle tree, a signature tree of identical size ($h_2 = 5$, 2^5 leaves and OTS keypairs) is instantiated. 32 signatures are possible before the next signature tree must be created. The total number of signatures possible is $2^{h_1+h_2}$ which in this case is $2^{10} = 1024$.

On a Macbook pro 2.7Ghz i5, 8gb ram creating OTS keypairs and a merkle certification tree for various sizes yielded the following results (unoptimised python code, Winternitz OTS): $2^4 = 0.5s$, $2^5 = 1.2s$, $2^6 = 3.5s$, $2^8 = 15.5s$. A hypertree consisting of initial generation of two 2^4 trees takes around $1s$ compared with $15.5s$ for standard 2^8 MSS tree for the same signature capacity.

Increasing the depth (or height) of a hypertree continues this trend. A hypertree composed of four chained 2^4 certification trees and a signature tree of size 2^4 is capable of $2^{20} = 1,048,576$ signatures with an added cost to the signature size but a creation time of only $2.5s$.

Figure 3. Hypertree construction

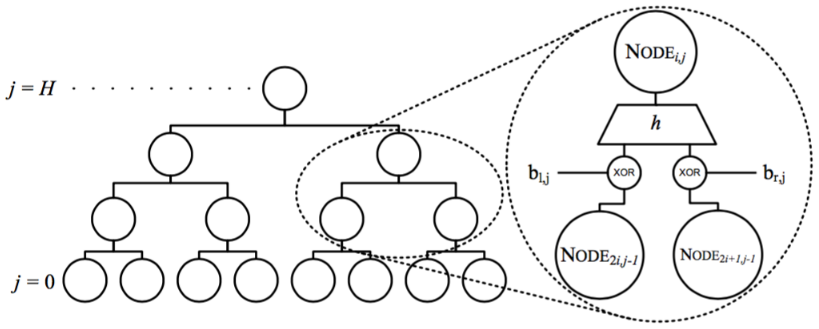


There is no requirement for a hypertree to be symmetrical and so if composed initially of two trees it could later be extended later by signing further layers of trees. Signatures from a ledger address would therefore start small and eventually rise as the hypertree depth increased. Using a merkle hypertree to create and sign transactions from a ledger address is never likely to require $> 2^{12}$ transactions. Thus, the ability to sign with computational ease 2^{20} times securely at a hypertree depth of $h = 5$ is more than sufficient.

6.4 XMSS

The extended merkle signature scheme (XMSS) was first reported by Buchmann et al. in 2011 and was published as an IETF draft last year[4][7]. It is provably forward secure with minimal security requirements: a PRF and a second pre-image resistant hash function. The scheme allows extension of one-time signatures via a merkle tree with a major difference being the use of bitmask XOR of the child nodes prior to concatenation of the hashes into the parent node. The use of the bitmask XOR allows the collision resistant hash function family to be replaced.

Fig. 1. The XMSS tree construction



The leaves of the tree are also not OTS keypair hashes but the root of child L-trees which hold the OTS public keys with, l pieces forming the base leaves. Winternitz OTS+ is used for the one-time signatures (though 2011 variant was first described).

The bit length of the XMSS public key is $(2(H + \lceil logl \rceil) + 1)n$, an XMSS signature has length $(l + H)n$, and the length of the XMSS secret signature key is $< 2n$.

Buchmann reports performance with an Intel(R) $i5$ 2.5 Ghz for an XMSS tree of height, $h = 20$, where $w = 16$ and the cryptographic hash function used is SHA-256 ($m = 256$) of up to around a million signatures. With the same parameters and hardware, signing took 7ms, verification 0.52ms and key generation 466 seconds. The security level achieved with such parameters was 196 bits for a public key size of 1.7kb, private key of 280 bits and signature 2.8kb. XMSS is an attractive scheme with the main drawback being the extremely long key generation time.

7 Proposed signature scheme

7.1 Security requirements

In the design of the QRL it is important that the cryptographic security of the signature scheme is secure against classical and quantum computing attack both in present day and also future decades. XMSS using SHA-256, where $w = 16$, offers 196 bit security with predicted safety against brute force computational attack until the year 2164[8].

7.2 QRL signatures

A stateful assymetrical hypertree signature scheme composed of chained XMSS trees is proposed. This has the dual benefit of utilising a validated signature scheme and allowing generation of ledger addresses with the ability to sign transactions avoiding a lengthy pre-computation delay. W-OTS+ is the chosen hash-based one-time signature in the scheme.

