Quadrans Blockchain

Yellow Paper - v1.0

Michele Battagliola*, Andrea Flamini†
Riccardo Longo‡, Alessio Meneghetti§, Massimiliano Sala¶

November 16, 2021

Abstract

In this document we present the design of Quadrans, a blockchain platform for supply chains and IOT devices, capable of performing trustworthy and transparent operations. We aim to achieve scalability without losing security or decentralisation. Furthermore we use an innovative approach to the cryptographic layer of the ledger that gives back control to users allowing great flexibility, achieving also resiliency in case of disrupting cryptanalysis advancements. We also natively support Post-Quantum algorithms and enforce their use in block signing, while supporting lightweight encryption for IOT devices. This paper has been revised by Davide Costa[1] and Fabio Fiori[2] from Quadrans Foundation.

Contents

*michele.battagliola@unitn.it, *Department of Mathematics, University of Trento*
†andrea.flamini.1995@gmail.com, *Department of Mathematics, University of Trento*
‡riccardolongomath@gmail.com, *Department of Mathematics, University of Trento*
§almenegh@gmail.com, *Department of Mathematics, University of Trento*
¶maxsalacodes@gmail.com, *Department of Mathematics, University of Trento; CryptoBoard Quadrans Foundation*
[1]Davide Costa, davide.costa@quadrans.io
[2]Fabio Fiori, fabio.fiori@quadrans.io

1 Introduction

The potential of distributed smart contracts and of the blockchain (a public data structure very resilient and openly auditable) has been apparent since the very beginning [7, 11], and great interest has risen especially in particularly suitable sectors such as financial institutions and supply chains. Nowadays there are various platforms that offer interesting solutions specifically tailored to improve or facilitate supply chains, but this sector is huge and variegated, and there still are many problems not yet addressed.

Here we propose the technical specification for a decentralised platform devoted to smart contracts [12, 13] with a specific focus on the needs of Industry, complex supply chains, IOT devices, with significant designing efforts on the security of this platform and its cryptographic and protocol-related aspects. The proposed blockchain is called Quadrans Blockchain (QB) and consists of three sub-blockchains, the SynchroChain, the MasterChain and the ShardChain. QB supports two currencies: Quadrans Tokens (QDTs) and Quadrans Coins (QDCs). Users possessing QDTs enjoy special priviledges, and are called TokenHolders.

Background

We started working taking into consideration the very long and articulated food supply chain because it starts from crop fields and it ends to restaurants and shops. Transparency in the processing and geographical indications schemes such as *DOC* and *IGP* can add substantial value to a product, and customers grow ever more conscious of the importance of the origin of what they eat. Food apart, there are many sectors we are working with, e.g. textile, pharma, energy and public administration. There are tangible economic incentives in adopting technologies that enforce a controlled, secure and publicly auditable supply chain. In this scenario there are many delicate aspects that need to be taken care of: data needs to be tamper-proof and reliably authenticated and verified, some information may be confidential and only shared with specific partners, participation has to be adequately rewarded while preventing and discouraging malicious behaviour.

2 Algorithms and Parameters Flexibility

Quadrans is a blockchain designed to support interaction among users with peculiar and heterogeneous characteristics: from IOT devices, that power monitoring and automatic update of product status in the supply chain, to end consumers and companies. Such diversity calls for considerable levels of flexibility when choosing cryptographic primitives, because the computational resources at the disposal of each actor may differ quite a lot.

Moreover the realm of information security is in constant evolution, so it is of paramount importance to have a flexible and resilient approach, capable of adapting to novel discoveries and improvements. In fact technological advancements may render current solutions insecure or open the way to much more efficient alternatives, and scientific research could have disruptive cryptanalytic breakthroughs or discover powerful new approaches.

For these reasons we adopt a flexible approach to the selection and usage of cryptographic primitives and their parameters, favouring some standard choices to optimise efficiency but allowing each user to employ its own choice, in order to balance between security and computational cost, or even to avoid suspicious constructions (i.e. fear of backdoors).

2.1 Smart Contract Cryptographic Kernel

To manage this ductile design we employ a system of special smart contracts that act like a sort of *cryptographic kernel*. In particular they define and encode the available algorithms for each type of primitive, and validate and organise their parameters.

Quadrans allows the usage of any algorithm of a non-static pre-approved list, which initially contains our starting selection. A smart contract updatable every epoch is responsible to introduce new approved algorithms and maintain a sub-list of up to 63 standards that are efficiently encoded, from a total of up to 2^{16} different algorithms (a limit that we consider over-conservative). For every scheme in the list, there is a dedicated smart contract that defines the operating parameters of the algorithm.

In more detail the smart contract specifies three standard sets of parameters that correspond to just as many levels of security:

- *Basic*: suitable also for less powerful devices and interaction with legacy applications;
- *Intermediate*: a middle tier that strengthens the security while balancing efficiency;
- *Enhanced*: a higher-security level, that is fit for more sensitive and long-term information.

In addition to these standards, users are allowed to adopt their own parameters, so that usage can be furthermore tailored to specific needs (for example users can adopt their own elliptic curves), submitting them to the smart contract. In fact the same smart contract is used also to validate and assess proposed sets of parameters, to ensure that they satisfy a minimum level of security (that can be lower than the one of the basic parameters, to reach out to users that might have stringent constraints).

This approach optimises storage space and validation effort, since each set is stored and validated only once. Each smart contract in the kernel specifies a list of set of parameters and three indexes that identify which entries correspond to the three standards, so that the standard can easily be updated to adapt to cryptanalytic breakthroughs maintaining optimal encoding for these preferred sets. To preserve integrity and backwards compatibility the previous standard indexes are also shown by the smart contract, alongside the timestamp (epoch block index) of the moment the outdated value has been superseded. At the beginning these standard indexes are `00000000`, `00000001`, `00000002`, so the first three entries in the parameter list are initialised with the starting choices for the three standards, later the list can be expanded up to $2^{32} - 1$ (more than four billion) entries than can therefore be indexed with a four-byte value.

The other information stored in the smart contract is a hash-link to the encoding specification that defines how to actually encode parameter sets and interpret related values (e.g. public keys for a signature scheme). Users can then submit their parameter set of choice to the smart contract, which will append it to the list (encoded according to the specification) if the entry is not a duplicate and if the parameters pass a series of tests that establish that they properly define a correct instance of the signature algorithm and sustain a minimum level of security.

More details and the initial proposals for the encodings are presented in [14].

2.2 Encoding

The encoding of algorithm choice has been developed so that the majority of cases require a single byte, that is called the *discerning byte*. In particular this byte encodes the index of the standard algorithm (in the six most significant bits), and the standard set of parameters (in the crumb composed by the two least significant bits). The crumbs 01, 10 and 11 correspond to the standard parameters associated to the aforementioned levels of security, namely the basic, the intermediate and the enhanced level; the crumb 00 signals a non-standard choice and therefore it must be followed by additional 4 bytes that specify the index of the chosen parameters in the smart contract list. Similarly the index 000000 for the standard algorithm signals a non-standard algorithm, so 2 additional bytes that specify the index of the chosen algorithm follow. A null byte indicates a non-standard algorithm with non-standard parameters, so it is followed by 6 bytes: first 2 for the algorithm, then other 4 for the parameters.

So the *discerning byte* (or the following 2 bytes) identifies the smart contract that lists the parameters and specifies if a standard set is used; if that is not the case the index of the chosen set follows, so one can retrieve the value of the parameters from the smart contract. At this point algorithm and parameters are established, so the following bytes can be correctly interpreted referring to the specification defined in the smart contract (e.g. as a public key).

Example 1. Suppose that the following string is the hexadecimal representation of a public key:

```
0582006e9398a6986eda61fe91674c3a108c399475bf1e738f19dfc2db11db1d28
```

The first byte (`05`) identifies the scheme and therefore how to process the rest. Its binary representation is:

```
000001 01
```

The first (and most significant) six bits identify the first standard algorithm defined in the *digital signature algorithms* smart contract (e.g. ECDSA), the last (and least significant) two bits say that the basic-level standard is used (referring to the *ECDSA parameters* smart contract, e.g. the curve *secp256k1*). This means that what follows the first byte is the encoding of the public key, since the curve parameters are established and known. Note also that the parameters define the length of the encoding, so the parser knows how many bytes to read. ECDSA's public key is a point of an elliptic curve, and the parameters used imply the quadratic residuosity of the second coordinate, so the remaining sixteen bytes of the public key are the first coordinate of the point.

3 Users

Users of Quadrans blockchain are identified by their address. Such an address is derived from a public key whose correspondent private key is owned by the user it identifies. Quadrans allows its users to choose from various digital signature algorithms for transaction signing, therefore the format of such public keys can vary but the format of the address is standardised for everyone. With "user"

we always mean a hardware/software interacting with the Quadrans Blockchain, rather than the actual person who owns the hardware/software.

3.1 Digital Signatures

Quadrans exploits the flexible adoption of various digital signature algorithms as described in Section 2 to let each user balance security and computational cost, moreover we distinguish between transaction-signing and block-signing, requiring much higher levels of security for block signatures.

In fact we can assume that blocks are created and validated by fairly powerful users of the network that can surely afford to invest more resources in order to strengthen the long-term safety of the whole chain. Considering also the relative proportion between blocks and transactions (the former comprise hundreds of instances of the latter) we can employ signatures that are much more space-consuming for the blocks without debilitating the overall efficiency. For these reasons Quadrans blocks must be signed with Post-Quantum secure algorithms.

3.1.1 Available Digital Signature Algorithms

The initial selection of approved schemes is the following, where we present public-key length, signature length, and compare their combined length (w.r.t. a standard security level of 128 bit).

- `ECDSA` [22] is the widespread standard, especially in the blockchain world. This means that implementations are widely available, even with optimisations for low-power devices. Both public keys and signatures are very short (32 bytes once the curve has been fixed), and together with its low computational cost this makes ECDSA an efficient signature. However the security is based on the difficulty of the discrete logarithm (DLOG) over the group of points of an elliptic curve, and Shor's algorithm for quantum computers breaks this assumption [31]. Therefore this choice is not suitable for block signing, where a post-quantum secure algorithm is required.

