ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

MASTER'S PROJECT

Using Pairing-Based Cryptography to Improve ByzCoin's Robustness to Faults

Author:
Christopher BENZ

Supervisor:
Eleftherios KOKORIS KOGIAS

School of Computer and Communication Sciences
Decentralized and Distributed Systems lab

June 29, 2018

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Abstract

School of Computer and Communication Sciences
EPFL

Using Pairing-Based Cryptography to Improve ByzCoin's Robustness to Faults

by Christopher BENZ

Scaling distributed ledgers has been a long-standing issue. One proven way of scaling out is by sharding; in sharding, nodes are partitioned into shards that each need to reach consensus on a particular part of the state. Hence, they can commit transactions in parallel.
Sharding, however, requires a strongly consistent consensus to be run inside each shard, in order to achieve fast finality and also requires this consensus to be run among hundreds to a thousand nodes, in order for the system to be secure. Previous work on consensus mechanisms has not managed to deliver a consensus algorithm suitable for sharding: (1) **PBFT** does not scale much. The original paper tested only 4 and 7 nodes, which is way too small for a true permissionless distributed ledger. (2) **Bitcoin**, the first and most prominent blockchain, is currently limited to 3-7 transactions per second and its finality guarantees are only probabilistic. Finally, (3) **ByzCoin** takes Bitcoin and improves on it with a PBFT-like consensus mechanism, using Collective Signing, which is based on the Schnorr signature scheme.
Unfortunately, in order for ByzCoin to scale it uses a tree communication pattern, which can be fragile in an adversarial environment. Furthermore, the use of Schnorr signatures makes ByzCoin susceptible to Denial-of-Service or slowdown attacks during the Schnorr randomness disclosure step.
We propose to use pairing-based cryptography and a fixed three-level tree communication pattern to create a consensus mechanism that both scales and is fault tolerant, even under strong adversaries.
Furthermore, we describe and implement **Bls-ByzCoinX**, which uses the Boneh-Lynn-Shacham signature scheme to improve scalability of the consensus mechanism and to make it more reliable. Finally, we evaluate the implementation performance by comparing it with existing solutions and show that Bls-ByzCoinX, which provides more robustness than ByzCoin, scales up to 1000 nodes while being at most two times slower than ByzCoin.

Contents

1 Introduction

Since its inception 10 years ago, blockchain technology has emerged as a new way of designing decentralized ledgers. Being truly permissionless and incentivizing nodes to join the network to make it more secure, Bitcoin [17] revolutionized the field and brought a new spark to the development of decentralized databases for many different use-cases [16] [9] [10].

While most decentralized ledgers have proven strength in decentralization and security [11], most do not scale out well [24], which means that augmenting the number of validators in the network does not scale the transaction throughput.

One proven way to scale out distributed systems is by sharding [7]: nodes are partitioned into separate shards that each compute a different state of the block validations. This allows for faster validation of transaction logs, as each shard can validate a subset of them in parallel. Previous work shows that around 600 validators per shard is good enough to scale out to thousands of nodes with very low probability of failure [14].

PBFT [3] is an early work that works well for very few nodes only (the original paper only mentioned 4 and 7 nodes, and most deployements never go above 16 nodes [15] [6] [8]) because of its communication pattern, which involves broadcasting each message to all other nodes. On top of that, it is not permissionless because the consensus group is closed. These reasons make it unsuitable for scalable consensus.

ByzCoin [13] and ByzCoinX [12] take Bitcoin and improve it with a PBFT-like consensus mechanism and a new tree communication pattern for high scalability. ByzCoin uses either a two-level flat tree, or a deeper tree communication pattern. ByzCoinX compromises between the two and uses a three-level tree. But both have a fault tolerance problems. A single node can make the protocol fail and start again without being able to discriminate the attacker by exploiting the Schnorr signature mechanism. Such an easy Denial-of-Service attack makes it easily exploitable to slow downs at a low cost for the attacker.

Pairing-based cryptography could solve the fault tolerance issue because it only needs one round of communication and does not use randomness, contrary to the Schnorr signature scheme, which eliminates the possibility of a Denial-of-Service attack. It also decouples the signing phase from the aggregation, which allows nodes to incrementally aggregate signatures and move rapidly to the next step once a threshold of participants is reached.

We introduce Bls-ByzCoinX, a consensus mechanism capable of issuing collective signatures that uses the Boneh–Lynn–Shacham signature scheme [1]. Its one-round trip defends against Denial-of-Service attacks that are possible in ByzCoin, making it more robust while keeping good scalability.

Our evaluation of Bls-ByzCoinX shows that it can confirm transactions with strong consistency under 30 seconds for a 1 MB block, which can make it useful for near real-time transactions in sharding protocols. We evaluated our implementation in terms of latency, block size and transactions per second and compared it to the aforementioned works.

2 Background

This section presents the necessary backgrounds needed to understand the report from a technical point of view.

2.1 Practical Byzantine Fault Tolerance

Historically, previous work on distributed consensus comes from Byzantine Fault Tolerance (BFT) research. It all begins with the question of *how to reach an agreement among multiple parties, while knowing some of them can by malicious or faulty*.

To explain better, the Byzantine General Problem story was imagined by Leslie Lamport, Robert Shostak and Marshall Pease in their 1982 paper, which is summarized here:

A group of generals, each commanding a portion of the Byzantine army, encircle a city. These generals wish to formulate a plan for attacking the city. In its simplest form, the generals must only decide whether to attack or retreat. Some generals may prefer to attack, while others prefer to retreat. The important thing is that every general agrees on a common decision, for a halfhearted attack by a few generals would become a rout and be worse than a coordinated attack or a coordinated retreat [23].

This allegory also transcribes to algorithmic consensus protocols. The generals become nodes in a network and, upon receiving a computation request, we want them to be able to agree on the result, even if some of the nodes break down or are corrupted (i.e. they are maliciously trying to falsify or block the result).

Published in 1999 by Castro and Liskov, the Practical Byzantine Fault Tolerance (PBFT) [3] algorithm is the first practical implementation of a Byzantine Fault Tolerant algorithm, and it worked in an asynchronous environment such as a computer network, which made it useful for real-world cases. Provided no more than $(n-1)/3$ nodes are faulty, the algorithm satisfies both safety ("nothing bad will ever happen", i.e. the protocol cannot get blocked in some state) and liveness ("something good eventually happens", i.e. the algorithm will always return a result), two properties that are important for real-world use-cases of a database [19].

However, PBFT was not designed for scalability and only works for very few replicas. Experiments often use only 4 or 7 nodes [15], which limits its use-case to keeping a system tolerant to software errors or single server breakdown, but not for scaling a decentralized system to thousands of nodes.

One of the reasons it doesn't scale well is because at every step, each node needs to broadcast its message to all other nodes, creating a complexity of $O(n^2)$.