7.3 Hypertree construction

7.3.1 Key and signature sizes

As the number of trees within a hypertree grows, key and signature sizes grow linearly - but signature capacity rises exponentially. Sizes for various XMSS tree derived public keys and signatures (based upon 2011 description), where $w = 16$, $m = 256$, H is tree height and SHA-256 is chosen as the cryptographic hash algorithm, are:

- $H = 1$, 2^1 signatures: public key 0.53kb, signature 2.1kb
- $H = 2$, 2^2 signatures: public key 0.59kb, signature 2.12kb
- $H = 3$, 2^3 signatures: public key 0.66kb, signature 2.16kb
- $H = 4$, 2^4 signatures: public key 0.72kb, signature 2.19kb
- $H = 5$, 2^5 signatures: public key 0.78kb, signature 2.21kb
- $H = 16$, 2^{16} signatures: public key 1.47kb, signature 2.56kb
- $H = 20$, 2^{20} signatures: public key 1.7kb, signature 2.69kb

Thus key and signature sizes for an XMSS-based hypertree of depth, $j \in \{0, 5\}$, composed of XMSS trees of height, h, can be inferred as:

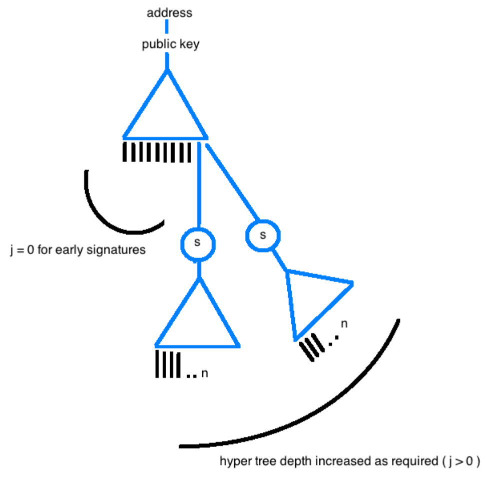
- $j = 0$, $h = 4$, 2^4 signatures: public key 0.72kb, signature 2.19kb
- $j = 1$, $h = 4$, 2^8 signatures: public key 1.44kb, signature 4.38kb
- $j = 2$, $h = 4$, 2^{12} signatures: public key 2.16kb, signature 6.57kb
- $j = 3$, $h = 4$, 2^{16} signatures: public key 2.88kb, signature 8.76kb
- $j = 4$, $h = 4$, 2^{20} signatures: public key 3.6kb, signature 10.95kb
- $j = 0$, $h = 5$, 2^5 signatures: public key 0.78kb, signature 2.21kb
- $j = 1$, $h = 5$, 2^{10} signatures: public key 1.56kb, signature 4.42kb
- $j = 2$, $h = 5$, 2^{15} signatures: public key 2.34kb, signature 6.63kb
- $j = 3$, $h = 5$, 2^{20} signatures: public key 3.12kb, signature 8.84kb

The tradeoff for creating an XMSS hypertree (4 trees, $j = 3$, $h = 5$) with eventual signature capacity of 2^{20} in less than 3s compared with 466s, for a signature of 8.84kb instead of 2.69kb is acceptable.

7.3.2 Asymmetry

Creating an asymmetrical tree allows early signatures to take place with with a single xmss tree construction, which is extended as required for later signatures at a cost to overall signature capacity. The rationale is that this is likely to be of no consequence for a blockchain ledger application and the wallet can give a user an option of signature capacity versus signature/key sizes. A maximum tree depth of $j = 5$ should suffice.

Figure 3. Asymmetrical hypertree


address
public key
j = 0 for early signatures
s
s
n
n
hyper tree depth increased as required (j > 0)

7.3.3 QRL hypertree specification

As previously discussed the general idea is to sign from a root certification tree a chain of further certification trees which terminate with a signature tree which is used to sign actual transactions.

The following parameters are to be adopted:

- $j = 3$, $h = 5$, upper bound of signatures possible: 2^{20}, minimum signature size: 2.21kb, maximum signature size: 8.84kb.

8 Cryptocurrency design parameters

The remainder of the white paper will set out the proposed design parameters for the QRL ledger. The focus of the ledger is to be a public blockchain which is highly secure against classical and quantum computing attack vectors. This is a first draft and thus every aspect is subject to potential change.

8.1 Proof-of-work

QRL is to be an open public blockchain ledger secured by proof-of-work. The exact proof-of-work algorithm has yet to be chosen but is unlikely to be SHA-256 based.

A more interesting question is over the longer term whether mining will remain quantum-resistant - see Grover's algorithm earlier. A ideal medium term view would include a transition to proof-of-stake as this is less energy intensive and not vulnerable to any form of computational attack.