- `EdDSA` [2, 4] is another signature scheme based on elliptic curves, that is gaining advantage in terms of usage on the more classical ECDSA. The particular types of curves employed (the so called Twisted Edwards curves [1]) make computations less susceptible to side-channel attacks and allow for further optimisation especially in batch signature verification. Another practical advantage is the deterministic nature of the signature that avoids the pitfalls of incorrect generation of random parameters, which have led to complete breaches of ECDSA signatures in the past [30]. Key and signature sizes are equal to those using ECDSA, the computational cost is comparable or lower, and the security is based on the same mathematical assumption: so EdDSA is an interesting alternative to ECDSA, but is likewise unsuitable for block signing.

- `CRYSTALS-DILITHIUM` [16] is a lattice-based post-quantum signature algorithm. The primary advantage of Crystals-Dilithium over similar PQ proposals is the avoidance of Gaussian Sampling [26], which is a primitive easy to misuse [5] and that may lead to weakening attacks hard to

detect. Public keys are a little more than 1 KB long, with signatures of roughly 2 KB. This is almost two orders of magnitudes more than elliptic curve-based (EC) schemes, but it is a price to pay for post-quantum security. This scheme has one of the lowest combined length of public key and signature among PQ algorithms.

- `FALCON` [20] is also a lattice-based post-quantum signature algorithm. It features small signatures and key sizes against other PQ schemes, but it uses Gaussian Sampling that has some potential issues (see above). Public keys are little less than 900 B, and signatures around 650 B: the shortest combined length among NIST round 2 candidates, but they are still an order of magnitude longer than EC signatures.

- `SPHINCS+` [3] is a post-quantum signature algorithm, this time based on hash functions. The construction makes few security assumptions (so it should be more resilient), and the implementation is quite similar to a scheme which has already been adopted internationally (XMSS [6]). However implementations should account for fault attacks, that could allow signature forgery at a reasonably-low computational cost. Public keys are only 32 B long, however signatures are quite long (around 8 KB).

- `GeMSS` [8] is a post-quantum signature algorithm, based on multivariate polynomials. It is an evolution of the well studied scheme QUARTZ [28], obtained by incorporating in it the latest research results in terms of security and efficiency. Like other multivariate schemes, it features small signatures but has large public keys: signatures are only around 48 B, public keys are 417.4 KB long.

- `PICNIC` [10] is a post-quantum signature algorithm, based on a combination of block ciphers, hash functions, and zero-knowledge proofs. The scheme is designed to be extremely compact for hardware implementations, allowing for easy hardware acceleration to make the cipher lightweight on low-power devices. It is also nearly compatible with current X.509 certificate schemes [21], and claims integrated tamper resistance. Similarly to SPHINCS+, public keys are very short (32 B), but signatures are definitely longer: around 12.5 KB (signature size is not fixed with this scheme).

- `RAINBOW` [15] is another multivariate post-quantum signature scheme. It has a simple mathematical construction, which eases a smooth and correct implementation, and it is a quite old scheme (first proposed in 2005). This means that its security has been widely researched and has withstood the test of time. It targets computational efficiency and small signature sizes, but has large key sizes: signatures are only 64 B, but public keys are around 58 KB.

The selection has been made by analysing the current standards and the state-of-the-art of digital signature algorithms, while referring in particular to the ongoing NIST selection regarding post-quantum algorithms [32] (see also [27]). All post-quantum schemes in our list have advanced to the third round of the NIST selection, either as finalists or as alternate candidates. In defining our list we gave priority to low combined-length of public key and signature and

variety on the underlying security assumptions, to improve resiliency against future unexpected attacks.

Besides the smart contract that defines parameter sets, Quadrans requires that for each digital signature algorithm there also is a smart contract that allows to verify any signature computed with that algorithm, so that it can be used as a reference implementation (although not optimised since it is general purpose for all parameters) and as last resource to verify signatures for wallet implementations that do not natively support that algorithm.

3.2 Addresses

User addresses in Quadrans are strings of hexadecimal characters that encode some bytes derived from the description of the user's authorised keys and common name, plus some other checksum bytes that help avoiding mistakes and clerical errors[3]. This means that users may check the formal validity and coherence of an address even offline.

The exact syntax of addresses will be decided at a later stage of the blockchain development according to [29].

4 Nodes

The Quadrans Blockchain is structured into three types of intertwined chains: a SynchroChain, a MasterChain, and multiple ShardChains (See Section 5). The Quadrans network is composed by two sets of active nodes:

- MasterNodes, which filter incoming transactions and are in charge of achieving a global consensus, maintaining the MasterChain and updating the Global State of the architecture, moreover they are also responsible of maintaining synchronization by managing the SynchroChain.
- Miners, which work in parallel on the ShardChains, validating transactions and running Smart Contracts, and are in charge of achieving a local consensus.

4.1 MasterNodes

To become a MasterNode, a TokenHolder needs to

- possess enough QDTs;
- (participate and) win a PoS competition.

If a TokenHolder wins the PoS competition (possibly with the help of other TokenHolders) during epoch h, then it will be a MasterNode during epoch $h+2$. MasterNodes are in charge of:

[3] At this preliminary stage we define addresses as a simple string of 64 hexadecimal characters encoding the 256 bit digest of Keccak-256 (to achieve retro-compatibility with Ethereum [7]) computed with the encoding of a single public key as input. However we intend to further study the matter and improve and expand the algorithm for address computation to achieve the goals presented in this section in a future release.

- managing incoming transactions (they include the incoming transactions that pass a formal check into a Transaction Pool);
- achieving the Global Consensus:
  - validating Miners's work and constructing the MasterBlocks;
  - managing the Global State by aggregating the Local States;

4.1.1 SynchroNodes

The TokenHolders that participate in the PoS competition become SynchroNodes, i.e. players in the consensus protocol of the SynchroChain (see Section 6.2). Therefore SynchroNodes are in charge of:

- enforcing time limits on block creation, enabling the other consensus mechanisms;
- deciding the behaviour of the Quadrans Blockchain:
  - deciding the rules to be used to determine the blockchain parameters, in order to optimise the workload of Miners, achieve the best possible throughput, and maintain strong security;
  - collecting solutions of the PoW competitions from prospective Miners.

4.2 Miners

If a TokenHolder wins the PoW [25, 24] competition during epoch h, then it will be a Miner during epoch $h + 2$ (see Section 6.3.1).
Miners are in charge of managing ShardChains by:

- contacting MasterNodes to obtain the transaction pool;
- running Smart Contracts as specified by valid transactions;
- updating the Local State;
- creating ShardBlocks;
- reaching a local consensus on the ShardBlocks not yet confirmed by the MasterChain.

5 Chain Structure

The Quadrans Blockchain is a set of blockchains, where Miners are divided into shards to work simultaneously on the computation of Smart Contracts, and therefore achieve a large throughput of executed transactions [23]. The blockchains managed by Miners are called *ShardChains*. In addition, MasterNodes work on a higher-level chain, the *MasterChain*, collecting the states of ShardChains to achieve a global consensus. Finally, time is divided into slots of fixed length, and each slot is pre-assigned to a single Miner per ShardChain and to a single MasterNode. To coordinate the work on these chains and enhance safety, Quadrans has an extra chain, called *SynchroChain*, that marks the beginning of time-slots, time-stamps the other blocks, and defines the parameters of the ledger.

5.1 The SynchroChain

The consensus mechanisms that regulate the Quadrans blockchains rely heavily on time-related concepts, therefore it is necessary to coordinate and synchronize their execution. The SynchroChain acts as a Trusted Third Party that time-stamps the blocks on the other chains, and regulates the flow of time. In Quadrans time is discretized on two levels:

- *epochs* inside which parameters of the chains have fixed values, but these values can change from one epoch to the other;
- *time-slots* that regulate the production of blocks (one block per chain per time-slot). The number of time-slots in an epoch is a parameter, therefore different epochs can have a different number of time-slots.

The SynchroChain is controlled by SynchroNodes, which in epoch h are the TokenHolders which were candidates in the MasterNode election held during epoch $h - 3$.

At the end of each time-slot the SynchroNodes produce a *SynchroBlock* that timestamps (and partially validates) all the blocks created during that time-slot. Consequentially the publication of a SynchroBlock marks the start of the next time-slot.

The last SynchroBlock of an epoch is called *EpochBlock* and contains additional information that regulates the operation on the chains (see Section 5.1.2 for more details):

- epoch parameters, fixing their values to be used in a future epoch (some values are dictated for the next epoch, some for the one after, and some for an epoch even further into the future);
- the information regarding the PoW competitions that regulate the ShardChains.

SynchroBlocks are referenced by the ShardBlocks created in the following time-slot, so it is necessary that the consensus on the SynchroChain is reached immediately, hence a BFT algorithm [9] has been chosen, as described in Section 6.2.

Every SynchroBlock gives preliminary validation to the blocks of the other chains created in its time-slot (at most one MasterBlock and one ShardBlock per shard) by hash-linking them.

Linked to the EpochBlock there is a *Delta State*: information useful to reconstruct the current state. The knowledge of the state at the time of the previous EpochBlock and of the Delta State allows the correct reconstruction of the current state. This feature can be seen as the inclusion of check-points of the state, and it allows both fast verification of the correctness of the state and fast update of the state. Any new node can request only the Delta States of EpochBlocks and correctly obtain the current state.

5.1.1 SynchroBlocks

The structure of SynchroBlock Sy_i^h is shown in Table 1.

SynchroBlocks are accompanied by *certificates*, which are essentially a collection of signatures on the SynchroBlock by a set of SynchroNodes attesting

HEADER	
$H(\mathsf{Sy}^h_{i-1})$	hash-pointer to previous SynchroBlock
T_s	timestamp
R_i	root of the Merkle tree of DATA
DATA	
$H(\mathsf{Ma}^h_i)$	
$H(\mathsf{Sh}^h_{1,i})$	hash-pointers to the blocks created
$\vdots$	during ith time-slot of epoch h
$H(\mathsf{Sh}^h_{N_h,i})$	
EPOCH DATA	(only in EpochBlocks, i.e. if $i = e_h$)
`parameters`	value of parameters for following epochs
`PoW DATA`	information that regulates the consensus on the ShardChains

Table 1: SynchroBlock Sy^h_i created at the end of the ith time-slot of epoch h.

the authenticity of the SynchroBlocks. They are not included in the blocks since there may be different certificates, equally valid, on the same block. We refer to Section 6.2 for more details.