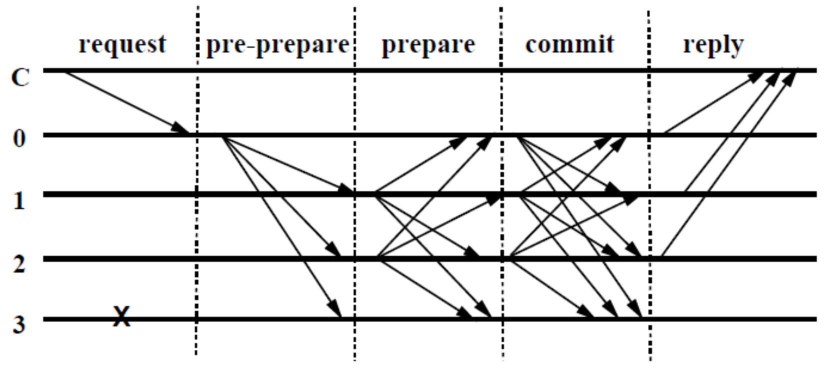

FIGURE 2.1: The normal case of a run of PBFT.

Additionally, the system is not permissionless, in the sense that it cannot accept any new node to join the network at any time. The nodes are a closed-group of servers that are defined before the system starts and cannot be changed. For a decentralized ledger, we need any node to be able to join the network in a free and open way.

We give a simple description of the main steps in the protocol, composed of three phases *pre-prepare*, *prepare* and *commit*:

One of the nodes is the *primary*, which will receive the request from the client, and all other nodes are the *backups*. When it receives a message, the primary will broadcast it to all the backups during a *pre-prepare* phase, along with the message's digest. Every backup verifies the message, and if everything is correct, it enters the *prepare* phase, during which it broadcasts the message alongside its own id to all other nodes. It then waits for receiving at least $\lceil 2/3 + 1 \rceil$ of the prepare messages from other nodes. After the threshold, it enters the *commit* phase, where it will once again broadcast the message to all other nodes. It then again waits for $\lceil 2/3 + 1 \rceil$ of the *commit* broadcasts from other nodes. When the threshold is reached, the node knows that the message is validated by the network, and it sends a confirmation to the client.

This mechanisme guarantees that the network has reached consensus if less than $1/3$ nodes are faulty.

To resume, the drawbacks of PBFT are the following:

- It doesn't scale out nor scale up. PBFT works well only for very few replicas. In the original paper, only 4 and 7 replicas were tested.
- It is not permissionless. Nodes are fixed and static, which makes the algorithm unusable for an open and permissionless system.
- The complexity of communication between nodes is $O(n^2)$. This complexity comes from the fact that each node in the network has to communicate with every other one during the algorithm.
- A client waits for a response from $\lceil 2/3 + 1 \rceil$ of the nodes in order to confirm that the transaction was completed and is correct. Before then, it cannot

know if the received answer is correct or not. It therefore has to keep track of how many responses it got from the network, how many nodes are in the network, and verify each response.

2.2 Bitcoin

Bitcoin [17] as presented in Satoshi Nakamoto's 2009 paper proposed a novel way of making consensus in a distributed ledger, by using a Proof-of-Work system.

The solution was revolutionary. Unlike other works such as PBFT, Bitcoin was permisionless in the sense that any node can arbitrarily join the network of validators without prior permission. Not only that, but the Bitcoin system is made so that nodes are *incentivized* to join the network [22], by earning monetary rewards in the form of bitcoin.

Today, Bitcoin's Proof-of-Work solution has proved its strength by scaling up to thousand of nodes without issues, and offers one of the most powerful decentralized distributed ledger against attacks. To attack the network, an attacker would need an enormous amount of hashing power. At the time of writing, the Bitcoin network has a total hashing power of 30 million TH/s. That puts the cost of a 51% attack, which is needed for a 100% attack success, at over $6 billion USD [5].

The computation put into the network by the miners is what makes Bitcoin secure, but it has the undesirable effect of consuming enormously lot of energy power [18], which has been largely critisized because of the effect on the environment.

Bitcoin can not scale out either: the transaction throughput is limited to 3-7 per second, whatever the number of validators is. Bitcoin's throughput is often compared to the one of Visa, considered as a competitor in the area of digital exchange of money, and a good point of comparison. At its peak, Visa achieves above 47000 transactions per second [20]. Bitcoin has no way of scaling out the number of transactions per second in its current form. We could even argue that with a growing number of validators, the Bitcoin network becomes slower and less decentralized because of the time for sending the new blocks through a bigger network.

Bitcoin also has a high transaction confirmation time. Assuming no group of miners has more than 50% of the total hashing power of the network, a transaction is confirmed with very high probability after one hour (6 blocks of 10 minutes) [**confirmation**]. This makes Bitcoin unsuitable for real-time transactions.

2.3 CoSi

CoSi [21] is a protocol for scalable **collective signing**. It allows a client to request that a message is publicly signed by a group of *witnesses*, resulting in a collective signature that has the same length and verification cost as a single signature.

Such a system has many applications, particularly in the area of signing authorities. Very important internet services are provided by **centralized authorities**: time and timestamp services, certificate authorities, directory authorities, software update services, randomness services, etc. These services are high-value targets for hackers.

CoSi is a proactive solution to replace or complement the authorities.

It works in the following way:

- When an authority publishes a new signing key (e.g. a browser root certificate bundle), it also publishes with it the identities and public keys of a large set of trusted witnesses.
- Whenever the authority signs a new authoritative statement (e.g. certificate, timestamp, log record, etc.), it sends the statement to the witnesses.
- It then collects all the cosignatures.
- Finally, it aggregates all the cosignatures along with its own signature. The final signature is attached to the statement, which is sent to the receiving client.
- The client verifies that the statement has been signed by a great enough subset of witnesses (e.g. 50%).

The client can therefore verify without communication that a large number of parties have witnessed and logged the statement *before* it accepts it.

With W witnesses, a MITM attacker needs the secret key of the authority and those of W witnesses, or submit faked statement to witnesses for cosigning, thereby exposing it publicly and risking detection.

We do not expect witnesses to detect all malicious statements immediately. But they can sanity-check the correctness and consitency of proposed statements before cosigning. If a conflict is detected, even if we do not know which is malicious, it is now publicly known that there is a conflict.

CoSi uses a spanning tree with the leader at the top and all witnesses distributed so that the tree has depth $O(logN)$. This communication pattern is utilized in other multicast and aggregation protocols [4] [2].