8.2 Fees

The larger transaction sizes compared with other ledgers necessitates a transaction fee must be paid with each transaction. The author is of the opinion that artificial fee markets (see bitcoin) are unnecessary and run counter to the ideal of an open public blockchain ledger. Each transaction if it pays a minimum fee should be as valid as any other. The minimum fee miners are willing to accept should float and be set by the market. i.e. nodes/miners competitively set the lower bound of fees between themselves. An absolute minimum value will be enforced at protocol level. Thus, miners will order transactions from the mempool for inclusion in a block at their discretion.

8.3 Blocks

8.3.1 Block-times

Bitcoin has a time between blocks of roughly 10 minutes, but with natural variance this can on occasion lead to fairly long periods before the next block is mined. Newer ledger designs such as ethereum have improved upon this and benefit from a much shorter block-times (15 seconds) without the loss of security or miner centralisation from high rates of orphan/stale blocks. Ethereum uses a modified version of the Greedy Heaviest Observed Subtree protocol which allows stale/orphan blocks to be included in the blockchain and rewarded[12, 5].

QRL will replicate the system used in ethereum (from [5]) to include uncle/ommer blocks as follows :

- An uncle included in block, B, must have the following properties:
- It must be a direct child of the k^{th} generation ancestor of B, where $k = \{x \in N | \ 2 \leq x \leq 7\}$
- It cannot be an ancestor of B An uncle must be a valid block header, but does not need to be a previously verified or even valid block
- An uncle must be different from all uncles included in previous blocks and all other uncles included in the same block (non-double-inclusion)
- For every uncle, U, in block B, the miner of B gets an additional 3.125% added to its coinbase reward and the miner of U gets 93.75% of a standard coinbase reward.

This will allow QRL to safely use a block-time of between 15 and 30 seconds.

8.3.2 Block-rewards

Each new block created will include a first 'coinbase' transaction containing a mining address into which a reward equal to the sum of the coinbase reward, the combined sum of transaction fees within the block and a 3.125% of the coinbase reward for each additional uncle block included.

The block-reward is re-calculated by the mining node every block and follows the coin emission schedule.

8.3.3 Blocksize

To avoid controversy an out-of-the-box adaptive solution modelled upon the Bitpay proposal to increase the blocksize based upon a multiple, x, of the median size, y of the last z blocks would be employed[11]. The use of the median prevents gaming by miners to include either empty or overstuffed blocks to alter the mean blocksize. x and z would then be hard consensus rules for the network to obey.

Thus, a maximum blocksize, b could be simply calculated as:

$$b = xy$$

8.4 Currency unit and denominations

The QRL will use a monetary token, the *quantum* (plural *quanta*), as the base currency unit. Each *quantum* is divisible to a smallest element as follows:

- 1 : Shor
- 10^3 : Nakamoto
- 10^6 : Buterin
- 10^{10} : Merkle

- 10^{13} : Turing
- 10^{16} : Quantum

Thus, each transaction involving a fraction of a *quantum* is actually a very large integer of *Shor* units. Transaction fees are paid and calculated in *Shor* units.

8.5 Accounts

User balances are kept in accounts. Each account is simply a unique re-usable ledger address denoted by a string beginning with 'Q'.

An address is created by performing a SHA-256 upon the merkle root of the MSS certification tree. A four byte checksum is appended to this (formed from the first four bytes of a double SHA-256 hash of the merkle root) and the letter 'Q' prepended. i.e. in pythonic pseudocode:

$$Q + sha256(merkle\_root) + sha256(sha256(merkle\_root))[:4]$$

A typical account address:

$$Qcea29b1402248d53469e352de662923986f3a94cf0f51522bedd08fb5e64948af479$$

Each account has a balance denominated in *quanta* divisible down to a single *Shor* unit.

Addresses are stateful with each transaction using a fresh OTS keypair and the QRL storing every public key ever used (this could be pruned as it can be regenerated on-the-fly from the transaction signature and message but would be operationally intensive) for each account. A transaction counter called a nonce is incremented with each transaction sent from an account. This allows wallets which do not store the entire blockchain to keep track of their location in the stateful merkle hypertree signature scheme.