5.1.2 EpochBlocks

The EpochBlock E_h, published at the end of epoch h, besides the hash-links contained in every SynchroBlock, collects the PoW solutions computed during epoch h, submitted by prospective miners of epoch $h + 2$. Lastly, the EpochBlock E_h defines a series of parameters that regulate almost every aspect of the Quadrans blockchain. These parameters are:

- N_{h+3}, the number of ShardChains that will be active in epoch $h + 3$;
- e'_{h+4}, the target number of time-slots for epoch $h + 4$;
- $m_{\max,h+3}$, the maximum number of ShardBlocks that a Miner will be able to mine during epoch $h + 3$;
- $\texttt{param}_{h+1}$, the specification for the LSAG signature to be used to generate the public keys submitted with prospective Miners' enrollment transactions in epoch $h + 1$, and that will be used to submit PoW solutions in epoch $h + 3$ to become Miners during epoch $h + 5$;
- $\alpha_{\sigma,h+1}$, for $1 \leq \sigma \leq N_{h+1}$, the coefficient to be used to compute the weight of ShardBlocks during epoch $h + 1$;
- l_{h+1} and w_{h+1}, the parameters that regulate the finalizability of ShardBlocks of epoch $h + 1$;

- $n_{\max,h+4}$ and $n_{\min,h+4}$, the maximum and minimum number of MasterBlocks that a MasterNode might create in epoch $h+4$;
- l'_{h+1} and w'_{h+1}, the parameters that regulate the finalizability of MasterBlocks of epoch $h+1$;
- H, the hash function instance to be used to compute hash-links between blocks in epoch $h+1$;
- $\mathcal{H}$, the specification of the pseudorandom function to be used in the PoWs computed in the epoch $h+1$;
- the specifications of the PQ-secure digital signature algorithms that may be used to sign ShardBlocks and MasterBlocks of epoch $h+1$;
- the specifications of the unique-signature and aggregable-signature algorithms to be used in the consensus protocol of the SynchroChain during epoch $h+6$: prospective candidates will publish related public keys during epoch $h+1$;
- $n, S, K, \Omega, \Lambda, \lambda, \epsilon$, the parameters of the consensus protocol of the SynchroChain to be used during epoch $h+2$;
- the specifications of the algorithm that, in the consensus protocol of the SynchroChain during epoch $h+2$, assigns the maximum value of the attempt counter to each SynchroNode (in the unique signatures that select Active SynchroNodes) according to its stake;
- the specifications of the algorithm that assigns shards to competitors for the PoW competition that will be held in epoch $h+3$;
- the specifications of the shuffling algorithms that assign PoW solutions and PoS candidates to time-slots of epoch $h+4$;
- the specifications of the reward and refund mechanisms for the Miners working in epoch $h+5$ and the MasterNodes working in epoch $h+6$;
- the minimal amount of token a TokenHolder has to possess in order to submit an enrollment transaction during epoch $h+1$ to become Miner in epoch $h+5$, and the amount necessary to submit a candidacy transaction in epoch $h+1$ to become MasterNode in epoch $h+6$.

The specific values of these parameters are determined through deterministic processes that take in consideration the status of the Quadrans ecosystem as a whole (chains, network, activity, ...). These processes are described in dedicated Smart Contracts managed by the Quadrans Foundations, which regulates the addition of new versions of these processes according to various heuristics. Note that for some parameters, e.g. the hash function to use to link blocks, these decisional processes may simply be a constant function, i.e. a single value. So in this case the Quadrans Foundations specifies in the smart contract which values are suitable, and the SynchroNodes decide among these options.

Therefore the EpochBlocks do not contain the actual value of the parameters, but rather a link to the rule to be used to compute it, i.e. a reference to the Smart Contract and the desired version, chosen among the options proposed by the Quadrans Foundation and approved by the TokenHolders.

5.2 ShardChains

The ShardChains are parallel blockchains where the Miners actually execute transactions and smart contracts. To manage the scalability, the number of ShardChains per Epoch (and therefore the degree of parallelization) is flexible. During epoch h there are N_h parallel ShardChains (a parameter specified in the EpochBlocks), and the Epoch lasts e_h time-slots, so during each Epoch a total number of $N_h \cdot e_h$ blocks can be created.

The selection of the TokenHolders that become Miners and create ShardBlocks during Epoch h is made through a PoW competition held during Epoch $h-2$, whose results are included in the EpochBlock E_{h-2} (see Section 6.3.1). In this way, Miners know in advance the time-slots in which they have to be active to create the assigned blocks. Each Miner active in a ShardChain is in charge of the creation of a maximum of $m_{\max,h}$ blocks. At each Epoch, the number of Active Miners is therefore at least $\frac{N_h \cdot e_h}{m_{\max,h}}$.

Each ShardChain has the duty of managing transactions, that are divided into distinct pools according to their input address.

When Miners create blocks during their assigned time-slots, they update the local state, and then send a Delta State to the MasterNodes via an off-chain communication. The MasterNodes use this Delta State to update the Global State.

Remark 1. Miners have to be aware of the balance of each account whose address is managed by their own Shard (they can do this by keeping a copy of their ShardChain's local state), and have to check in the Global State for transactions coming from other Shards which may have changed the balance (they can do this by off-chain communication with MasterNodes).

5.2.1 ShardBlocks

The structure of ShardBlock S_{h,j_h} created by Miner m is shown in Table 2.

5.2.2 Transactions between shards

When a transaction `tx` is related to one shard, but its effect ends in a state update on another shard, `tx` has to be validated by both shards. This means that `tx` generates multiple sub-transactions, one for each shard involved into the state update. `tx` is placed in a "pending" status until all the subsequent sub-transactions are validated by the involved shards. This may require an iteration since sub-transactions could generate new sub-transactions recursively. Until all the iterations are concluded, all the involved transactions remain in "pending" status. At the end of this process all the transactions are placed together and sent to a Masternode to be inserted in the corresponding MasterBlock. However, at the end of each epoch, all pending transactions will be purged and have to be mined again. Therefore some iterations could not finish and even `tx` will be purged and it will have to be mined again.

5.3 The MasterChain

The MasterChain is a blockchain where the *MasterNodes* aggregate the computations of each ShardChain that have reached local consensus into a global state.

HEADER	
$H(\mathsf{Sh}^h_{\sigma,j})$	hash-pointer to ShardBlock $\mathsf{Sh}^h_{\sigma,j}$
$H(\mathsf{Ma}^h_s)$	hash-pointer to MasterBlock Ma^h_s
$H(\mathsf{Sy}^h_{i-1})$	hash-pointer to previous SynchroBlock
(x, K, c)	PoW data (see Section 6.3.1)
R_{h,j_h}	Merkle root of executed transactions
$H(\Sigma^h_{\sigma,i})$	digest of the local state
$\mathtt{sig}_m(\mathsf{Sh}^h_{\sigma,i})$	signature of the header by Miner m
DATA	
$\mathtt{tx}_{h,j_h,1}$ $\mathtt{tx}_{h,j_h,2}$ $\vdots$	list of executed transactions
LOCAL STATE	

Table 2: ShardBlock $\mathsf{Sh}^h_{\sigma,i}$ created by Miner m during the ith time-slot of Epoch h, on the shard σ. $\mathtt{sig}_m(\mathsf{Sh}^h_{\sigma,i})$ is the signature of the first five entries of the header of the block $\mathsf{Sh}^h_{\sigma,i}$. The signature is formally put inside the header, since hash-pointers are defined as the hash value of a header.

Since time is divided into Epochs of fixed length, during each Epoch h the MasterNodes create e_h blocks of the MasterChain. During its assigned time-slot, the active MasterNode creates a block of the MasterChain.

Each MasterBlock has the role of finalizing the State of the Quadrans Blockchain by putting together all local states updated by Miners during the creation of ShardBlocks. All information on the ShardStates is collected by MasterNodes via off-chain communication as soon as Miners create new ShardBlocks.

The consensus for the MasterChain is obtained by looking at the chain with larger weight (see Section 6.4.2)

5.3.1 MasterBlocks

The structure of MasterBlock M_t created by Miner α is shown in Table 3.

6 Consensus

We consider three levels of consensus:

- a local level, driven by Miners working on ShardChains;
- a global level, reached by MasterNodes working on the MasterChain;

HEADER	
$H(\mathsf{Ma}_s^h)$	hash-pointer to MasterBlock Ma_s^h
$H(\mathsf{Sy}_{i-1}^h)$	hash-pointer to previous SynchroBlock
$H(\mathsf{Sh}_{1,i_1}^h)$	
$\vdots$	hash-pointers to finalized ShardBlocks
$H(\mathsf{Sh}_{N_h,i_{N_h}}^h)$	
$H(\Sigma_i)$	digest of the global state
$\mathtt{sig}_\alpha(\mathsf{Ma}_i^h)$	signature of the header by Miner α
GLOBAL STATE	

Table 3: MasterBlock Ma_i^h created by MasterNode α during the ith time-slot of epoch h. The signature is formally put inside the header, since hash-pointers are defined as the hash value of a header.

- a coordination level, reached by SynchroNodes through the SynchroChain.

The levels are intertwined: MasterNodes rely on the local consensus reached by Miners to achieve the global consensus, Miners look at the choices made by MasterNodes to safely reach a consensus on their ShardChain, and everyone relies on the parameters and time-stamps established by the SynchroChain.
The first two levels of consensus are reached using a weight-based rule, thus the heaviest chain dictates the established state, while the SynchroChain needs immediate finality, so is governed by a Byzantine Fault Tolerant consensus mechanism.

Each MasterBlock and ShardBlock is created by a single TokenHolder (a MasterNode and a Miner respectively), and afterwards is validated (or discarded) by subsequent Blocks, that may endorse it (alongside every block it endorses, in a recursive manner) by hash-linking it. Honest TokenHolders consider valid, and thus may endorse, only blocks that meet a series of criteria:

- *timing*: the block must have been created and broadcast inside the limits of the time-slot;
- *authorization*: the blocks on the ShardChains and the MasterChain in a given time-slot have to be created by specific TokenHolders, thus the creator has to sign the block and include a proof that it actually is authorized to create the block;
- *formal correctness*: the block must respect the format dictated by the chain it belongs to and the values of the parameters in the current epoch, moreover the signatures included in the block must be verified, and the hash-links must be *admissible* (see Sections 6.3.7 and 6.4.2);

- *semantic correctness*: the block correctly processes the data pertaining to its chain, and updates the state accordingly.