To perform one round, the CoSi protocol needs **2 round-trips** (composed of four messages) between the leader and all the witnesses: the leader first sends an **Announcement** down the tree. All the nodes then respond with a **Commit** message, which contains a Schnorr randomness. While going up the three, the commits are aggregated sequentially by nodes to arrive at one aggregated commit. The leader then sends a **Challenge** back down the tree, containing a collective Schnorr challenge. The witnesses finally respond with a **Response** containing the answer to the collective challenge, and they aggregate the messages similarly to the commit phase.

CoSi can be used for collective signing, but for reaching consensus, we cannot simply run a single round because of the following issues:

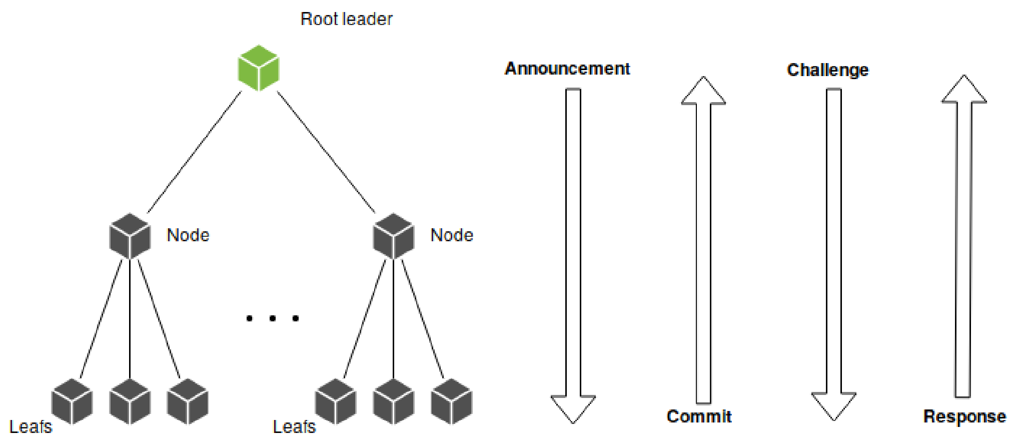

FIGURE 2.2: Communication pattern of CoSi.

- If a node fails to respond at any step of the protocol, we have no way of knowing if it is malicious or if it failed due to one of its children. In other words, the deep tree structure makes it impossible to blame one node for failing without him blaming another honest node.
- If a node fails to respond during the second round-trip, we cannot start another run of CoSi while guaranteeing that the node will not finalize the signature later on.

2.4 ByzCoin

Bitcoin only provides *probabilistic consistency*: because of the possibility of a fork (e.g. multiple miners find different valid blocks before the network has reached consensus, or in the case of an attacker with a high amount of hash power), it is recommended to wait 6 subsequent blocks (called confirmations) [**confirmation**] before being able to consider a transaction to be valid with high probability.

ByzCoin [13] improves on this and provides *strong consistency*, meaning that once a transaction is commited to the blockchain, it can immediately be considered as valid. There is also no wasted computation power on inconsistent forks.

To do so, ByzCoin is based on PBFT and uses CoSi to make it more scalable. Two rounds of CoSi are performed: the first one is to ensure that all the nodes are willing to sign the statement. The protocol moves on to the second round only if a threshold of participating nodes is attained during the first round.

If a node doesn't respond during the first round, we restart the protocol and ignore the non-responding node. If a node participates in the first round and for whatever reason doesn't respond during the second round, all nodes can detect that he did promise during the first round that he would participate. The node can then be blamed by banning it from the system, before launching a new instance of CoSi.

ByzCoin can either work with a deep tree structure or a flat tree (two-level). The latter choice is less scalable because of the network overhead on the root, and is also more susceptible to Denial-of-Service on the root, who has to handle messages with all the other nodes. In the evaluation, we compare only to a deep tree structure, which is more fair to ByzCoin's scalability.

ByzCoin has a communication cost of $O(logN)$ with N the number of nodes, thanks to the tree communication pattern, which scales way better than PBFT's full broadcast system.

ByzCoin solves the problems that we have with a single round of CoSi (see 2.3), but it still has drawbacks and is not completely fault tolerant because it is susceptible to Denial-of-Service attacks: during the Commit Phase, a malicious node can begin by sending a correct commitment message, which contains a Schnorr randomness, to its parent. At the Challenge phase, this node can then not send its Response. The aggregated Commitment and Challenge will need this node's Response in order for the protocol to work. By not responding, the node can make the system wait for a timeout. Then, the network has to begin a whole new round of the signing process. It is therefore very easy for a single node to create Denial-of-Service attacks on the full protocol.

Note that even without malicious nodes, failure of a (single) node at the Response step results in the whole protocol to restart. With the goal in mind of building a scalable blockchain, the probability of a node failing gets higher with a large number of nodes in the network and a good system should support without problem the failure of a single node.

2.5 ByzCoinX

ByzCoinX [12] is an improved version of ByzCoin that is able to prevent the Byzantine Denial-of-Service attack.

It uses two rounds of CoSi for the same reason as ByzCoin, but uses a three-level tree, so that there's only one intermediate layer of nodes in the communication pattern. Nodes are separated into groups of equal size, and each group has a subleader (second-level nodes) which is in charge of collecting the signature of all its children (leafs).

With this communication pattern, a non-responding node can be ignored by the subleader, who waits for collecting a certain threshold of responses from all its children. If the subleader itself isn't responding, the root leader elects a new subleader inside the group. If it's the root node who fails, the protocol is restarted with a new leader.

2.6 The Boneh–Lynn–Shacham signature scheme

The Boneh–Lynn–Shacham (BLS) signature scheme [1] uses pairing-based cryptography, which is an extension of elliptic curve cryptography, to allow a party to verify that a signer is authentic. One of its strengths is the ability to easily aggregate signatures, which we use here for cosigning messages.

First, we describe the basic operation of BLS.

Let G_0, G_1 and G_T be three cyclic groups of prime order q, and let g_0, g_1 and g_t be their respective generators. We introduce the following definitions, which are necessary ingredients for BLS:

- **Pairing $e : G_0 \times G_1 \rightarrow G_T$**, with a bilinear property satisfying

  $$e({g_0}^a, {g_1}^b) = e({g_0}^b, {g_1}^a) = e(g_0, g_1)^{ab}$$

  Basically, applying the exponents to either generator or the output of the pairing gives the same result. The pairing is efficiently computable and non-degenerate.

- **Hash function $H : \mathcal{M} \rightarrow G_0$**, which takes an input (the message) and maps it to a pseudorandom point on G_0.

The scheme is then composed of three functions:

- $\boldsymbol{GenerateKey()}$: the private key is a random: $x \in \mathbb{Z}_q$ and the public key is $pk \leftarrow {g_1}^x \in G_1$.
- $\boldsymbol{Sign(x, m)}$: $\sigma \leftarrow H(m)^x \in G_0$. Returns the signature σ.
- $\boldsymbol{Verify(pk, m, \sigma)}$: returns *True* if $e(g_1, \sigma) = e(pk, H(m))$, *False* otherwise.