8.6 Coin issuance

8.6.1 Historical considerations

Bitcoin was the first decentralised cryptocurrency and initially experimental with no monetary value, so it was appropriate to distribute the currency entirely through mining. More recently Zcash has chosen the same process with a % of the coinbase mining reward in the early period of emission being passed to the open source project - with resultant incredible price volatility.
Other ledgers such as ethereum have instead sold a large % of the final coin supply as part of an initial coin offering (ICO). This has the benefit that early adopters still potentially gain through supporting the project, but additionally the project itself may generate funds to continue development and bootstrap and grow the project from infancy. The ICO approach also allows a market to develop easily as a larger float of coins is available to investors to buy and sell from the genesis block.
Auroracoin (2014) took a different approach offering everyone in Iceland an equal share of the ICO, while developers kept 50% of the entire coin supply for themselves.
Other cryptocurrencies have either simply cloned bitcoin entirely or started afresh with a new chain but different codebase.

8.6.2 Interchain balance transfer

It is possible to issue the QRL based upon a state snapshot of the current bitcoin ledger inserted in the QRL genesis block. The general idea would be to allow users to create a single use 'import' transaction containing a unique message and signature (i.e. a randomly generated QRL wallet address signed with a bitcoin private key from an address with a bitcoin balance at the time of the snapshot). This feature could remain active up to a certain blockheight and thereafter the remainder of the coin supply would be mined as normal. The initial bloat in the genesis block would be pruned at the same blockheight. A drawback with this is that whilst fair it penalises holders of other cryptocurrencies other than bitcoin and is technically potentially challenging for new users to perform. A technical concern may be that it is possible to recover an ECDSA public key from just a signature and the message. This would permanently expose public keys to bitcoin addresses used in the process and it would be important to move funds afterwards to a new randomly generated bitcoin address to mitigate this.

(?could allow the same feature for ethereum holders)

8.6.3 Proposed Issuance - draft

The QRL initial issuance will be the following:

- ICO of 1 million *quanta* (4.7% final coin supply) prior to launch.
- A state snapshot of all bitcoin address balances above 0.01 btc used to form the initial QRL genesis block. Anyone wishing to directly transfer coins at a 1:1 ratio from the bitcoin ledger to the QRL ledger will be able to do so until blockheight 518400 (3 - 6 months) via the node wallet.
- A further 1 million *quanta* will be held in a genesis block address for use by foundation
- The remaining supply will be mined ($21,000,000 - (2,000,000+$btc balances imported by blockheight 518400))

8.7 Coin emission schedule

A defining feature of bitcoin is the scarcity and fixed upper limit to issuance of the underlying monetary token. QRL will follow bitcoin in this regard with a fixed upper limit to the coin supply of 21x10^6 *quanta*. A smoothly exponential decay in the block-reward is favoured up to the hard ceiling of coin supply. This will eliminate the volatility associated with the bitcoin 'halving' phenomenon.

The total coin supply, $x = 21\text{x}10^6$, minus coins created at the genesis block, y, will exponentially reduce from Z_0 downwards forever. The decay curve is calculated to distribute mining rewards for approximately 200 years (until 2217AD, 420480000 blocks at 15s block-times) until just a single *quanta* is left unmined (though mining could continue thereafter).

The remaining coin supply at block t, Z_t, may be calculated with:

$$Z_t = Z_0 e^{-\lambda t}$$

The coefficient, λ, is calculated from: $\lambda = \frac{lnZ_0}{t}$. Where t, is the total number of blocks in the emission schedule until the final *quanta*. Until block 518400, $\lambda = 3.98590885111\text{x}10^{-08}$. The block reward, b is calculated for each block with:

$$b = Z_{t-1} - Z_t$$

Between the genesis block and block number 518400 bitcoins balances may be transferred onto the ledger via import transactions. At block 518401 the emission schedule will retarget locking in the new imported balances, reducing Z_t and adjusting the block-reward accordingly.

References

[1] http://oxt.me/charts.

[2] D. Bernstein. Sphincs: practical stateless hash-based signatures. 2015.

[3] J Buchmann. On the security of the winternitz one-time signature scheme.

[4] J. Buchmann. Xmss – a practical forward secure signature scheme based on minimal security assumptions. 2011.

[5] V Buterin. Ethereum whitepaper. 2013.

[6] A. Hulsing. W-ots+ - shorter signatures for hash-based signature schemes. 2013.

[7] A. Hulsing. Xmss: Extended hash-based signatures. 2015.

[8] A. Lenstra. Selecting cryptographic key sizes. 2001.

[9] R. Merkle. A certified digital signature. *CRYPTO*, 435, 1989.

[10] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[11] S Pair. A simple, adaptive blocksize limit. 2016.

[12] Yonatan Sompolinsky. Accelerating bitcoin's transaction processing fast money grows on trees, not chains. 2014.

[13] A. Toshi. The birthday paradox. 2013.