6.1 Definitions

We state here some definitions that are necessary to define the validity and finality of a block (both on a ShardChain and the MasterChain).

Definition 2 (Ancestor and Descendant)**.** Let B_i and B_j be two blocks on the same chain, with $i \leq j$ (with this we imply that B_i has not been created after B_j). Then B_i is an *ancestor* of B_j and B_j is a *descendant* of B_i, if there is a sequence of blocks (of the same chain) $\{B_{k_l}\}_{l\in\{0,\dots,n\}}$ with $k_l \leq k_{l'}$ if $l \leq l'$, such that $i = k_0, j = k_n$ and B_{k_l} hash-links $B_{k_{l-1}}$ for $l \in \{1,\dots,n\}$. If $n = 1$ (and $i < j$) B_j is an *immediate descendant* of B_i.

Note that by this definition a block is an ancestor and a (non-immediate) descendant of itself.

Definition 3 (Simple Line)**.** A *simple line* is a connected directed graph where one node has no inbound edges, one node has no outbound edges and every other node has exactly one inbound and one outbound edge.

Definition 4 (Unresolved Branch)**.** An *unresolved branch* is a simple line of blocks (i.e. a graph where the blocks are the nodes, the internal hash-links are the edges) that starts from an immediate descendant of a finalized block and ends in a leaf (i.e. a block such that there is not a more recent block that hash-links it).

Paraphrasing the definition, given a tree of blocks rooted in a finalized block, the unresolved branches are the paths from the root (excluded) to the leaves.

Definition 5 (Seniority of Unresolved Branches)**.** Given Γ_1 and Γ_2, two unresolved branches of the same chain as per Definition 4, then Γ_1 is older than Γ_2 if:

$$\min\left(i : B_i \in \Gamma_1, B_i \notin \Gamma_2\right) < \min\left(i : B_i \in \Gamma_2, B_i \notin \Gamma_1\right), \tag{1}$$

where we continue assuming that B_i has been created before B_j if and only if $i < j$.

6.1.1 Blocks Finalization Assumptions

As we will see, the consensus on the various chains proceeds with different rules and different paces, so blocks may become *final* (i.e. definitively accepted as correct and part of the blockchain by all honest Nodes) with different speeds, depending also on possible fragmentations or attacks.

However, an upper bound is necessary to analyze security and correctness of protocols, so we will assume that, at the end of Epoch $h+1$, the blocks created during Epoch h which are not finalized will never be finalized, and therefore can be discarded.

6.2 SynchroChain Consensus

Every SynchroBlock gives preliminary validation to the blocks of the other chains created in the same time-slot (at most one MasterBlock and one ShardBlock per shard) by hash-linking them. This validation is a necessary but not sufficient condition for any block to be considered valid and be included in the final blockchain (that prunes the blocks excluded by the consensus), since it only concerns timing. That is, the network only checks that the block has been broadcast inside the limits of the time-slot.

The SynchroBlocks are created via a Cob Protocol run [17, 18, 19], a leaderless Byzantine Fault Tolerant protocol which allows a network to reach agreement on a vector of time-stamps of a set of events happened in a given time interval. Each event corresponds to a specific component of the vector and the consensus process is carried out in parallel on each component. The nodes that manage this consensus are called SynchroNodes, and each of this nodes halts the protocol execution only when it gets hold of a *certificate* for the new block. The protocol guarantees that such a certificate is created within $S + K$ steps, and that only one block may be certified in each run.

For the SynchroBlock consensus process, the events to be recorded are the MasterBlock and ShardBlocks broadcast within the prescribed time-slot, and the vector of time-stamps is a vector containing the hash digests of such blocks. Therefore the network must reach consensus on which digests to accept, declaring that the corresponding blocks have been created and broadcast in the prescribed time.

For the EpochBlock consensus process, the events to be recorded are expanded to include also the parameter list described in Section 5.1.2, and the solutions to the Proof of Work competition performed and submitted during the current epoch by the Miners.

The agreement process is performed in parallel on each component of the vector and, if agreement on some component is not reached after a number S of steps, such component is discarded. For the SynchroBlock this means that the corresponding block will not be considered to be valid and discarded as if it was not created by the Miner (or MasterNode) in charge. However, the Cob Protocol guarantees that, if the block creator is honest and broadcasts its block in time, the agreement will be reached on the corresponding component at the beginning of the protocol run and it will not be discarded during the protocol execution. Therefore the hash of the block will be included in the corresponding SynchroBlock and will be eligible to be included in the final blockchain.

6.2.1 SynchroNodes Selection

During epoch h, the Cob protocol that determines the consensus on the SynchroBlocks Sy_i^h (and on the EpochBlock E_h) is run by the TokenHolders that during epoch $h - 5$ submitted a candidacy to become a MasterNode during epoch h. Note that this set includes both the MasterNodes of epoch h and those who were candidates but ultimately did not accumulate enough stake to become MasterNode. These TokenHolders are called SynchroNodes and are all expected to follow the protocol execution, which is divided in steps.

Every step essentially consists in collecting and propagating data from and to the network for a specified amount of time and then compute a message

based on this data and diffuse it on the network. Every SynchroNode should help propagating messages (so maximum diffusion is achieved) and update its internal state, but at each step only a subset of the SynchroNodes are *Active* and can compute and broadcast new messages at the end of the step.

The selection of which SynchroNodes will be Active in each step is made via a *verifiable random function*, and the SynchroNodes will accept a message only once they can verify that the sender was authorised. This peculiar function is a *unique-signature* algorithm, which has only one valid output on a fixed input, and this output (called signature) satisfies a certain condition with known probability. Specifically the condition can be parametrized in order to adjust the probability with which the signature satisfies it. We call a signature for a step that satisfies the condition a *winning signature* for that step.

The input to be signed is essentially the concatenation of a time-slot counter, a protocol step counter and an attempt counter, so a SynchroNode is selected to be Active in a step if it can produce a winning signature where the attempt counter is below a threshold. This limit is proportional to the total stake accumulated by the SynchroNode during the PoS competition, so with enough stake a SynchroNode may be selected and thus act as an Active SynchroNode multiple times. Basically, each SynchroNode is selected with probability proportional to its stake. The parameters of the signature condition are calibrated so that at every step there is an expected number n of Active SynchroNodes.

The signature can be computed only with a private key which is kept by the SynchroNode, but it can be checked by anyone with a corresponding public key, which is published by the TokenHolder contextually with its candidacy. This means that simply by attaching a winning signature to a protocol message the SynchroNode demonstrates that it is an Active SynchroNode and has the right to broadcast the message.

Note also that the signatures can be computed in advance, so, when the Cob protocol parameters to be used during epoch h are published at the end of epoch $h-2$, each SynchroNode can check whether or not it will be active during each possible step of each time-slot of epoch h, and prepare in advance.

6.2.2 SynchroBlock Creation and Certificates

At every protocol run only one SynchroBlock can be certified but two distinct nodes might be in possession of two different certificates for the same SynchroBlock. In fact a certificate consists of a set of t_H messages from each of two consecutive steps of the Cob protocol (actually a Coin-Fixed-To-0 and a Coin-Fixed-To-1 step) if it is produced within S steps, otherwise the certificate will be produced at the end of STEP $S+K$ and will consists of t_H messages from such step. The set of messages which compose the certificate must support the same vector.

In order to improve the efficiency of SynchroBlock propagation, the signature algorithm used to certify Cob protocol messages are computed with an *aggregable-signature* algorithm. This allows to aggregate multiple signatures of the same data into a single signature that is verifiable against a public key that is derived from the original public keys corresponding to the individual signatures. This enables the compression of certificates into:

- a bit-string that encodes which SynchroNodes signed the certificate: they are ordered according to their stake, and for each SynchroNode there

are as many bits as the maximum value of the attempt counter for this SynchroNonde (which is proportional to the stake), then each bit signals whether the corresponding SynchroNode (and attempt counter value) signed the certificate or not;

- the list of the unique-signatures that certify that the SynchroNode was indeed Active (in the same order as before so they can easily be checked against the corresponding public keys, which are published contextually with the candidacy of the SynchroNodes);
- the aggregated signature, that can be checked computing the corresponding public key using the individual keys indicated by the bit-string.

6.2.3 EpochBlocks

At the end of epoch h, the SynchroNodes must reach agreement on the EpochBlock E_h. The EpochBlocks, besides the hash-links contained in every SynchroBlock, define the parameters listed in Section 5.1.2, and contain the list of PoW solutions computed during epoch h, submitted by prospective miners of epoch $h+2$.

The consensus process is carried out by the SynchroNodes as for the SynchroBlocks, just with a different vector of values to agree on. This vector is built in the following way:

- the component 0 refers to the newly created MasterBlock;
- the subsequent N_h components $(1, \ldots, N_h)$ refer to the newly created ShardBlocks;
- the subsequent 28 components are the parameters that must be defined in the EpochBlock (see Section 5.1.2);
- the other components contain the PoW solution: every legitimate solution s is inserted in the component $\mathtt{H}(s)$ seen as the binary representation of a natural number.

The output of the function $\mathtt{H}$ is the digest of the hash function H (the same used to hash-link blocks) truncated to the c-th leftmost bit. Note that in the EpochBlock E_h there may be at most $m_{\max,h+2}M_{h-2}$ submitted solutions, where M_{h-2} is the number of TokenHolders that enrolled in the PoW competition of epoch h submitting an enrolling transaction during epoch $h-2$. Since H is collision-resistant, assuming that $m_{\max,h+2}M_{h-2} \ll 2^{\frac{c}{2}}$ we can approximate the probability of a collision as:

$$\frac{m_{\max,h+2}M_{h-2}(m_{\max,h+2}M_{h-2}-1)}{2^{c+1}} \tag{2}$$

so, the parameter c is determined in such a way that the collision probability is less than the negligible threshold ϵ, a parameter of the Cob protocol.

Note that a vector computed in such a way may be extremely large in dimension, but also very sparse, so we will adopt a representation which allows us to omit the blank components.