Thus, the pairing operation is used for the signature verification to check if both the public key pk and the signature σ were generated by the same private key.

BLS allows for easy aggregation of signatures as follows. Let (pk_i, m, σ_i) for $i = 1, ..., n$ be triples for n users and a same message m. The signatures can be aggregated by $\sigma \leftarrow \sigma_1 \cdots \sigma_2 \in G_0$, i.e. addition in the group G_0.

To verify the aggregate signature σ, we then verify that

$$e(g_1, \sigma) = e(pk_1 \cdots pk_n, H(m))$$

If the equation holds, we know that the aggregated signature was signed by all participating parties, thus also proving that the message was witnessed by all of them.

The resulting signature is an element of an elliptic curve group and holds in 64 bytes.

As one can observe, the BLS scheme offers a very easy way of doing signature aggregation and verification of signature for authentication.

This scheme is a central point of the project. Combined with a tree structure, BLS can produce a collective signature in a single round-trip, instead of two with the usual Schnorr signatures: a message is sent down the tree, and the nodes simply respond with the signature, while intermediate nodes aggregate step by step to obtain a single final collective signature.

The one-round trip of BLS mitigates Denial-of-Service attacks. The powerful difference of BLS with more classical Schnorr signatures is the ability to aggregate signatures after they are created. With Schnorr (as used in CoSi), the signatures have to be aggregated when they are computed. With BLS, it is possible to sign a message, and aggregate multiple signatures later on, in any order.

3 System Overview

3.1 Interface

From a client's point of view, the system is a service that offers to make the collective signature of a given message. After receiving a client's request for signature, the network of validators computes the collective signature and sends the result back to the client. Upon receiving the response from the server, the client knows that the network has reached consensus and the collective signature proves that at least $\lceil 2/3 + 1 \rceil$ nodes witnessed the message and took part in the signing.

Any party can then verify that the message was indeed signed by an at-least $\lceil 2/3 + 1 \rceil$ majority of the nodes.

3.2 Assumptions

Any node in the network can join the consensus group. Each of them creates a public-private key pair as described in 2.6, and makes the public key available to the other nodes. There is no need for a trusted third-party key provider, so the system is truly decentralized and open.

We assume the network has a weak synhrony property, as described in PBFT [3].

The consensus group contains a root leader (e.g. elected by the sharding protocol), who is reponsible for creating the subtrees and electing subleaders.

3.3 Threat Model

With N total validators, we assume at most f with $3 * f = N$ validators are Byzantine. This means we must have at least $\lceil 2/3 + 1 \rceil$ honest nodes that are complaisant and follow the protocol correctly.

We assume the malicious nodes can behave arbitrarily: they can be coordinated by an attacker to disrupt the state of the system, can refuse to participate, break down, not respond to some of the messages, etc.

Bls-ByzCoinX is vulnerable to Denial-of-Service on some of the nodes in the consensus group: if the attacker controls a subleader or the root node, it can not respond to annoucements, which will trigger the parent node only after the timeout. Single leaf nodes cannot trigger Denial-of-Service alone. For $n = \sqrt{N}$ subtrees, there are $\sqrt{N} + 1$ (all subleaders and the root node) that can slow down the protocol.

3.4 System Model

Behind the scenes, the nodes in the network are arranged in a three-level tree structure.

Let N be the number of validator nodes in the network. One of the N nodes is the root leader. The remaining $N-1$ nodes are partitioned into n subtrees of same size and a subtree has a single subleader (directly below the root leader) and other nodes in the sub-tree are leafs.

We use $\left\lceil \sqrt{N} \right\rceil$ for the number of subtrees because it's a fair compromise between more subtrees or more nodes per subtree. Simple simulations on our protocol support this argument.

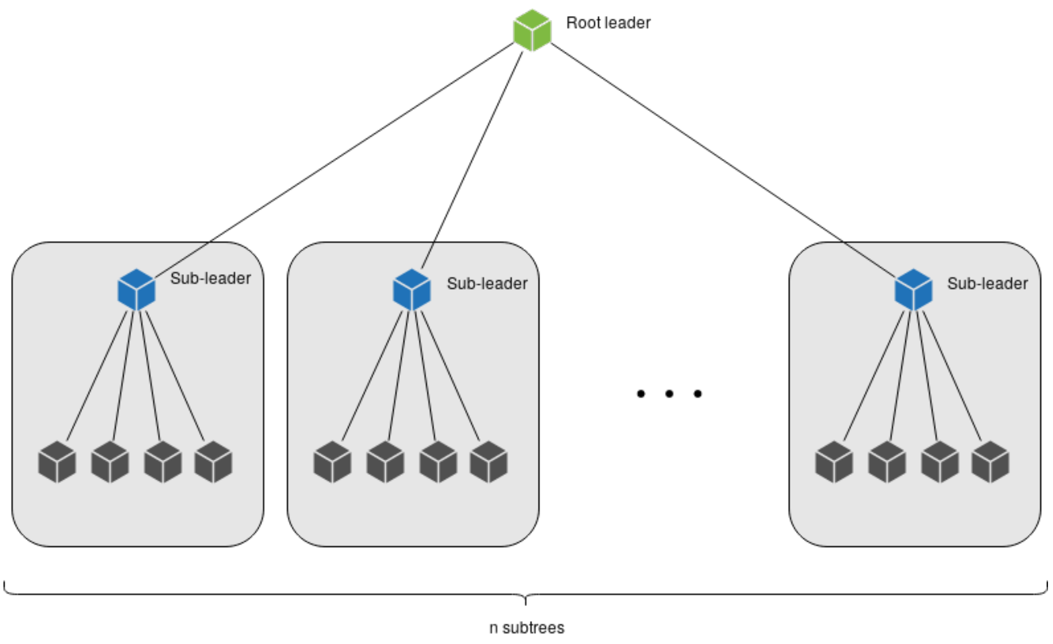

FIGURE 3.1: Bls-ByzCoinX's tree structure.

This structure allows for a better responsiveness in case of malicious or erronous nodes: if a subleader is not responding, its subtree elects a new subleader, but the protocol continues in the meanwhile. Because we only need $\lceil 2/3+1 \rceil$ of all the nodes to sign (under our assumption of no more than 1/3 dysfunctioning nodes), we do not always have to wait on all nodes to respond.

A two-level tree makes the root node too easily vulnerable to a Denial-of-Service attack. As the network scales, the root leader would find itself responsible for more and more child nodes. While sending and receiving messages, the overhead would augment linearly with the number of validators.

With more than three levels, the algorithm gets slowed down by the latency between the root node and the leaves. Augmenting the number of levels in the tree increases the round-trip time.