The messages exchanged during a Cob protocol run contain a vector of values in the first 2 steps and a vector of bits in the following steps; in any case the

portion of the vector v that corresponds to the PoW solutions must be encoded as

$$((v_{c,1}, c_1), (v_{c_2}, c_2), \dots (v_{c_k}, c_k))$$

where $c_i \in \{0, 2^c\}$ and v_{c_i} is not a blank value.

We remark that the vector components corresponding to the parameter list will likely reach consensus in the very first steps of the Cob protocol, and therefore not discarded in the final EpochBlock. In fact these components identify which rules have to be used to compute the parameters from known values, and these rules are expected to remain quite stable and be changed only once there is wide consensus (likely established through off-chain discussions amongst TokenHolders). As a consequence, all honest SynchroNodes are expected to begin the Cob protocol with the same value in these components, which assures rapid finalization.

However, if agreement can not be reached on one of these parameter rules, and therefore the corresponding component becomes blank in the EpochBlock, then the rule to be used is the one established in the most recent EpochBlock which is non blank in the corresponding component, promoting rule stability.

6.3 ShardChains Consensus

The consensus on the ShardChains is reached through PoW-based mechanisms, that proceed independently on each shard.

6.3.1 Proof of Work for Miners in ShardChains

From now on we will call *Competitors* the TokenHolders who want to become Miners, when needed we will divide them between *MiningCompetitors* (the Competitors who are already active Miners during the epoch in which they compete) and *SimpleCompetitors* (the ones who are not Miners in the same epoch).

During each epoch, TokenHolders may compete in the PoW competition. The results of the competition are listed on the next EpochBlock, and are used to determine the Miners for the subsequent epoch. In order to avoid attacks aimed to the Miner of a specific time-slot, its identity must be kept private until it publishes the block. In particular, no one should know which TokenHolder is going to mine a new block in a specific time-slot in the future, but also each miner must be able to prove that, according to the data contained in the EpochBlock, it was the one selected by the competition. To accomplish this goal *Competitors* must produce some commitments that guarantee that they own the solutions listed in the EpochBlock and that they have found the solutions in a specified time-slot.

6.3.2 Enrollment Transaction and Shard Assignment

In order to become a Competitor, a TokenHolder must possess enough QDTs: to become miners in epoch $h + 2$ a TokenHolder must submit a transaction of $\texttt{Mi}_{\min}$ QDTs during epoch $h-2$. From now on we refer to this transaction as the *enrollment transaction*. This transaction will allow the TokenHolder to submit solutions to the PoW competition that selects the Competitors who will become Miners.

Let $m_{\max,h+2}$ be the maximum number of blocks a Miner can create in the epoch $h+2$ (a parameter defined in the EpochBlock E_{h-1}), with its enrollment transaction a Competitor communicates $m_{\max,h+2}$ distinct *one-time LSAGSS*[4] *public keys* that will be used to anonymously authenticate solution submissions, i.e. with these keys the Competitor will be able to demonstrate that the fee has been paid, without revealing the TokenHolder's Identity. These public keys are in the form:

$$P_i = \text{PKgen}(x_i, \texttt{param}_h), \quad i = 1, \ldots, m_{\max,h+2} \tag{3}$$

where $\texttt{param}_h$ is the set of parameters for the LSAG signature scheme to be used to submit the solutions of the PoW competitions in epoch h, and PKgen is the function that outputs the public key corresponding to a private key and a set of parameters.

Example 2. For example $\texttt{param}'$ may define an elliptic curve with base point $\mathcal{B}$ and a hash function that maps an EC point into another EC point (that will be used to compute the signature), in this case $\text{PKgen}(x, \texttt{param}') = x\mathcal{B}$.

As a response to these transactions, the SynchroNodes will assign to each Competitor a shard on which they will mine if they win the competition. The steps are the following:

1. the SynchroNodes collect all the transactions coming from TokenHolders who want to become Competitors;
2. the SynchroNodes agree on a division of the Competitors among the shards:
   (a) the SynchroNodes estimate the computational power of each Competitor counting the number of blocks it mined in the latest N epochs;
   (b) the SynchroNodes divide the Competitors in brackets of similar estimated computational power;
   (c) to each shard is assigned a number of Competitors for each bracket with the goals of distributing the estimated computational power and the members of each bracket as evenly as possible among all shards;
   (d) the actual Competitors are chosen randomly from the members of the bracket, so that no one can predict to which shard a Competitor will be assigned;
3. at the end of epoch $h-1$, the Competitors can compute which shard is assigned to them using the specifications published in the EpochBlock E_{h-1}.

6.3.3 Simple Competition and Mining Competition

The competition takes place during Epoch h, and it is slightly different for MiningCompetitors and SimpleCompetitors. Let e_h be the number of time-slots in epoch h, Sy_i^h be the i-th SynchroBlock of epoch h. The competition proceeds as follows:

[4] Linkable Spontaneous Anonymous Group Signature Scheme

- the SimpleCompetitors wait until the creation of $\mathsf{Sy}^h_{e_h-2}$ and they compete all together during the penultimate time-slot of epoch h, and can submit a single solution to the crypto-puzzle (see Section 6.3.4);
- the MiningCompetitors instead can submit up to $m+1$ solutions, where m is the number of time-slots assigned to them during epoch h. In fact they can submit a solution for each block they create (competing individually during the time-slot preceding the one in which they have to create the block) and, if $m < m_{\max,h+2}$, they can also compete as SimpleCompetitors. The extra solutions are tied to the blocks the MinerCompetitor creates, i.e. they are considered only if the blocks are valid and included in the final ShardChain.

The competition is designed to limit the waste of computational resources and peaks of consumption. Moreover the MiningCompetitors' additional competition rewards effective and honest Miners with more chances of remaining a Miner, and the potential of receiving incrementally more slots, with a (capped) positive feedback mechanism.

6.3.4 Crypto-Puzzle

When a Competitor partakes in a PoW competition, it tries to find the best possible solution to a crypto-puzzle. Specifically, it tries to find a string `nonce` such that the Hamming distance between a given string `target` and the digest $\mathcal{H}(\texttt{target}'\|\texttt{nonce})$ is as small as possible, where $\texttt{target}'$ is related to `target`, and $\mathcal{H}$ is a computationally-expensive pseudo-random function. The strings `target` and $\texttt{target}'$ are tied to three elements:

- the SynchroBlock Sy^h_j that marks the start of the time-slot in which the competition is supposed to take place, to prevent jump starts;
- the secret key x corresponding to one of the one-time LSAGSS public keys communicated with the enrollment transaction, to tie the competition to the competitor (preventing the sharing of solutions), and verify the correctness of the solution submission;
- a random string K chosen by the competitor, to mask the targets and avoid de-anonymization of solutions via brute-force.

Let m be the number of time-slots of epoch h assigned to a Competitor, then the value of these target strings is defined slightly differently for SimpleCompetitors (for which $m = 0$), general MiningCompetitors, and special corner cases of MiningCompetitors:

- for SimpleCompetitors that compete in the penultimate time-slot of epoch h the targets are defined as:

$$\texttt{target}' = \texttt{target} = \mathcal{H}(\mathsf{Sy}^h_{e_h-2}\|x_{m+1}\|K); \tag{4}$$

- for a MiningCompetitor that is scheduled to produce m ShardBlocks in non-consecutive time-slots $\{j_1, \dots j_m\}$ (i.e. $|j_i - j_{i'}| > 1\ \forall i, i' \in \{1, \dots, m\}$)

the m competitions are independent and performed during the time-slot $j_i - 1$ with targets:

$$\texttt{target}'_i = \texttt{target}_i = \mathcal{H}(\mathsf{Sy}^h_{j_i-2} \| x_i \| K_i), \tag{5}$$

for $i \in \{1, \ldots, m\}$, where each K_i is chosen uniformly at random;

- for a MiningCompetitor that is scheduled to produce $v \geq 2$ ShardBlocks in consecutive time-slots $\{j, \ldots, j + v - 1\}$, to avoid a split of resources between the PoW and smart-contract/transaction execution the competition takes place only during time-slot $j - 1$, but produces v solutions. For $i \in \{1, \ldots, v\}$, the competitor chooses uniformly at random K_i and sets:

$$\texttt{target}_i = \mathcal{H}(\mathsf{Sy}^h_{j-2} \| x_i \| K_i), \tag{6}$$

and:

$$\texttt{target}'_i = \texttt{target}' = \texttt{target}_1 \| \ldots \| \texttt{target}_v. \tag{7}$$

Note that with these definitions, given a candidate solution `nonce`, the value $\mathcal{H}(\texttt{target}' \| \texttt{nonce})$ can be computed just once and then compared to the v targets $\texttt{target}_1, \ldots, \texttt{target}_v$ to see if some of the distances are good enough. Given that the cost of computing a distance is almost negligible compared to the cost of evaluating $\mathcal{H}$, this approach allows to compress v competitions in a single time-slot, and the small disadvantage may be seen as a compensation for the reduced computational cost. In any case the competitors cannot directly influence slot assignation, and the probability of having v consecutive slots is low and decreases rapidly as v grows.

- for a MiningCompetitor that is scheduled to produce the ShardBlocks in the time-slot $e_h - 1$ (the same time-slot when the Simple Competition takes place) or in the time-slot e_h (and therefore its Mining Competition should take place in the same time-slot as the Simple Competition), there is a special case to improve fairness. If this Competitor can compete also in the Simple Competition, then the two competitions are joined and take place during the same time-slot j of the competition for the block $e_h - 1$ (i.e. $j = e_h - 2$ if the very same MiningCompetitor is not the Miner for the time-slot $e_h - 2$) or e_h, and proceeds with the same method of the competition for multiple consecutive time-slots, counting the Simple Competition as an extra consecutive time-slot. Therefore it selects a further random string K^*, computes the extra target $\texttt{target}^* = \mathcal{H}(\mathsf{Sy}^h_j \| x_{m+1} \| K^*)$, and modifies $\texttt{target}' := \texttt{target}' \| \texttt{target}^*$ (note that this value of $\texttt{target}'$ is used both for the Mining and Simple Competition).

Of course if a MiningCompetitor has both consecutive and non-consecutive time-slots, it partakes in each PoW competition with the appropriate rules, ordering the competitions with the chronological order of the time-slots they correspond to (i.e. in the i-th competition the Competitor uses the private key x_i to compute $\texttt{target}_i$ and $\texttt{target}'_i$).