Therefore, a three-level tree is a good compromise between the two.

Each node in the network has a public/private key pair (pk_i, sk_i) generated with the BLS $\boldsymbol{GenerateKey}()$ function. The public keys pk_i for all nodes i are known by every node in the system.

A round-trip happens in the following way during the protocol: the root leader multicasts a message to all his children (the sub-leaders), which in turn multicast the message to all their own children. Each node n_i then signs the message m with the $\boldsymbol{Sign(sk_i, m)}$ method, using its private key pk_i to get a signature σ_i. When a leaf has computed the signature, it sends it up to its subleader. The subleader of every subtree is responsible for collecting all of its child's signatures, aggregating them all together with its own signature, then sending the aggregated signature for the subtree to the root leader. The root leader therefore receives one aggregated signature per subtree. It can then compute its own signature, and aggregates all the received aggregated signatures together. The final result is a single cosigned signature, which is sent back to the client.

Here lies the power of the Boneh–Lynn–Shacham signature scheme: a signature can be computed by a node and sent to its parent, who aggregates the signatures later, without the need of the node who created the signature anymore. Without pairing-based cryptography, signature aggregation has to be done at the moment of creating the signature.

3.5 Failures

Bls-ByzCoinX is fault tolerant and can handle up to $1/3$ Byzantine nodes.

Each subtree can tolerate up to $1/3$ failures, as $\lceil 2/3 + 1 \rceil$ is the threshold for the subleader to aggregate the signatures and send the result to the root. Therefore, leaf failures are very well tolerated.

If a subleader is no responding after a timeout, the root node elects a new subleader in that subtree.

Finally, if the root is failing, which is detected by the client after a timeout, the system has to elect a new one and restart the round.

Compared to ByzCoin, we have removed the possibility of a single leaf node to make the entire protocol fail. First- and second-level nodes are still a point of failure. For n subtrees, this means that $n + 1$ nodes (subleaders and the root) are a more fragile point of failure, while $N - (n + 1)$ nodes are resistant.

3.6 Advantages

Bls-ByzCoinX solves many of the other solutions problems, and brings additional advantages.

Because the BLS signature scheme allows for a one round-trip (instead of two in the case of ByzCoin and ByzCoinX), we remove the possibility for a single node to perform an easy Denial-of-Service attack, by commiting during the first round-trip then ignoring the second. With Bls-ByzCoinX there is not commit phase, therefore a node can not promise to participate then dissapear during the response phase. A subleader waits for $\lceil 2/3 + 1 \rceil$ of its children to respond, and once that threshold is reached it can aggregate the children's signatures and send back to the root leader. With the BLS mechanism, a single failing or malicious leaf cannot block the protocol and is ignored. The robustness gain is enormous in this sense: Denial-of-Service attacks are way more difficult and require many more resources, because the attacker has to control a greater portion of the total nodes.

Another powerful trait of BLS is its ability to separate the signing phase from the aggregation, which makes it easier to use in asynchronous environments. Subleaders can incrementally aggregate signatures until the $2/3 + 1$ threshold is reached, at which point it can propagate back up the result.

3.7 Augmenting fault tolerance

In our implementation, once the subleader reaches the threshold, it sends the aggregated signature the the root node and finishes its work, discarding any additional childs that would send their response. This means that the root node collects $\lceil 2/3 + 1 \rceil$ responses only from each subtree, and therefore he needs the responses from each subleader in order to reach the total threshold (which is also $\lceil 2/3 + 1 \rceil$).

To improve fault tolerance even more, we could make the subleaders wait for more than $\lceil 2/3 + 1 \rceil$ responses from its subtree before sending up to the root node. This way, a subleader would collect more signatures than needed, and the root node could reach the global threshold faster, without having to wait on every single subleader to respond. This different mechanism could thus tolerate Byzantine subleaders.

Another similar way of improving fault tolerance is for the subleaders to send the aggregated signature once the $\lceil 2/3 + 1 \rceil$ of the subtree is met, and send additional signatures later on, individually or grouped, to complete its contribution incrementally. This solution makes the system less simple and augments the network overhead on the root node, who has to manage more responses, but also allows for Byzantine subleaders.

4 Implementation

4.1 Bls-ByzCoinX

We implemented the protocol described in 3 in Go, using the onet framework of DEDIS lab and made it available on Github. Below we describe the subtleties of the implementation and also the intention of each file.

4.1.1 Messages

The protocol uses two messages:

- **Annoucement** Which is sent from the root down the tree. It contains the message to be signed and data to verify it.
- **Response** Which is sent back up the tree in response to the annoucement, starting from the leaves up to the root. It contains a signature or an aggregation of signatures, which are both binary marshaled elliptic curve points.

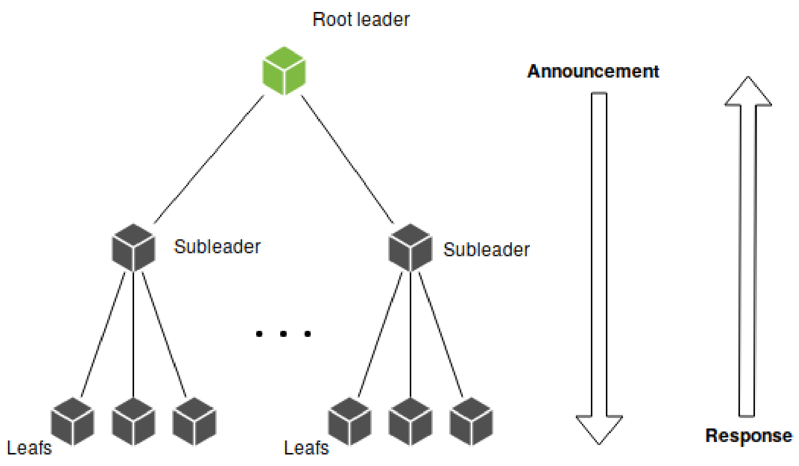

FIGURE 4.1: Communication pattern and messages in Bls-ByzCoinX.

4.1.2 Tree structure abstraction

The root leader receives the signing request from the client and he starts the protocol. From his point of view, the leafs are hidden behind the abstract nodes which are the subleaders. All he sees are the subleaders, which take care of their own subtree.

For n subtrees, the root leader runs n different protocols, each having one unique child, which is a subleader of one of the subtrees. When each protocol has finished aggregating the signatures for a subtree, the root leader aggregates together the n aggregated signatures to obtain the final signature.

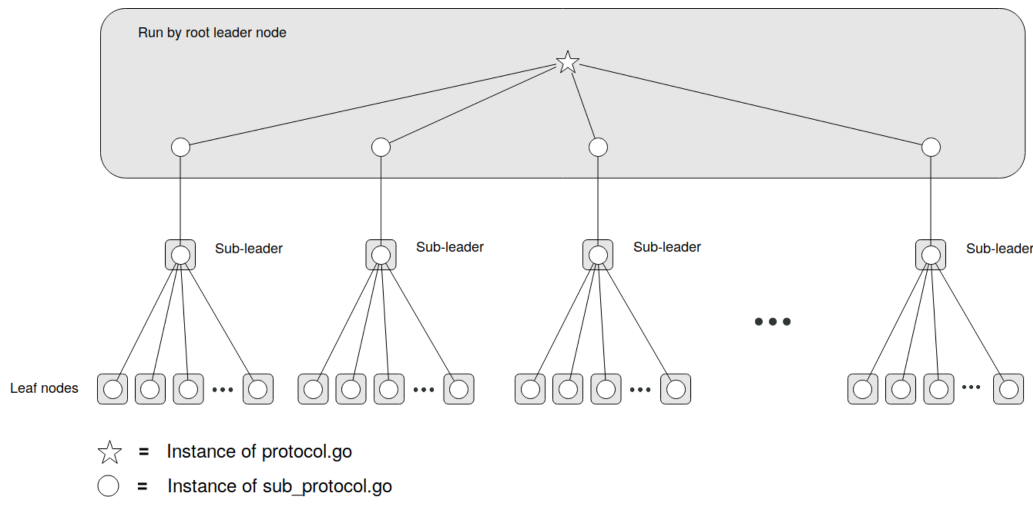

FIGURE 4.2: Instances run by each node.

4.1.3 Bitmask

To keep track of which node has sent a valid signature, we use bitmasks, i.e. arrays of boolean flags. When a subleader receives the signature of a child, he verifies it then sets the bitmask value of that child to 1. For childs that do not send correct signature or simply do not send a signature, the bitmask value will be 0. This allows us to know which node has participated in the final signature for the verification step.

4.1.4 Timeouts

Because any node can be non-responsive, we use timeouts at all levels of the protocol, in case a node is Byzantine or fails. If a node does not respond to a request after the amount of time defined by the timeout, it is considered to have failed, and the appropriate measures are taken. We use a global default timeout at the root leader level, and for subleader, that timeout is divided by two. Our experiments show that a root timeout of 60 seconds is good for 600 nodes and a 1 MB message size.

4.1.5 Files

struct.go

Defines the different messages that are sent across the network and their content.

protocol.go

This is the protocol run by the root leader. It is the entrance point of the whole protocol.

sub_protocol.go

Every node runs one instance of *sub_protocol.go*. As stated in 4.1.2, the root leader runs a number of instances of *sub_protocol.go* equal to the number of subtrees.

In a few words, this file tells the node to send the message signature request to all its children (if any), wait for their responses, aggregate all answers, then send the result to its parent.

helper_functions.go

Contains the logic for aggregating signatures, verifying them, and other utility methods.

mask.go

Defines the Mask object and its functions. Is used for the bitmask: creating a bitmask, setting values to 0 or 1, etc.

gen_tree.go

Generates the tree structure that is needed for Bls-ByzCoinX: a three-level structure with the root leader as root, from the list of all nodes.

4.2 PBFT

For evaluating Bls-ByzCoinX, we wrote a PBFT implementation in Go using the same framework for fair comparison.

PBFT has two round-trips with four messages, as defined in 2.1. At every step, a node sends the message to all other nodes. On the receiving side, a node must wait for $\lceil 2/3 + 1 \rceil$ responses. Once it has received that amount of messages, it confirms that step and continues the protocol.

At every step, the messages are digitaly signed to avoid spoofing. The receiving node can verify the correct identity of the sender.

We use Schnorr signatures for creating the signature.

4.3 ByzCoin

We use the ByzCoin implementation from DEDIS, as described in the original paper.

We use the deep tree structure instead of the flat one, because it scales better, which compares better to Bls-ByzCoinX.

5 Evaluation

In this section, we evaluate our Bls-ByzCoinX implementation with respect to network and consensus latency, transaction throughput and overall scalability. We compare it to a PBFT and a ByzCoin implementation and discuss the differences.

We ran simulations using a cluster of 35 physical machines and up to 980 virtual nodes. To keep a constant bandwidth across all simulations with different number of virtual nodes, we modified the physical machines' bandwidth in order to keep a constant 35 Mbps for every experiment. We also set a simulated round-trip latency of 200ms between each machine. These values are meant to mimick real-world conditions.

As messages to be signed, we use real Bitcoin blocks containing Bitcoin transactions. This simulates once again real-world situations and allows us to compute the throughput of the system and compare it to Bitcoin. The maximum Bitcoin block size is 1 MB, which is equivalent to around 3000 transactions.

All experiments were run on 10 rounds and we took the mean values for our results.

5.1 Number of nodes

First, we explore the scalability of Bls-ByzCoinX in terms of the number of nodes participating in the consensus, and compare it to ByzCoin and PBFT.

We set the block size to 1 MB, which is equal to Bitcoin's current limit and vary the number of validators, ranging from 5 to 980 for each protocol. As explained above, the bandwidth of the machines were modified for each number of nodes to keep a constant 35 Mbps per simulated node.

For 5 nodes, PBFT has the smallest latency of the tree protocols, but when scaling it quickly becomes unusable. It started failing at 105 nodes, so we display its line to 100 nodes only. With it's $O(n^2)$ communication complexity, scalability is not possible and thus it cannot be used for large consensus groups in scalable ledgers.

Bls-ByzCoinX's latency augments linearly with the consensus group size, at least up to 980 nodes. Up to 600 nodes, we deem the latency acceptable and in range with other solutions (20-30 seconds).

As we can observe in figure 5.1, Bls-ByzCoinX has a slightly higher latency than ByzCoin over the full range of total validators. This is because Bls-ByzCoinX's root node neads to send the message to more nodes than ByzCoin, which has a smaller branching factor with a deeper tree. The difference accentuates as the

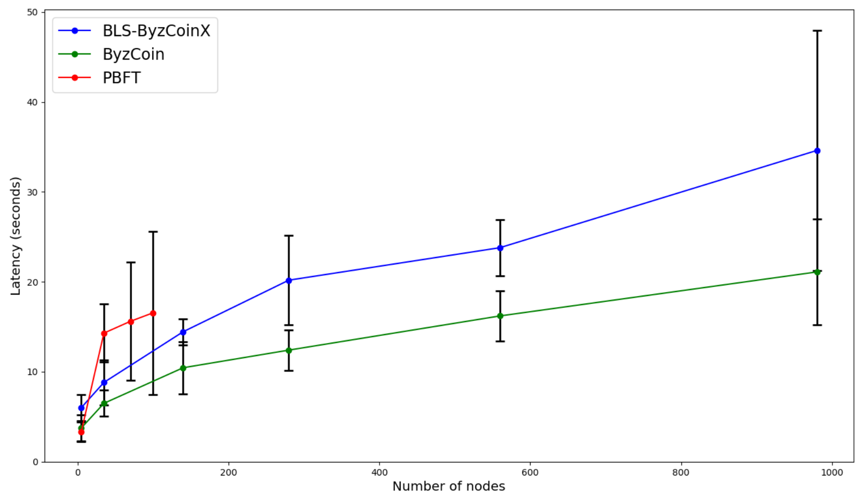

FIGURE 5.1: Influence of the number of validators on the latency for the three protocols.

number of nodes grows, which is expected because the tree in Bls-ByzCoinXwill grow large faster than ByzCoin.