6.3.5 Solution Submission and Commitment

The solutions found by a competitor have to be sent to the SynchroNodes in the last time-slot of the epoch to be included in the ranking. A solution is sent anonymously as the tuple:

$$s = (\sigma, d, \texttt{nonce}, \rho), \tag{8}$$

where σ is the shard assigned to the Competitor, d is the achieved distance, and ρ is a LSAGSS signature on $\sigma \| d \| \texttt{nonce}$. The signature ρ is computed with the secret key x associated to the solution (i.e. used to compute the targets) and a group of one-time public keys chosen randomly from those communicated by the competitors assigned to the same shard σ (these public keys can be found in the the enrollment transactions submitted during epoch $h-2$), obviously including the public key P corresponding to x. Note that ρ includes an identification value $I = \text{IDgen}(x, \texttt{param}_h)$ that exposes multiple usage in signatures of the same key, thus assuring that only one solution may be submitted with each public key communicated with the enrollment transaction, but does not reveal which public key of the group has actually signed the submission, maintaining anonymity.

Example 3. Reprising Example 2, let $\mathcal{B}$ and $\widetilde{H}$ be respectively the base point and the hash function defined by $\texttt{param}'$, then $\text{IDgen}(x, \texttt{param}') = x\widetilde{H}(x\mathcal{B})$.

It is important to notice that, even if a TokenHolder could theoretically send all solutions at once, it is advisable to send them separately, in order to preserve the privacy. For the same reason MiningCompetitors should not send their solutions right after they have computed them.

The PoW competitions are supposed to last for the duration of one time-slot. For the Simple Competition this is given from the fact that the targets cannot be computed until the SynchroBlock $\mathsf{Sy}^h_{e_h-2}$ has been published marking the start of time-slot $e_h - 1$, and the solutions must be communicated in time to be included in the EpochBlock E_h that is computed during time-slot e_h. For the Mining Competition this property is achieved by having the MiningCompetitors to include in the ShardBlock they create in the time-slot j a commitment to the solution found in the competition that took place during the time-slot $j-1$:

- if the solution is single (i.e. for the case of non-consecutive time-slots) the commitment is:
$$c = \mathcal{H}(\texttt{nonce} \| K), \tag{9}$$
where K is the same used to compute the target;
- if the MiningCompetitor is scheduled to produce v consecutive ShardBlocks starting from the time-slot j, then it has computed v solutions during the time-slot $j-1$, so it computes the commitments as:
$$c_i = \begin{cases} \mathcal{H}(\texttt{nonce}_{v-1} \| K_{v-1}) \| \mathcal{H}(\texttt{nonce}_v \| K_v) & i = v \\ \mathcal{H}(\texttt{nonce}_{i-1} \| K_{i-1}) \| \mathcal{H}(c_{i+1}) & 1 < i < v \\ \mathcal{H}(c_2) & i = 1 \end{cases} \tag{10}$$
and includes the commitment c_i in the block created in the time-slot $j + i - 1$. Note that c_1 references all of the solutions, but has the same

format (i.e. single hash digest) of a single commitment, this means that all solutions must be computed before c_1, but it is not disclosed that the following ShardBlock will be produced by the same Miner. Note also that successive commitments have the same format (i.e. concatenation of two hash digests), so it is not disclosed whether the sequence of consecutive time-slots assigned to the same Miner is over or not. Finally note that from these commitments it is possible to extract all of the single commitments as in Eq. (9), but the i-th solution is included in the $(i+1)$-th block (except for the last one), so the $(i+1)$-th block must be valid in order to the i-th solution to be accepted.

- a MiningCompetitor assigned to the time-slot j, with $j = e_h - 1$ or $j = e_h$, has to include the commitment c^* to the extra solution that computes instead of competing in the Simple Competition. This extra commitment c^* is simply appended to the regular commitment included in the block j (computed as explained above), and is computed as follows:
  - if the MiningCompetitor has already (counting the commitment in the block j) committed to the maximum number of solutions allowed ($m_{\max,h+2}$) then c^* is simply the empty string;
  - if $j = e_h - 1$ and the MiningCompetitor is not entitled to partake in the Simple Competition because it will make the $m_{\max,h+2}$-th commitment in the following block, then $c^* = \mathcal{H}(K)$ where K is the same random string used in the target (and commitment) of the solution included in the block $e_h - 1$;
  - if the MiningCompetitor is entitled to partake in the Simple Competition then $c^* = \mathcal{H}(\texttt{nonce}^*, K^*)$ as for single solutions.

In the EpochBlock E_{h-1}, published at the end of time-slot e_{h-1} of epoch $h-1$, SynchroNodes fix the number N_{h+2} of shards in epoch $h+2$, while in the EpochBlock E_{h-2}, they fixed e'_{h+2}, a target value for e_{h+2}, that is the number of time-slots in epoch $h+2$ (this value is finalized at the end of epoch h, given the results of the PoS competition, see Section 6.4.1). Having received all the valid solution submissions, the SynchroNodes publish in the EpochBlock E_h a table with the e_{h+2} best submissions for every shard (overall $N_{h+2} \cdot e_{h+2}$ submissions are chosen). Fixed a shard σ, a solution $s_1 = (\sigma, d_1, \texttt{nonce}_1, \rho_1)$ is better (and thus ranked higher) than another solution $s_2 = (\sigma, d_2, \texttt{nonce}_2, \rho_2)$ if:

$$\begin{aligned} & (d_1 < d_2) \\ \vee\ & ((d_1 = d_2) \wedge (\texttt{nonce}_1 < \texttt{nonce}_2)) \\ \vee\ & ((d_1 = d_2) \wedge (\texttt{nonce}_1 = \texttt{nonce}_2) \wedge (\rho_1 < \rho_2)) \end{aligned} \tag{11}$$

where strings are compared using the lexicographic order. Each solution included in the table gives the right to create a ShardBlock in the corresponding shard during the epoch $h+2$. The actual time-slot in which this block must be created is determined by the output of a pseudo-random function (specified in EpochBlock E_{h-2}) with an input that depends on E_h, effectively shuffling the order.

For every shard, the SynchroNodes that create the EpochBlock are rewarded proportionally to the goodness of the worst solution that gets a block in that

shard, thus SynchroNodes are incentivized to include the best solutions available.

6.3.6 Time-Slot Claiming and Enrollment Refunds

The Competitors that have one or more solutions included in E_h become Miners in epoch $h+2$. Note that Competitors can verify if one of their solutions has won a block, but from a solution it is impossible to tell who submitted it. Miners know well in advance when they will have to produce a ShardBlock, so they can prepare themselves.

When a Miner produces a ShardBlock, it must include in the block a proof that it was indeed the Miner that submitted the solution that was assigned to that time-slot in that shard. This proof is the pair:

$$(x, K), \tag{12}$$

where x and K are the same ones used to compute the targets of the solution.

The LSAGSS one-time private key x is used to compute the corresponding public key $P = \text{PKgen}(x, \texttt{param})$, to check that the Miner paid an enrollment transaction that announced P and was assigned to the shard σ, and finally to check that the value $I = \text{IDgen}(x, \texttt{param})$ of the LSAGSS signature ρ used to authenticate the solution is correct.

Then it is checked whether the Miner produced ShardBlocks in epoch h, and using x, K, and the appropriate Sy_j^h (see Section 6.3.4 for the various cases) it is possible to compute $\texttt{target}$ and $\texttt{target}'$. Note that Miners that were MiningCompetitors have multiple candidates for Sy_j^h, so they are encouraged to communicate off-chain which is the correct one, even if it is possible to find it by trial and error. The targets' values are then used to check that the distance d included in the solution is correct. Finally, K is used to check the correctness of the commitment, comparing the value of $\mathcal{H}(\texttt{nonce} \| K)$ to the one included in the prescribed ShardBlock. Note that this check is not executed for most solutions of the Simple Competition (i.e. those not computed by the miners active in the last two time-slots of epoch h), since they have not been committed in a block, but for these solutions it is checked that the Competitor was allowed to submit that solution in the Simple Competition. In particular it is checked that the Competitor did not produce the block $\mathsf{Sh}^h_{\sigma,e_h-1}$ or the block $\mathsf{Sh}^h_{\sigma,e_h}$, and that no other solution for the simple competition has already been claimed by the same Competitor in the epoch $h+2$.

In the EpochBlock E_{h-4} are specified the refund mechanisms: starting from the epoch $h+3$ the Competitors are refunded of the QDTs they paid with their enrollment transactions. These refunds are carried on considering separately each fraction of $\frac{\texttt{Mi}_{\min}}{m_{\max,h+2}}$ QDTs corresponding to one individual public key announced with the enrollment transaction:

- the keys whose private counterpart did not produce a winning solution (i.e. $P = \text{PKgen}(x, \texttt{param})$ such that x has not been used to sign the submission of a solution included in E_h) are refunded after the Competitor discloses x, so it is possible to check that indeed that key was not effective (checking that $P = \text{PKgen}(x, \texttt{param})$ and that $I = \text{IDgen}(x, \texttt{param})$ is not present in E_h);

- the keys whose private counterpart did produce a winning solution and the corresponding ShardBlock has been correctly created and finalized, can be refunded after the block becomes final (i.e. it is considered valid and supposed to remain valid in the future, see Section 6.3.7). Note that the corresponding private key x is disclosed in this case too;
- the keys whose private counterpart did produce a winning solution but the corresponding ShardBlock has not been created or finalized, can be refunded not before epoch $h + 4$ if the Competitor discloses x, so it is possible to check that the submission was valid and that the Competitor was entitled to submit that solution.

The refund policy incentivizes honest behaviour in the submission of solutions, where Competitors could potentially lie about the distance achieved d to be included among the winners. In this case, however, they could not actually claim the time-slot and create the ShardBlock since they cannot provide a correct proof. This would create a gap in the block creation and deprive an honest Competitor of its chance to become Miner, so this malicious behaviour may be punished with the seizure of part of the QDTs pawned with the enrollment transaction.