5.2 Block size

Next, we experiment how the block size affects the system's latency.

Block size is an important factor for distributed ledgers. A greater block size allows more transactions to be logged per epoch, but induces longer latency when the block has to be propagated between nodes during the consensus.

We vary the block size betwen 1, 2, and 4 MB. For 8 MB and higher, our implementation of Bls-ByzCoinX started failing quickly because of timeouts, so we kept to smaller sizes. As a reminder, Bitcoin uses a 1 MB block size limit.

As we can observe in figure 5.2, a classic 1 MB block achieves latency that is in an acceptable order of magnitude (below 30 seconds) and scales linearly.

Doubling the block size to 2 MB nearly doubles the latency across the range of number of validators. The latency for 560 validators stays acceptable but more nodes start to be out of range of desired values.

A 4 MB block also grows linearly with the number of validators, and has latencies that are approximately two times greater than the 2 MB block.

Thus, we conclude that the latency scales proportionally to the block size.

5.3 Throughput

We ran an experiment to determine the maximum throughput in transactions per second of Bls-ByzCoinX.

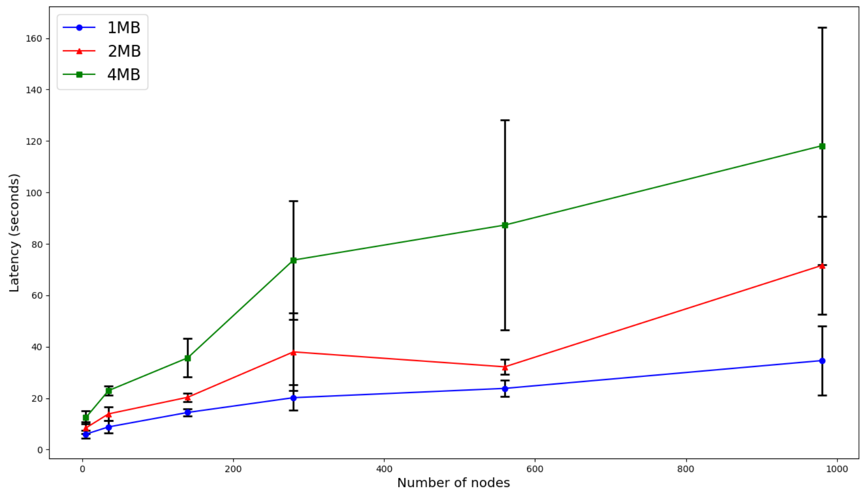

FIGURE 5.2: Block size influence on latency in Bls-ByzCoinX

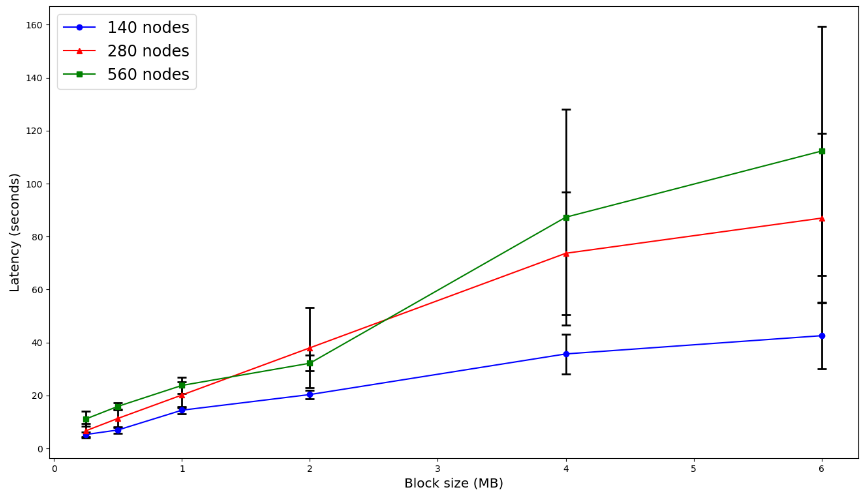

FIGURE 5.3: Latency in Bls-ByzCoinX for different block sizes.

We fixed the number of nodes in the consensus group to three different values turn by turn: 140, 280 and 560, and experimented with different block sizes.

From 8 MB and up, the protocol timed out so we stick with block sizes ranging from 0.25 MB to 6 MB.

As figure 5.2 shows, the latency grows linearly with the block size. For a 1 MB block (Bitcoin's block size), we have a latency of around 20 seconds for each consensus group size. For 140 nodes and a 6 times bigger block, the latency is 2 times higher at 40 seconds. For 280 nodes and a 6 times bigger block it is 4 times higher at 80 seconds, and for 560 nodes and the same size multiplier we get a latency that is a bit less than 6 times greater. We observe that for a typicaly sharding ledger consensus group size, a bigger block size does not scale very well. A 1 MB block yields good enough results but it is difficult to use a bigger block size without compromising on latency.

We can compute the throughput of these runs by knowing the number of transactions each block size contained. We can then see what latencies are achievable for different throughputs. Results are presented in figure 5.4.

We see that for high number of nodes, the throughput hardly goes above 200 transactions per second.

A more modest number of nodes (140) will be able to achieve higher throughput. This suggests that a sharding blockchain using Bls-ByzCoinX should either stick with 1 MB block sizes or use less nodes than 600 per shard, to be able to acheieve better throughput.