6.3.7 ShardBlock Validity and Finality

A ShardBlock is considered valid if:

1. it is hash-linked by a SynchroBlock (this implies timing validation)
2. its Miner is authorized;
3. it is formally correct (this implies that the hash-links are admissible);
4. it is semantically correct (the transactions included are valid and properly accounted for, the smart-contract executions are correct, the hash-links point to valid blocks);
5. it is finalized by the MasterChain or included in the oldest unresolved branch (see Definition 5).

A ShardBlock is *finalized* by the MasterChain when there is a MasterBlock that hash-links it or one of its descendants and this MasterBlock is final (see Section 6.4.2).

Let Ma_i^h be the MasterBlock created in the i-th time-slot of epoch h, and $\mathsf{Sh}_{\sigma,j}^h$ the ShardBlock on the ShardChain σ created in the j-th time-slot of epoch h.

A ShardBlock $\mathsf{Sh}_{\sigma,i}^h$ should hash-link the most recent valid ShardBlock on the same shard and the most recent final MasterBlock (see Section 6.4.2), so its hash-links are admissible if:

- there are exactly one hash-link to a ShardBlock on the same shard, one to a MasterBlock, one to the previous SynchroBlock, and no other hash-links;
- let $\mathsf{Sh}_{\sigma,i'}^{h'}$ be the ShardBlock hash-linked by $\mathsf{Sh}_{\sigma,i}^h$, then:

$$(h' < h) \vee ((h' = h) \wedge (i' < i)) \,; \tag{13}$$

- let $\mathsf{Ma}_{i^*}^{h^*}$ be the MasterBlock hash-linked by $\mathsf{Sh}_{\sigma,i}^{h}$, then:

$$(h^* < h) \vee ((h^* = h) \wedge (i^* < i))\,; \tag{14}$$

- let $\mathsf{Sh}_{\sigma,\hat{\imath}}^{\hat{h}}$ be the ShardBlock on the shard σ hash-linked by $\mathsf{Ma}_{i^*}^{h^*}$, then $\mathsf{Sh}_{\sigma,\hat{\imath}}^{\hat{h}}$ is an ancestor of $\mathsf{Sh}_{\sigma,i'}^{h'}$;

- let $\mathsf{Ma}_{\tilde{\imath}}^{\tilde{h}}$ be the MasterBlock hash-linked by $\mathsf{Sh}_{\sigma,i'}^{h'}$, then $\mathsf{Ma}_{\tilde{\imath}}^{\tilde{h}}$ is an ancestor of $\mathsf{Ma}_{i^*}^{h^*}$;

The weight of a ShardBlock is defined as:

$$w(\mathsf{Sh}_{\sigma,j}^{h}) = \frac{\alpha_{\sigma,h}}{d} \sum_{f \in \mathsf{Sh}_{\sigma,j}^{h}} f, \tag{15}$$

where $\alpha_{\sigma,h}$ is a parameter defined in E_{h-1} (chosen to keep the weight of blocks as stable as possible), f represents the fee of a transaction contained in the block $\mathsf{Sh}_{\sigma,j}^{h}$ and d is the distance from the target of the solution that allowed the Miner to produce that ShardBlock.

The EpochBlock E_{h-1} specifies the value of two parameters l_h and w_h that regulate whether a ShardBlock $\mathsf{Sh}_{\sigma,j}^{h}$ is finalizable by a MasterBlock:

- l_h ensures that the ShardBlock is sufficiently old, in fact it must have been created at least l_h time-slots before the MasterBlock;

- w_h ensures that disputes in a branch are resolved, in fact the ShardBlock must belong to an unresolved branch that is sufficiently heavier than all other unresolved branches in the shard:

$$\mathsf{Sh}_{\sigma,j}^{h} \in \Gamma^* : w(\Gamma^*) > w_h + w(\Gamma) = w_h + \sum_{B \in \Gamma} w(B) \quad \forall \Gamma \in \sigma, \Gamma \neq \Gamma^* \tag{16}$$

The values of l_h and w_h are chosen in a way that allows the honest Miners, in case of attack, to overcome the attacker's branch and not to be afraid to hash-link the correct block on the older branch, even if that branch is shorter.

Once the consensus between Miners is reached, the MasterNodes can make up the time that might have been lost and finalize more than one ShardBlock at once by hash-linking a non-immediate descendant of the last finalized ShardBlock.

6.4 MasterChain Consensus

The consensus on the MasterChain is reached through a *Proof of Stake* mechanism, where the TokenHolders elect the MasterNodes considered most efficient, trustworthy, and reliable.

6.4.1 Proof of Stake for MasterNodes in the MasterChain

The MasterNodes are selected after a PoS competition. During epoch $h-2$ every TokenHolder that aims to become a MasterNode during epoch $h+3$ submits a *candidacy transaction* of $\mathtt{Ma}_{\min}$ QDTs to a special account called

stake account that is linked to its address and accumulates the *Stake* supporting this *Candidate*. These candidacy transactions should contextually specify two public keys to be used in the SynchroChain consensus of epoch $h+3$: one for the unique-signature algorithm and one for the aggregable-signature algorithm.

Then, during epoch h every TokenHolder can support one or more Candidates by sending some QDTs on their stake accounts. These TokenHolders will be compensated with a share of the reward that the Candidate will earn if it becomes a MasterNode. The share given to a TokenHolder is proportional to the amount of stake it contributed in support of the Candidate, in relation to the total stake accumulated by the Candidate itself.

At the end of epoch h, it is possible to compute the final ranking: from M_1, the Candidate that accumulated the most stake (that amounts to A_1 QDTs), down to M_n, the Candidate who accumulated the least stake (that amounts to A_n QDTs).

Let $n_{\max,h+3}$ and $n_{\min,h+3}$ be respectively the maximum and minimum number of MasterBlocks that a MasterNode might create in epoch $h+3$ and e'_{h+3} the target value for the number of time-slots in epoch $h+3$, these values are published in the EpochBlock E_{h-1}. Then anyone can compute the number of time-slots for MasterBlock creation assigned to each Candidate with the following procedure:

1. to M_1 are assigned $k_1 = n_{\max,h+3}$ time-slots in epoch $h+3$. Each MasterBlock Ma_j^{h+3} that M_1 will create in one of these slots will have weight:

$$w\left(\mathsf{Ma}_j^{h+3}\right) = w_1 = \frac{A_1}{k_1}; \tag{17}$$

2. proceeding in order for $i = 2, \ldots, n$, to M_i are assigned k_i time-slots with:

$$k_i = \begin{cases} \max\left\{n_{\min,h+3}, \min_k\left\{\frac{A_i}{k} \leq \frac{A_{i-1}}{k_{i-1}}\right\}\right\} & \text{if } \sum_{j=1}^{i} k_j \leq e'_{h+3} \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

   Again, each MasterBlock Ma_j^{h+3} that M_i will create in one of these slots will have weight:

$$w\left(\mathsf{Ma}_j^{h+3}\right) = w_i = \frac{A_i}{k_i}. \tag{19}$$

Consequentially the actual number of time-slots in epoch $h+3$ is:

$$e_{h+3} = \sum_{j=1}^{i} k_i. \tag{20}$$

As in the time-slot assignation in the ShardChains, the actual time-slot in which this block must be created is determined by the output of a pseudo-random function with an input that depends on E_h, effectively shuffling the order.

Note that a simple signature is enough for a TokenHolder to prove that it is the rightful MasterNode in a given time-slot, since the slot allocation is publicly computable.

In the EpochBlock E_{h-3} are specified the refund mechanisms: starting from the epoch $h+4$ the stake of each Candidate may be refunded to the original

supporters (including the Candidate themselves), starting from the Candidates to which no time-slot has been assigned, then proceeding with a fraction $\frac{1}{k_i}$ for each MasterBlock produced by M_i that becomes final (i.e. this fraction of the original supporting transaction is refunded sending to each supporter of M_i a part proportional to its contribution to M_i's stake). The remaining stake of M_i, that amounts to $\frac{d}{k_i}A_i$ QDTs where d is the number of time-slots of epoch $h+3$ that were assigned to M_i but during which it failed to produce a block considered valid by the consensus, is subject to a punitive treatment. This punishment might be the seizure and/or the delay of the refund of the entirety or part of this remaining stake.

6.4.2 MasterBlock Validity and Finality

A MasterBlock is considered valid if:

1. it is hash-linked by a SynchroBlock (this implies timing validation, see Section 6.2);
2. its MasterNode is authorized;
3. it is formally correct (this implies that the hash-links are admissible);
4. it is semantically correct (the global state is correctly computed, it hash-links only finalizable ShardBlocks);
5. it is *final* or included in the most ancient unresolved branch (see Definition 5).

A MasterBlock Ma_i^h should hash-link the most recent finalizable ShardBlock (see Section 6.3.7) on each ShardChain and the most recent valid MasterBlock (see Section 6.4.2), so its hash-links are admissible if:

- there are exactly one hash-link to a ShardBlock from each ShardChain, one to a MasterBlock, one to the previous SynchroBlock, and no other hash-links;
- let $\mathsf{Ma}_{i'}^{h'}$ be the MasterBlock hash-linked by Ma_i^h, then:
$$(h' < h) \vee ((h' = h) \wedge (i' < i))\,; \tag{21}$$
- let $\mathsf{Sh}_{\sigma,i_\sigma}^{h_\sigma}$ be the ShardBlock on the shard σ hash-linked by Ma_i^h, then:
$$(h_\sigma < h) \vee ((h_\sigma = h) \wedge (i_\sigma < i)) \qquad \forall\sigma; \tag{22}$$
- let $\mathsf{Ma}_{\hat{i}_\sigma}^{\hat{h}_\sigma}$ be the MasterBlock hash-linked by $\mathsf{Sh}_{\sigma,i_\sigma}^{h_\sigma}$, then $\mathsf{Ma}_{\hat{i}_\sigma}^{\hat{h}_\sigma}$ is an ancestor of $\mathsf{Ma}_{i'}^{h'}$ for every σ;
- let $\mathsf{Sh}_{\sigma,\tilde{i}_\sigma}^{\tilde{h}_\sigma}$ be the ShardBlock on the shard σ hash-linked by $\mathsf{Ma}_{i'}^{h'}$, then $\mathsf{Sh}_{\sigma,\tilde{i}_\sigma}^{\tilde{h}_\sigma}$ is an ancestor of $\mathsf{Sh}_{\sigma,i_\sigma}^{h_\sigma}$ for every σ;

Simply put, the blocks that are indirectly hash-linked cannot be more recent than the blocks that are directly hash-linked.