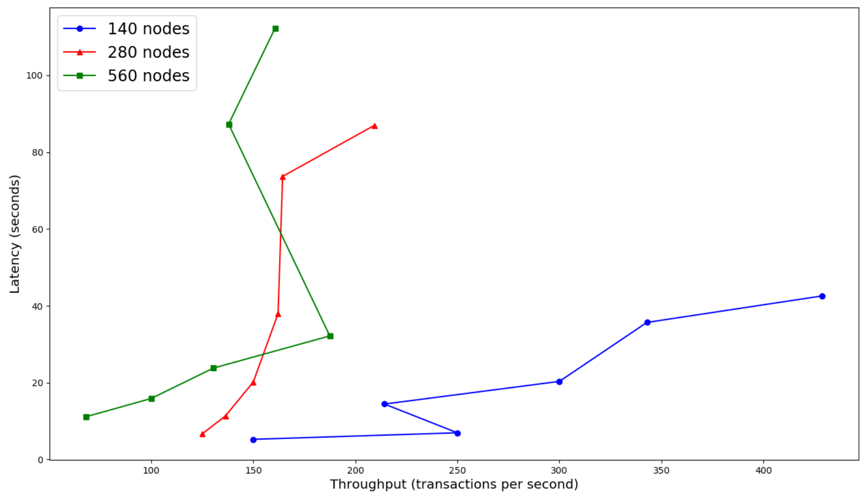

FIGURE 5.4: Latency for different throughputs.

6 Conclusion

In this project we have developed a fault tolerant consensus protocol that handles up to $\lceil 2/3 + 1 \rceil$ failures and scales out to 600 nodes with a latency of 20-30 seconds using a 1 MB block, making it suitable for decentralized ledgers that use sharding for scalability, while guaranteeing security and staying permissionless.

It offers strong consistency, unlike Bitcoin's weak consistency, meaning that within a 20-30 seconds latency from transaction input, we can know that it is confirmed, which makes it suitable for near real-time transactions.

Bls-ByzCoinX makes use of the BLS signature scheme and a three-level tree structure to mitigate the Denial-of-Service attack that is possible in ByzCoin, making it more robust to Byzantine nodes and malicious adversaries that want to slow the system down.

We implemented the solution and tested it out on a cluster of machines to show and confirm its capabilities. While having a greater latency than ByzCoin, it stays at most two times slower than it for any number of nodes, which stays acceptable.

Acknowledgements

I want to express my gratitude to all the people who helped me during this project, for their technical advice, emotional support or any other help they brought to me. A special thanks to:

Lefteris Kokoris Kogias, my supervisor, who gave me the main guidance and followed through the project every week. He made himself available even on week-end days to help me.

Kelong Kong, for helping me understand and fix a lot of the bugs I encountered during the project.

Linus Gasser, for explaining to me the frameworks for launching and testing my protocol.

Steve from Deterlab, for his reactivity in making the machines available for the simulations.

All other students from the office, who made the semester enjoyable and also discussed ideas with me.

Bibliography

[1] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short Signatures from the Weil Pairing". In: *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*. ASIACRYPT '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 514–532. ISBN: 3-540-42987-5. URL: `http://dl.acm.org/citation.cfm?id=647097.717005`.

[2] Justin Cappos and John H. Hartman. "San Fermín: Aggregating Large Data Sets Using a Binomial Swap Forest." In: *NSDI*. Ed. by Jon Crowcroft and Michael Dahlin. USENIX Association, 2008, pp. 147–160. ISBN: 978-1-931971-58-4.

[3] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186. ISBN: 1-880446-39-1. URL: `http://dl.acm.org/citation.cfm?id=296806.296824`.

[4] Miguel Castro et al. "SplitStream: High-bandwidth Multicast in Cooperative Environments". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 298–313. ISSN: 0163-5980. DOI: `10.1145/1165389.945474`.

[5] *Cost of a 51% attack @ONLINE*. Accessed: 2018. URL: `https://gobitcoin.io/tools/cost-51-attack/`.

[6] James Cowling et al. "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance". In: *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*. Seattle, Washington, Nov. 2006.

[7] Kyle Croman et al. "On Scaling Decentralized Blockchains - (A Position Paper)". In: *Financial Cryptography Workshops*. 2016.

[8] Rachid Guerraoui et al. "The Next 700 BFT Protocols". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 363–376. ISBN: 978-1-60558-577-2.

[9] Steve Huckle et al. "Internet of Things, Blockchain and Shared Economy Applications". In: *Procedia Computer Science* 98 (2016). The 7th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2016)/The 6th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2016)/Affiliated Workshops, pp. 461 –466. ISSN: 1877-0509.

[10] Viktor Jacynycz et al. "Betfunding: A Distributed Bounty-Based Crowdfunding Platform over Ethereum". In: *Distributed Computing and Artificial Intelligence, 13th International Conference*. Ed. by Sigeru Omatu et al. Cham: Springer International Publishing, 2016, pp. 403–411. ISBN: 978-3-319-40162-1.

[11] Ghassan Karame. "On the Security and Scalability of Bitcoin's Blockchain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 1861–1862. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2976756.

[12] E. Kokoris-Kogias et al. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: *2018 2018 IEEE Symposium on Security and Privacy (SP)*. Vol. 00, pp. 19–34. DOI: 10.1109/SP.2018.000-5. URL: doi.ieeecomputersociety.org/10.1109/SP.2018.000-5.

[13] Eleftherios Kokoris-Kogias et al. "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing". In: *CoRR* abs/1602.06997 (2016). arXiv: 1602.06997. URL: http://arxiv.org/abs/1602.06997.

[14] Eleftherios Kokoris Kogias et al. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: (2018), p. 16.

[15] R. Kotla et al. "Zyzzyva: Speculative Byzantine Fault Tolerance". In: *Communications of the ACM* (Nov. 2008).

[16] Tsung-Ting Kuo, Hyeon-Eui Kim, and Lucila Ohno-Machado. "Blockchain distributed ledger technologies for biomedical and health care applications". In: *Journal of the American Medical Informatics Association* 24.6 (2017), pp. 1211–1220. eprint: /oup/backfile/content_public/journal/jamia/24/6/10.1093_jamia_ocx068/1/ocx068.pdf.

[17] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

[18] K. J. O'Dwyer and D. Malone. "Bitcoin mining and its energy footprint". In: *25th IET Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014)*. 2014, pp. 280–285. DOI: 10.1049/cp.2014.0699.

[19] *Safety and Liveness Properties @ONLINE*. Accessed: 2018. URL: https://pdfs.semanticscholar.org/presentation/114f/db6fa1051d2ab60f7d80ba72191c1210659f.pdf.

[20] *Stress Test Prepares VisaNet for the Most Wonderful Time of the Year, VISA @ONLINE*. Accessed: 2018. URL: https://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html.

[21] Ewa Syta et al. "Decentralizing Authorities into Scalable Strongest-Link Cothorities". In: *CoRR* abs/1503.08768 (2015). arXiv: 1503.08768.

[22] *The Bitcoin Wiki, Mining @ONLINE*. Accessed: 2018. URL: https://en.bitcoin.it/wiki/Mining.

[23] Wikipedia contributors. *Byzantine fault tolerance — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-May-2018]. 2018.

[24] *Wikipedia, The Bitcoin Scalability Problem @ONLINE*. Accessed: 2018. URL: https://en.wikipedia.org/wiki/Bitcoin_scalability_problem.