Similarly to the case of the ShardChains, the EpochBlock E_{h-1} also specifies the value of two parameters l'_h and w'_h that regulate whether a MasterBlock Ma_j^h may be considered final:

- l'_h ensures that the MasterBlock is sufficiently old, i.e. at least l'_h time-slots have passed since the MasterBlock's creation;
- w'_h ensures that disputes in a branch are resolved, in fact the MasterBlock must belong to an unresolved branch that is sufficiently heavier than all other unresolved branches in the MasterChain:

$$\mathsf{Ma}_j^h \in \Gamma^* : w(\Gamma^*) > w'_h + w(\Gamma) = w'_h + \sum_{B \in \Gamma} w(B) \quad \forall \Gamma \neq \Gamma^* \tag{23}$$

The values of l'_h and w'_h are chosen in a way that allows the honest MasterNodes, in case of attack, to overcome the attacker's branch and not to be afraid to hash-link the correct block on the older branch, even if that branch is shorter.

7 Quadrans Tokens and Quadrans Coins

The Quadrans Blockchain handles two kind of currencies: Quadrans Tokens (QDTs) and Quadrans Coins (QDCs).

7.1 Quadrans Tokens

A total of 600 Million QDTs are created once in the initialisation blocks of the Quadrans Blockchain. Details on the genesis block are decided by [29].
A TokenHolder with at least 1.000 QDTs can participate in PoW competitions to become a Miner (see Sections 6.3.1). A TokenHolder with at lease 100.000 participate in PoS competitions to become a MasterNode (see Section 6.4.1).
Another use of QDTs is the payment for the creation of new Smart Contracts, which will be decided according to the [29]. (see Section 8).
Finally, a TokenHolder earns QDCs through coinbase transactions, proportionally to the quantity of owned QDTs. This proportions will be decided according to the [29].

7.2 Quadrans Coins

The primary use of the QDCs is to provide gas to pay for Smart Contracts' execution (see Section 8). New QDCs are minted by MasterNodes through a process involving both the PoS competition and the collaboration of Miners working on ShardBlocks.
QDCs' inflation is set to be large enough to discourage hoarding and financial speculation[5].

7.3 Minting new QDCs

Coinbase transactions are included in ShardBlocks, according to the address of the TokenHolder that earns QDCs according to the rules set by [29].

To the creation of a MasterBlock it corresponds a coinbase transaction, containing the newly minted QDCs corresponding to the creation of MasterBlock, to be distributed[6] between:

[5]The quantity of minted QDCs may increases every year by a quantity according to [29].

[6]The proportion of QDCs for the MasterNode, the Miner and the TokenHolder will be decided according to [29]

- the MasterNode α that has created the MasterBlock,
- the TokenHolders that have bet a stake for the election of α;
- the Miner that has included the coinbase transaction in the ShardBlock,
- all TokenHolders, proportionally to the quantity of QDTs owned, including their frozen QDTs used in the PoS competitions.

Coinbase transactions are managed by Miners not before the finalization of the corresponding MasterBlocks.

8 Smart Contracts

In addition to *kernel smart contracts* Quadrans encourages the development of a rich ecosystem of *user smart contracts* that can be added to the Quadrans Blockchain through dedicated transactions. The cost of the deployment of a new contract is twofold: the gas price is paid in QDCs, while the right to create a new contract is paid using QDTs. These QDTs are moved to a special account, known as *Enhanced Account*, managed by the Quadrans Foundation.

Smart Contracts are divided into three categories, according to their gas cost:

- Standard Contracts, whose gas cost is paid by the user requesting the execution;
- Autonomous Contracts, whose gas cost is paid (partially or in toto) by a reserve of QDCs held by the contract itself;
- Favoured Contracts, whose gas cost is paid only partially by the user and the remaining cost is paid using the Enhanced Account.

9 Future Works

The development of the Quadrans Blockchain and its ecosystem is still in progress, and a continuous effort is expected in the future in order to keep the technology always up-to-date.

In this section we delineate the topics that are expected to be considered in the near future.

9.1 Developments on Addresses

Currently, an address corresponds to a single public key, and a signature verifiable with that key is sufficient to authorise any operation on behalf of this address. We refer to these as simple addresses.

Here we discuss some possible developments on this topic.

9.1.1 Authorised Keys

The first and obvious evolution of simple addresses should be addresses that natively support multi-signatures and even more complex access policies. A direction could be to define addresses embedding information about the public keys whose corresponding private keys can withdraw from these addresses' balances. Additionally, more sophisticated addresses may specify various public keys (even of different signature protocols) and a policy that indicates which signatures are required to authorise transactions. For example an address could be associated to a public key for a post-quantum algorithm plus three other ECDSA keys and specify that authorisation may come through a single post-quantum signature OR at least two out of three ECDSA signatures.

9.1.2 Common Name

Another possible evolution of Quadrans addresses could be the embedding of common names for their users. This feature is useful to avoid duplicated insertions of the same public key to the blockchain: once an address has been disclosed, all information (including the public key and the common name) are known to the community, hence further transactions can refer to address and common name only, without the need of specifying the entire public key. We remark that common names could also be not fully human-readable: for example, they can also encode IOT devices IDs in a way that only the owner may recognise and identify them, or even be random strings if some users wish to preserve their pseudo-anonymity.

References

[1] Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: International Conference on Cryptology in Africa. pp. 389–405. Springer (2008)

[2] Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering 2(2), 77–89 (2012)

[3] Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS+ signature framework. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 2129–2146 (2019)

[4] Bernstein, D.J., Josefsson, S., Lange, T., Schwabe, P., Yang, B.Y.: EdDSA for more curves. Cryptology ePrint Archive 2015 (2015), `https://eprint.iacr.org/2015/677`

[5] Bruinderink, L.G., Hülsing, A., Lange, T., Yarom, Y.: Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 323–345. Springer (2016)

[6] Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - a practical forward secure signature scheme based on minimal security assumptions. In: International Workshop on Post-Quantum Cryptography. pp. 117–129. Springer (2011)

[7] Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform. `https://github.com/ethereum/wiki/wiki/White-Paper` (2013)

[8] Casanova, A., Faugere, J.C., Macario-Rat, G., Patarin, J., Perret, L., Ryckeghem, J.: GeMSS: a great multivariate short signature. Submission to the NIST's post-quantum cryptography standardization process (2017), `https://www-polsys.lip6.fr/Links/NIST/GeMSS_specification_round2.pdf`

[9] Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: OSDI. vol. 99, pp. 173–186 (1999)

[10] Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Zaverucha, G.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1825–1842 (2017)

[11] Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions. CoRR abs/1608.00771 (2016)

[12] Costa, D., Fiori, F., Milan, P., Sala, M., Vitale, A., Vitale, M.: Quadrans whitepaper (2019), `https://quadrans.io/content/files/quadrans-white-paper-rev01.pdf`

[13] Costa, D., Fiori, F., Sala, M., Vitale, A., Vitale, M.: Introducing Quadrans (2019), `https://quadrans.io/content/files/quadrans-light-paper-en.pdf`

[14] Di Chiano, N., Longo, R., Meneghetti, A., Mula, M., Tognolini, G.: Quadrans's Cryptographic Kernel and Primitives Encoding (2021)

[15] Ding, J., Schmidt, D.: Rainbow, a new multivariable polynomial signature scheme. In: International Conference on Applied Cryptography and Network Security. pp. 164–175. Springer (2005)

[16] Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems 2018(1), 238–268 (Feb 2018), `https://tches.iacr.org/index.php/TCHES/article/view/839`

[17] Flamini, A.: A Byzantine Fault Tolerant Consensus Protocol for Parallel Time-Stamping (2021), `https://webapps.unitn.it/Biblioteca/it/Web/Tesi`

[18] Flamini, A., Longo, R., Meneghetti, A.: Cob: a multidimensional byzantine agreement protocol for asynchronous incomplete networks. arXiv preprint arXiv:2108.11157 (2021)

[19] Flamini, A., Longo, R., Meneghetti, A.: Multidimensional byzantine agreement in a synchronous setting. arXiv preprint arXiv:2105.13487 (2021)

[20] Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier lattice-based compact signatures over NTRU. Submission to the NIST's post-quantum cryptography standardization process (2018), https://www.di.ens.fr/~prest/Publications/falcon.pdf

[21] ITU: X.509 : Information technology - open systems interconnection - the directory: Public-key and attribute certificate frameworks (Oct 2019), https://www.itu.int/rec/T-REC-X.509-201910-I/en

[22] Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). International Journal of Information Security 1(1), 36–63 (2001)

[23] Meneghetti, A., Parise, T., Sala, M., Taufer, D.: A survey on efficient parallelization of blockchain-based smart contracts. Annals of Emerging Technologies in Computing (AETiC) 3(5) (2019)

[24] Meneghetti, A., Sala, M., Taufer, D.: A note on an ECDLP-based pow model. CEUR Workshop Proceedings Vol-2580 DLT 2020 (2020)

[25] Meneghetti, A., Sala, M., Taufer, D.: A survey on pow-based consensus. Annals of Emerging Technologies in Computing (AETiC) 4(1) (2020)

[26] Micciancio, D., Walter, M.: Gaussian sampling over the integers: Efficient, generic, constant-time. In: Annual International Cryptology Conference. pp. 455–485. Springer (2017)

[27] National Security Agency: Quantum computing and post-quantum cryptography: Frequently asked questions (Aug 2021), https://media.defense.gov/2021/Aug/04/2002821837/-1/-1/1/Quantum_FAQs_20210804.PDF

[28] Patarin, J., Courtois, N., Goubin, L.: Quartz, 128-bit long digital signatures. In: Cryptographers' Track at the RSA Conference. pp. 282–297. Springer (2001)

[29] Sala, M., Costa, D., Fiori, F., Crotta, M.: Quadrans Blockchain - Code of Conduct (2021), https://quadrans.io/f/code-of-conduct

[30] Schmid, M.: ECDSA-application and implementation failures (2015), https://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Schmid.pdf

[31] Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th annual symposium on Foundations of Computer Science. pp. 124–134. IEEE (1994)

[32] of Standards, N.I., Technology: Post-quantum cryptography standardization - post-quantum cryptography, https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization