RZ 3910 (#ZUR1704-014) 04/04/2017
Computer Science 42 pages

Research Report

Threshold Signatures for Blockchain Systems

C. Stathakopoulou‡, C. Cachin*

‡Swiss Federal Institute of Technology
Zurich

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

Research
Africa • Almaden • Austin • Australia • Brazil • China • Haifa • India • Ireland • Tokyo • Watson • Zurich

Threshold Signatures for Blockchain Systems

Chrysoula Stathakopoulou cstathak@student.ethz.ch

Christian Cachin cca@zurich.ibm.com

IBM Research - Zurich

4 April 2017

Abstract

Blockchain, introduced as the backbone of the Bitcoin cryptocurrency, is an emerging technology. Abstracting the currency logic away opens Blockchain to endless applications from finance to healthcare and Internet of Things. Asymmetric cryptography and more specifically digital signatures are a key component of the blockchain system. In this work, we introduce threshold signatures for the Hyperledger Fabric, an implementation of a permissioned blockchain system. Threshold signatures enhance the resilience and robustness of the system while preserving the distributed nature of the Blockchain. In particular: (1) we implement two threshold signature schemes, (2) evaluate their performance, (3) discuss their advantages and limitations and (4) introduce a Framework to hide the implementation details and make threshold signatures available to any potential application in Hyperledger Fabric.

Acknowledgment

This report documents the results of the M.Sc. thesis project of Chrysoula Stathakopoulou as a student at D-ITET of ETH Zurich, carried out in collaboration with IBM Research - Zurich. We thank Prof. Roger Wattenhofer of the Distributed Computing Group (DCG) at the Computer Engineering and Networks Laboratory (TIK) of ETH Zurich for supporting this work.

We would like to thank Elli Androulaki, Angelo De Caro, Andreas Kind, Alessandro Sorniotti, Marko Vukolić and the whole blockchain team for their feedback and support. Many thanks also to Manu Drijvers for his help with the AMCL library and the technical support.

This work was supported in part by the European Commission through the Horizon 2020 Framework Programme (H2020-ICT-2014-1) under grant agreement number 643964 SUPERCLOUD and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the Swiss Government.

Contents

Chapter 1

Introduction

1.1 Motivation

In this work, we introduce threshold signatures for the Hyperledger Fabric (HLF) blockchain system. As we see in detail in the following chapters, signatures are an indispensable component of blockchain systems. Our goal is to provide a group of participants with the functionality to generate a single fault tolerant signature.
Distributing trust is not a new idea. Shamir [20] already in 1979 argues about the necessity of threshold cryptography for key management. He indicates that storing a key in a single location is not robust, while keeping multiple copies of the same key introduces security breaches. Also, he points out that no single entity should be trusted to keep a company's secret signature key.
Reiter and Birman [18] introduce threshold cryptography to replicate a service in a way that it remains available, correct and maintains causality of the requests even if several replicas are corrupted. Threshold cryptography allows the client to maintain only one public key for the service, instead of one public key for each replica. In this way, the client needs no more storage and computational cost than in the case of a non-replicated service.
Based on the aforementioned idea of Reiter and Birman, Cachin [2] describes an architecture for distributing trusted services in an asynchronous environment such as the Internet and suggests real world applications. In detail, he suggests that certification authorities (CA) and secure directories, which are used for example for DNS authentication, are both examples of services that can be distributed. Another example given is a digital notary.
The advantages of threshold signatures for DNS are further addressed by Cachin and Samar [5]. DNS Security Extensions (DNSSEC) use a technique called zone signing to provide authentication. However, the private key for signing the zone must be stored somewhere. Having, a single entity and subsequently a single private key introduces reliability and security issues. To tackle this issue, they use a threshold RSA signature scheme to securely replicate the authoritative servers.
Another use case of threshold cryptography for digital signatures is the COCA system by Zhou et al. [25]. They suggest a certification authority for online certificate validation in an asynchronous communication model and they use replication to achieve availability.
As the aforementioned literature suggests, threshold cryptography is a powerful tool that has been widely explored for service replication. We believe that the time has come to introduce threshold cryptography in blockchain systems. Goldfeder et al. [12] already discuss the advantages of a threshold signature scheme for Bitcoin. In this work we implement threshold signature schemes suitable for an asynchronous distributed blockchain system, such as the HLF.

1.2 Contribution

The primary contribution of this work is an implementation in Go language of a threshold signature library for HLF. Two different schemes are implemented as options: a threshold RSA scheme [21] and

a threshold version of the BLS signatures [1]. The schemes are benchmarked and compared in terms of efficiency.
The library is integrated in the Hyperledger Fabric project[1] and, as an application, it is used in the core of the transaction endorsement procedure.

1.3 Overview

The rest of this paper is organized as follows. In Chapter 2 a background regarding the blockchain systems and threshold cryptography is given. In the Chapter 3 the problem is introduced and the suggested solution is outlined. In Chapter 4 the implementation of the cryptographic libraries and their use in the blockchain fabric is discussed. In Chapter 5 results of benchmarks on the two cryptographic libraries are presented. In Chapter 6 this work is concluded.

[1] `https://github.com/hyperledger/fabric`

Chapter 2

Background

2.1 Blockchain systems

A blockchain system consists of three key components: (1) a public ledger, (2) a consensus algorithm, and (3) a smart contract which is the application that runs on top of the blockchain. This architecture allows participants to engage into transactions without trusting each other and without revealing their identity.

The public ledger consists of a chain of blocks, as the name Blockchain suggests. The blocks are a log of the transactions performed by the participating in the blockchain entities. Each block contains a cryptographic hash of the previously created block, hence constructing a conceptual chain. This chain provides the ordering of the transactions and guaranties causality. If somebody were to perform the same transaction twice, a problem known as a double spending attack, they would have to change all the blocks on the chain back to the block where the transaction appears for the first time. The ledger however is replicated and each of the participants of the system has its own replica. Therefore, an effort to manipulate the ledger would be detected. In the same sense, if somebody were to claim that a transaction has not been performed, or that the order of the transactions were different, they would have to forge the ledger, resulting in a different view than the one the legitimate participants maintain and, hence, be detected. In conclusion, the ledger is immutable and one can only append new blocks of transactions.

From the examples above it becomes evident that the participants need to maintain a synchronized view of the ledger. This is why the second component, the consensus algorithm, is needed. The consensus algorithm should guarantee the ordering of the transactions on the replicas of the ledger on each of the participants. For the blockchain system to be secure under realistic assumptions, the consensus algorithm should be able to guarantee the ordering of the transactions in an asynchronous and Byzantine environment. The latter means that an arbitrary number of participants can fail or misbehave. There is a known upper bound to the number of misbehaving participants. Strictly less than $\frac{1}{3}$ of the nodes of the network can have such a behavior while the consensus is still possible.

Finally, smart contracts provide the business logic of the Blockchain. Szabo first introduced the idea of smart contracts [23] and Wood redefined it in the context of a blockchain for Ethereum project [24]. They allow two or more participants to enforce an agreement in the form of application code that is deployed over the blockchain. A cryptocurrency exchange is a special case of such a contract.

2.1.1 Hyperledger Fabric

Hyperledger Fabric (HLF) is an implementation of a permissioned blockchain system for running smart contracts. Fabric is developed under the umbrella of the Hyperledger Project[1], an open source collaborative effort to create an enterprise, cross-industry blockchain system.

[1] `https://www.hyperledger.org`

The Hyperledger Fabric provides the infrastructure for performing transactions among mistrusting parties, which are logged in the immutable distributed ledger. There are three types of transactions:

- Deployment transactions initialize the smart contract and install it on the blockchain. We will refer to smart contracts hereinafter using the term chaincode.
- Invocation of the chaincode is the execution of the application code.
- Query transactions on the distributed ledger read the current state related to a certain chaincode.

To maintain the same view of the ledger on every participating node HLF uses an ordering service. It runs on an independent set of nodes for scalability, called *orderers*. The orderers run a deterministic consensus protocol such as PBFT [6] or Paxos[15].
Finally, as mentioned above, Hyperledger Fabric is a permissioned blockchain. To this effect a membership service provider is implemented as a component of the Fabric. An instance of a membership service provider is responsible for issuing and validating certificates. Each organization that constitutes a stakeholder of the blockchain should have its own membership service so as to authorize its members to submit transactions associated with certain chaincodes.

2.1.2 Other Blockchain Systems

Bitcoin [17] is undoubtedly the most popular blockchain system. However, Bitcoin is fundamentally different from the blockchain systems we examine in this work. It is a permissionless blockchain system, which means that anyone can participate. Instead we focus on permissioned blockchain systems as HLF. Moreover, Bitcoin is a cryptocurrency system, whereas we focus on general purpose blockchain systems, decoupled from the currency logic.
Ethereum [24], as mentioned in section 2.1 is such a general purpose blockchain system but, as Bitcoin, is permissionless. However, there are permissioned blockchain systems based on Ethereum such as Quorum[2]. Other examples of permissioned blockchain systems available today are Tendermint[3] and Kadena[4]. The concepts discussed in this work are also applicable to these permissioned blockchain systems.

2.2 Threshold Cryptography

As mentioned in the previous section, a blockchain system consists of three key components: the public ledger, the ordering mechanism and the smart contracts. Yet, there is another indispensable element that binds everything together and this element is cryptography and, more specifically, digital signatures. Digital signatures provide the blockchain system with integrity, pseudonymity, non-repudiation and authenticity. Integrity is guaranteed because if a signed message is modified, there is no efficient way to modify the signature so that it matches the message. Pseudonymity is provided since the participants do not need to disclose any personal information, instead they use a private cryptographic key to sign their transactions. Finally, an entity that has signed a transaction digitally cannot later deny doing so, while no other entity having access to the corresponding public key can fake a valid signature and therefore impersonate the original signer.
While digital signatures are being widely used in blockchain systems, such as Bitcoin and the current implementation of Hyperledger Fabric, there is a technology invented more than 20 years ago, which, as we will explore in detail in the following chapter, fits the needs of a blockchain system but has not been exploited so far to a great effect. And this technology is threshold cryptography.
Threshold cryptography, was introduced by Desmedt [9]. Let us assume a group of n parties $\mathbf{P} = \{P_1, P_2, \cdots, P_n\}$ up to t of which may be corrupted, n, t being fixed integers. $\mathbf{P}$ replaces the operation

[2] `https://www.jpmorgan.com/country/US/EN/Quorum`
[3] `https://tendermint.com/`
[4] `http://www.kadena.io`

of a cryptosystem $\mathcal{C}$, while maintaining the robustness and security properties of $\mathcal{C}$. We call such a system a $(t+1)$-out-of-n threshold cryptosystem. We will focus on asymmetric or public key cryptography and more specifically threshold digital signature schemes.
Digital signature scheme is a triple ($KeyGen$, $Sign$, Ver) of efficient algorithms.

- $KeyGen$ is the key generation algorithm. It outputs a key pair (P, S). P is the public key and S the secret or private key.
- $Sign$ is the signing algorithm. Given a message μ and the secret key S, it outputs a digital signature σ.
- Ver is the verification algorithm. Given a message μ, the corresponding signature σ and the public key P, it succeeds if σ is a valid signature of the message μ.

A digital signature scheme must have the two following security properties:

- The authenticity of a digital signature σ generated by a secret key S must be able to be verified by the corresponding public key P.
- It must be computationally infeasible for an adversary to generate a valid signature σ without knowing the secret key S that generates the signature.

For a threshold signature scheme we have n, t as before. For a signature reconstruction at least k valid signature shares are required. The number k is not necessarily equal to $t+1$. Instead n, k and t must satisfy the following more general requirement.

$$t < k \leq n - t \tag{2.1}$$

This scheme is defined in [4] as an (n, k, t) dual-threshold signature scheme.
We will study non-interactive threshold signature schemes. This means that upon a signature request of a message μ, each signing party independently calculates a signature share and the client that requests the signature of the message μ must construct it by combining the signature shares. These schemes are comprised of the following algorithms.

- A key generation algorithm $ThresKeyGen$ that generates a key pair (P, S) and a set of n secret key shares $\mathbf{S} = \{S_1, S_2, \cdots, S_n\}$. Moreover, it generates a set of verification keys $\mathbf{V}$ that is specific to the particular scheme.
- A signing algorithm $ThresSig$ that, given a message μ and a secret key share S_i, outputs a signature share σ_i.
- A signature share verification algorithm $SigShareVer$, that given a message μ, a signature share σ_i, the public key P, and the appropriate for the scheme verification key V, succeeds if σ_i is a valid signature share of the i^{th} party for the message μ.
- A share combination algorithm $SigShareComb$ that given a set of at least k valid signature shares $\{\sigma_{i_1}, \sigma_{i_2}, \cdots, \sigma_{i_k}\}$ outputs the signature σ.
- A signature verification algorithm Ver equivalent to the one for the non-threshold scheme.

The security properties that a digital signature scheme must satisfy are extended for the threshold signature schemes as follows:

- The authenticity of a digital signature σ generated by a set of k secret keys shares $S_1, \cdots, S_k$ must be able to be verified by the corresponding public key P of the scheme. The authenticity of a digital signature share σ_i generated by a secret key share S_i must be able to be verified by the corresponding verification key V_i. A combination of k valid signature shares $\{\sigma_{i_1}, \cdots, \sigma_{i_k}\}$ must produce a valid signature σ.

- It must be computationally infeasible for an adversary to generate a valid signature without submitting a signing request to at least $k - f$ uncorrupted signing participants.

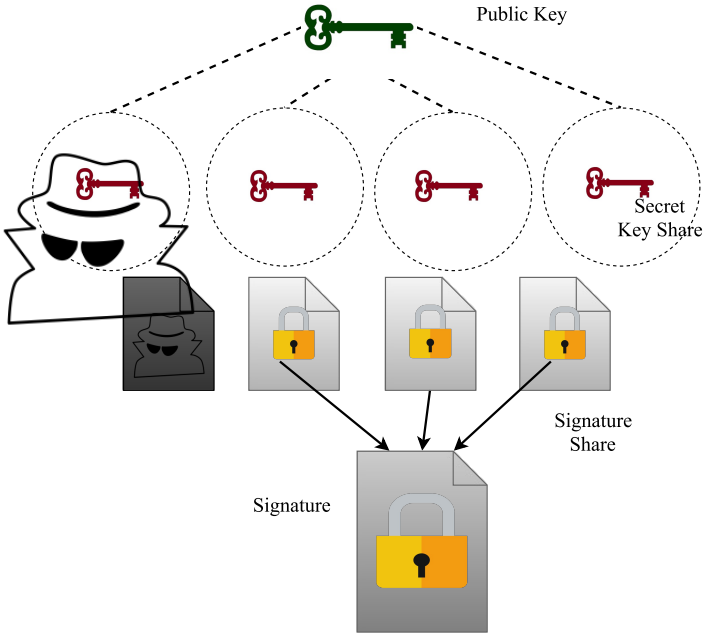

Figure 2.1. Threshold signature construction from signature shares.

We must point out that for the schemes we study in this work the verification algorithm is the same for the threshold and corresponding non-threshold scheme since the signature generated by the $SigShareComb$ algorithm is the same as the one generated by the Sig algorithm of the non-threshold scheme with input the secret key S and the same message μ. Also, the resulting signature is the same regardless of the combination of valid signature shares. These nice properties are required for the use of the threshold signatures the HLF.

Let us now have a look at a $(t+1)$-out-of-n secret sharing scheme, which is in the core of the $SigShareComb$ algorithm. The goal is to share a secret s, element of a finite filed $\mathbb{F}_q$, among n parties $\mathbf{P}$ in such a way that at least $t+1$, or, generally, some $k > t$ parties are needed to reconstruct the secret and any group of t or fewer parties cannot infer any information about s.

Shamir Secret Sharing [20]: A trusted dealer $D \notin \mathbf{D}$ chooses uniformly at random a polynomial $f(X)$ over $\mathbb{F}_q$ of degree t, such that $f(0) = s$. The dealer generates secret shares $s_i = f(i), i \in \{1, .., n\}$ and secretly sends s_i to P_i. The secret s can be reconstructed from $t+1$ shares with indeces in the set $\mathcal{S} \subset \{1, \cdots, n\}$ using polynomial interpolation:

$$s = f(0) = \sum_{i \in \mathcal{S}} \lambda_{0,i}^{\mathcal{S}} s_i \tag{2.2}$$

where

$$\lambda_{0,i}^{\mathcal{S}} = \prod_{j \in \mathcal{S}, j \neq i} \frac{j}{j-i} \tag{2.3}$$

are the Lagrange coefficients. Dishonest parties may contribute incorrect shares at reconstruction time. The scheme is, however, robust since we can exclude the corrupted shares using the share verification algorithm.

2.2.1 Threshold RSA Signatures

In this section, we present the RSA based threshold signature scheme proposed by Shoup [21]. The security proof is based on the random oracle model.

The actors are, as before, a group of n parties $\mathbf{P} = \{P_1, P_2, \cdots, P_n\}$ up to t of which may be corrupted and a trusted dealer $D \notin \mathbf{P}$.

Key Generation

The dealer chooses $P = 2p + 1$, $Q = 2q + 1$ large prime numbers of equal length such that p, q are also prime numbers. Such numbers as P, Q are called *safe prime* numbers. The RSA modulus is calculated as $N = PQ$. A prime number for the public key exponent e is chosen, such that $e > n$. The public key is then the pair (N, e). The secret key d is computed, such that $de = 1 \mod m$. The dealer then constructs a secret polynomial $f(X) = \sum_{i=0}^{t} a_i X^i \in \mathbb{Z}[X]$ such that $f(0) = d$ by choosing the coefficients a_i uniformly at random from $\{0, \cdots, m-1\}$, $m = pq$. The secret key shares are

$$s_i = f(i)\Delta^{-1} \mod m,\ i \in \{1, \cdots, n\} \tag{2.4}$$

where $\Delta = n!$. The dealer also generates a group of verification keys that accompanies the public key. The global verification key v is a value chosen at random in Q_n, where $Q_n = \mathbb{Z}_m \times \mathbb{Z}_2 \times \mathbb{Z}_2$ is the subgroup of squares in $\mathbb{Z}_m^*$ of order m. The local verification keys are calculated as $v_i = v_i^s \in Q_n$. Finally, a random value $u \in \mathbb{Z}_n^*$ with Jacobi symbol $(u \mid n) = -1$ is added to the verification key.

Signature Share Generation

Given a hash function H, a message μ and the random element $u \in \mathbb{Z}_n^*$ with Jacobi symbol $(u \mid N) = -1$ the message is hashed as follows

$$x = \mathcal{H}(\mu) = \begin{cases} \hat{x} & \text{if } (\hat{x} \mid N) = 1 \\ \hat{x}u^e & \text{if } (\hat{x} \mid N) = -1 \end{cases} \tag{2.5}$$

where $\hat{x} = H(\mu)$. Each party P_i can generate a signature share as

$$\sigma_i = x^{2s_i} \in Q_n \tag{2.6}$$

The signature share must be accompanied by a proof of correctness for the signature share verification. To that end the paper adopts a non-interactive version of ChaumFLs and PedersenFLs protocol [8]. Shortly, in [8] the prover chooses a random $r \in \mathbb{Z}_q$, where q is here the prime order of the group G_q, generated by $g \in G_q$ and g is part of the public key. Then, the prover sends $(a, b) = (g^r, x^r)$ to the verifier. The verifier chooses a random challenge c and sends it to the prover and the prover sends back the $z = r + cs$, where s is the secret key.
To collapse this communication step and make the protocol non-interactive, Shoup uses a hash function H' with output of length L_1, where L_1 is a secondary security parameter. The party P_i (prover) choses uniformly at random a number $r \in \{0, \cdots, 2^{L(N)+2L_1}\}$, where $L(N)$ is the length of the RSA modulus. P_i calculates $v' = v^r$ and $x' = \tilde{x}^r$, as the equivalent of (a, b) in [8], where $\tilde{x} = x^4$, and gets the challenge from H'

$$c = H'(u, \tilde{x}, v_i, x_i^2, v', x') \tag{2.7}$$

and

$$z = s_i c + r \tag{2.8}$$

The proof of correctness that accompanies the signature share s_i is the tuple (c, z).

Signature Share Verification

The party that assembles the signature, in order to validate the signature shares, needs to prove that $log_{\tilde{x}}(x_i^2) = log_v(v_i)$. To do that it verifies that

$$c = H'\Big(u, \tilde{x}, v_i, x_i^2, \frac{v^z}{v_i^c}, \frac{x^z}{x_i^{2c}}\Big) \tag{2.9}$$

Notice that z is of length $L(N) + 2L_1$ and therefore the exponentiation to the power of z in 2.9 is an expensive operation, the repercussions of which we will examine in chapter 4.

Signature Share Combination

To get the signature σ we need to collect at least k valid signature shares, where k needs to satisfy 2.1. As before, x represents the hash of the signed message and $\mathcal{S} = \{i_1, i_2, \cdots, i_k\} \subset \{1, 2, \cdots, n\}$ is the set of the indeces of k collected valid signature shares. We compute

$$w = \prod_{j \in \mathcal{S}} \sigma_j^{2\lambda_{0,j}^{\mathcal{S}}} \tag{2.10}$$

where

$$\lambda_{i,j}^{S} = \Delta \frac{\prod_{j' \in \mathcal{S} \setminus \{j\}} (i - j')}{\prod_{j' \in \mathcal{S} \setminus \{j\}} (j - j')} \tag{2.11}$$

and because of $\Delta = n!$ the $\lambda_{i,j}^{S}$ coefficients are always integers. From the Lagrange interpolation we have that

$$d = f(0) = \sum_{j \in \mathcal{S}} \lambda_{0,j}^{\mathcal{S}} s_j \mod m \tag{2.12}$$

and from eq. 2.6, 2.10, 2.12 it follows that

$$w^e = \big(\prod_{j \in \mathcal{S}} x^{2 s_j 2\lambda_{0,j}^{\mathcal{S}}} \big)^e = x^{4e \sum_{j \in \mathcal{S}} \lambda_{0,j}^{\mathcal{S}} s_j} = x^{4ed} = x^4$$

We now want to compute the signature σ such that $\sigma^e = x$. Since e is a prime and thus $gcd(4, e) = 1$, from the Extended Euclidean Algorithm we can compute a, b such that $4a + eb = 1$ and therefore $\sigma = w^a x^b$.

Signature Verification

The signature verification is simple and equivalent to the verification of a standard RSA signature. For $\mathcal{H}, \mu$ as before the verifier must verify that $\sigma^e = \mathcal{H}(\mu)$.

2.2.2 Threshold BLS Signatures

The second scheme we implement was proposed by Boneh et al. [1]. The main contribution of their work is a signature scheme with considerably shorter signatures than, for instance, the RSA scheme, while maintaining the same level of security. The scheme assumes that the computational Diffie-Hellman problem is hard to solve on certain elliptic curves over a finite field but the decision Diffie-Hellman problem is easy.

In detail, we define G_1, G_2 as multiplicative cyclic groups of prime order p and g_1, g_2 fixed generators for G_1, G_2 respectively. Moreover, there exists an efficient isomorphism $\psi : G_2 \to G_1$ with $\psi(g_2) = g_1$. Finally, e is a bilinear map $e : G_1 \times G_2 \to G_T$, such that $|G_1| = |G_2| = |G_T|$. The map e is bilinear, i.e. for all $v \in G_1, u \in G_2$ and $a, b \in \mathbb{Z}$ $e(v^a, u^b) = e(v, u)^{ab}$, and non degenerate, i.e. $e(g_1, g_2) \neq 1$.

Before we can describe the signature scheme, we should discuss Gap co-Diffie-Hellman groups. For co-CDH (computational Diffie-Hellman) problem, given $g_2, g_2^a \in G_2$ and $h \in G_1$ as input, one must compute $h^a \in G_1$. For the co-DDH (decision Diffie-Hellman) problem, given $g_2, g_2^a \in G_2$ and $h, h^b \in G_1$ as input, one must output "yes" if $a = b$, "no" otherwise. When the answer is "yes" (g_2, g_2^a, h, h^b) is called a *co-Diffie-Hellman tuple*. A *gap co-Diffie-Hellman group pair* is a pair of groups (G_1, G_2) on which the co-DDH is easy but the co-CDH is hard.

Now we can present the BLS signature, based on elliptic curves.

Let $E/\mathbb{F}_q$ be an elliptic curve over the finite field $\mathbb{F}_q$ and P a point of prime order p, where p does not divide $q(q-1)$ and p^2 does not divide $|E(\mathbb{F}_q)|$. Let also $a > 1$ be a security multiplier and we assume that $a < p$. Then there exists a point Q linearly independent of P. With such points P,Q as generators we set $G_1 =< P >$, $G_2 =< Q >$. Then the *Weil Pairing* on the curve E gives a computable, non-degenerate bilinear map $e : G_1 \times G_2 \to \mathbb{F}_{q^a}^*$, which enables us to solve the co-DDH problem on the group pair (G_1, G_2). The Weil Pairing is described in [16].

Key Generation

We pick a random $s \in \mathbb{Z}_p$ and compute $V \leftarrow sQ$. The public key is $V \in E(\mathbb{F}_{q^a})$ and the private key is s.

Signature Generation

Given a private key $s \in \mathbb{Z}_p$ and a message μ, we compute $R \leftarrow \tilde{H}(m) \in G_1$ and $\sigma \leftarrow sR \in E(\mathbb{F}_q)$. $\tilde{H}$ is a hash function, as described in [1], that maps a message $\mu \in \{0,1\}^*$ to an element of the group G_1. In fact, we can output as a signature only the x-coordinate of σ which results in a signature of half the length. But in this case for the verification we must accept also the symmetric, with respect to x-axis, point.

Signature Verification

Given the public key $V \in G_2$, a message $\mu \in \{0,1\}^*$ and a signature $\sigma \in E(\mathbb{F}_q)$ we must verify if (Q, V, R, σ) is a co-Diffie-Hellman tuple, i.e. if $e(\sigma, Q) = e(R, V)$.
As suggested in [1], from the signature scheme described above, we can build a non-interactive threshold signature scheme based on Shamir secret sharing and polynomial interpolation, as with threshold RSA. The actors are a group of n parties $\mathbf{P} = \{P_1, P_2, \cdots, P_n\}$ up to t of which may be corrupted and a trusted dealer $D \notin \mathbf{P}$

Key Generation

The trusted dealer generates a public - secret key pair (s, V), $V = sQ$, where s is a random element in $\mathbb{Z}_p$ and $V \in G_2$, same as for the non-threshold scheme. Also, the dealer constructs a random polynomial $\alpha \in \mathbb{Z}_p$ of degree t, such that $\alpha(0) = s$. The secret key share is

$$s_i = \alpha(i), i = 1, \cdots, n \tag{2.13}$$

Also the dealer calculates n public key share values

$$V_i = s_i Q, i = 1, \cdots, n \tag{2.14}$$

that serve as verification keys.

Signature Share Generation

Given a message $\mu \in \{0,1\}^*$ and the secret key share $s_i \in \mathbb{Z}_p$, the message is mapped to G_1, as before, with a hash function $R \leftarrow \tilde{H}(\mu)$ and the signature share is calculated as

$$\sigma_i = s_i R \in G_1 \tag{2.15}$$

Signature Share Verification

The signature share verification is equivalent to the signature verification. Given a message $\mu \in \{0,1\}^*$, a signature share $\sigma_i \in G_1$ and the verification key $V_i \in G_2$ the prover must verify that $(Q, V_i, H(m), \sigma_i)$ is a co-Diffie-Hellman tuple. Q is the generator of the group G_2, which is a public parameter of the system, and $\tilde{H}$ is the hash function that maps the message μ to an element R in G_1. If the verification is successful, σ_i is a valid signature share of the party P_i.

Signature Share Combination

To get the signature σ, as in the threshold RSA scheme, we need to collect at least k valid signature shares, where k needs to satisfy eq. 2.1. Let $\mathcal{S} = \{i_1, i_2, \cdots, i_k\}$ be the set of indeces of the collected signature shares. Then the signature is reconstructed by the equation

$$\sigma = \prod_{i \in \mathcal{S}} \sigma_i^{\lambda_i} \tag{2.16}$$

where

$$\lambda_i = \frac{\prod_{j \in \mathcal{S} \setminus \{i\}} 0 - j}{\prod_{j \in \mathcal{S} \setminus \{i\}} i - j} \tag{2.17}$$

are the Lagrange coefficients.

Signature Verification

Given a message $\mu \in \{0,1\}^*$, the signature $\sigma \in G_1$ and the public key $V \in G_2$ the prover must verify that $(Q, V, \tilde{H}(\mu), \sigma)$ is a co-Diffie-Hellman tuple. Q is the generator of the group G_2, which is a public parameter of the system, and $\tilde{H}$ is the hash function that maps the message μ to an element R in G_1.

2.3 Related Work

Even though there is vivid academic research in blockchain technology and Bitcoin, introduced already in 2008 [17], and despite that threshold cryptography had been introduced even earlier, the applications of threshold cryptography in blockchain systems have only very recently started being explored.
Goldfeder et al. [12] introduce threshold signatures for Bitcoin. First, they suggest a threshold ECDSA scheme. They argue that if a company wants to perform a Bitcoin transaction, multiple employees must contribute to the signature instead of one. Whereas Bitcoin already supports multisignatures, they present the advantages in terms of flexibility, confidentiality, anonymity, and scalability that threshold signatures have. To allow more flexible access control policies they extend the key generation so that fresh keys can be derived from previous keys without revealing knowledge about the private key. They also discuss how to maintain accountability in the case of a shared wallet. Moreover, they suggest the use of threshold signatures for secure delegation. Finally, they suggest the use of threshold signatures for a two-step authentication so that the users do not need to store their private key in a single device.
Another notable contribution is the threshold signature scheme suggested by Gennaro et al. [10] for the enhancement of Bitcoin security. The authors point out that it is insecure to store the cryptographic key for authorizing transactions in a single location. Instead they suggest storing secret key shares in multiple of the userFLs devices and they propose a two-factor authentication scheme based on threshold signatures. The suggested signature scheme is a threshold version of the ECDSA signature scheme based on [12]. The scheme is presented as optimal, in the sense that it requires a minimum number of parties $n \geq t + 1$ to protect from an adversary that compromises up to t parties. However, the corrupted parties just try to learn information about the encryption scheme while complying with the protocol. If the participants are allowed to diverge from the protocol and generate invalid signature shares, the scheme requires $n \geq 2t+1$, same as the signature schemes described in the previous sections. Moreover, the scheme suggested in [10] is interactive and is executed in rounds and therefore introduces a communication overhead.
Another effort to leverage distributed signatures to enhance the security and performance of Bitcoin is the work of Kogias et al. [14]. They introduce ByzCoin, a cryptocurrency that replaces the proof-of-work used to reach a consensus in Bitcoin with a dynamic version of the PBFT protocol to achieve strong consistency. They use collective signing (CoSi) [22] to improve the scalability of PBFT. The actors in the collective signing scheme is an authority, which produces the signatures, and a group of witnesses that participate in the signing. The scheme is built on Schnorr signatures [19]. However, the

CoSi protocol is performed in rounds and therefore introduces communication cost. Also, CoSi does not specify a threshold of required co-signatures f above which the scheme is guaranteed to work correctly. More importantly, the authority that also acts as a leader for the protocol remains a single point of failure. The contributions presented above examine how to leverage distributed cryptography for Bitcoin. Regarding Bitcoin, the threshold ECDSA scheme by Gennaro et al. [10] has the advantage of compatibility, since the transaction can be indistinguishable from the non-threshold scheme. Instead this work focuses on a permissioned blockchain, Hyperledger Fabric. Fabric has a modular design and allows the user to choose the cryptographic scheme. Therefore, we don't need to stick to ECDSA. In the next chapter, we examine in depth the applications of threshold signatures for Hyperledger Fabric and justify the schemes we decided to implement.

Chapter 3

Threshold Signatures for the Blockchain

"All for one and one for all"
— Alexandre Dumas, *The Three Musketeers*

3.1 Problem Description

A single signing identity introduces a single point of failure. More importantly, the very nature of a blockchain system is decentralized and the key idea is the distribution of trust among the participants. A single signing identity cancels the notion of distributed trust.

A straightforward solution is to deploy multiple signing identities instead of just one and replace the digital signature with a group of signatures. The trust is now distributed since no signing identity can sign alone. However, this approach has several shortcomings. A validator must store the public keys of all the signing identities and verify each of the required signatures. Also, a transaction that is signed by a group of private keys has a length that grows linearly with the number of signatures. Therefore, the transmission time is linearly increased. Moreover, assuming n is the number of required signatures, instead of having one point of failure we now introduce n points of failure, since a missing or an invalid signature is enough to make the whole group invalid.
What we want is a distributed yet fault tolerant signature scheme. A single signature should be generated by a group of signing identities, a subset of whom we can tolerate to fail or be corrupted.

3.2 Solution

A threshold signature scheme is the solution to the aforementioned problem. As described in the previous chapter, for a k-out-of-n threshold signature scheme k participants are required to reconstruct a valid signature while the rest $n - k$ participants can fail to deliver any or can deliver invalid signature shares. Therefore, no single point of failure exists anymore and an attacker cannot submit a valid signature unless they compromise at least k nodes. The validator now needs to know only one public key and needs to verify only one signature. Moreover, the identity of the k signers who collaborated to reconstruct the signature cannot be inferred, unlike the case with the multiple signatures where each signature is linked with the corresponding public key of the signing identity.
We should point out that secret sharing, although it enhances the availability, is not a secure solution, despite what is claimed in the recent work of Zhou et al. [26]. In the secret sharing scenario, the signing identities would collaborate to reconstruct the private key from their private key shares and then use the private key to sign. But this means that at some point at least one of the participants would know the private key and they would be able to use it in the future to sign without the contribution of the rest of the participants.
We choose to implement two threshold signature schemes because each has distinct advantages.

The first scheme is a threshold version of the RSA signature scheme, as it is introduced by Shoup [21] and described in Section 2.2.1. This scheme has the following advantages:

- Non-interactive: No interaction among the signing parties is needed to generate the signature. Each signer generates a signature share independently and sends it to the node that needs to assemble all the signature shares to reconstruct the signature. This property requires a simple communication protocol; a unicast or broadcast functionality is enough to exchange the signature shares. Moreover, it makes the reconstruction more efficient, since the signing parties can generate and send their signature shares in parallel.
- Deterministic: The reconstructed signature is the same, regardless the of combination of signature shares that produced it.
- Equivalent to non-threshold RSA signature: The reconstructed signature is the same as the signature that would have been produced by the private key corresponding to the public key of the threshold signature scheme.

However, threshold RSA has some shortcomings:

- Long keys: Today the suggested length of an RSA key is 2048 bits[1].
- Trusted key dealer: A trusted third party is required to generate the key shares and distribute them to the signing parties.

The second scheme is a threshold version of the short signatures suggested by Boneh et al. [1], an elliptic curve pairing-based signature scheme, as described in Section 2.2.2. This scheme, same as the threshold RSA scheme is non-interactive and deterministic and the reconstructed signature is equivalent to the non-threshold BLS signature. Additionally, it comes with the following advantages:

- Short keys: As indicated in [1] 342 bits are enough to provide a security level equivalent to DSA using a 2048 bits prime, which is comparable to RSA with 2048-bit keys.
- Suitable for proactive security: In proactive sharing schemes, the secret shares can be refreshed in a way such that the secret key remains the same while at the same time an exposure of a share to an attacker does not compromise the system after the share has been refreshed. Proactive security not only enhances the security of the system but allows great flexibility in the access control policies. Cachin et al. [3] suggest a protocol for proactive cryptosystems for a discrete logarithm based scheme in asynchronous networks.

However:

- BLS signatures are not standardized and not widely used.
- Trusted key dealer: as with threshold RSA a trusted third party is needed for key share distribution. However, because BLS signatures are a discrete logarithm scheme, an algorithm could be implemented in the future, based on [11], so that the key share generation is also done in a distributed way. Kate and Goldberg [13] propose a realistic distributed key generation scheme for an asynchronous communication model.

3.3 Threshold Signatures for Hyperledger Fabric

Numerous potential applications of a threshold signature scheme exist within the Hyperledger Fabric. First, as described in Section 2.1.1 Hyperledger Fabric is a permissioned blockchain and comes with a membership service provider implementation. Each peer is associated with an identity, i.e. a certificate.

[1] `https://www.keylength.com/en/5/`

Currently, the certificate is signed by exactly one root Certificate Authority (CA). The root CA can be a commercial CA. However, the dependency on a single CA as a trusted third party contradicts the distributed nature that a blockchain system should have. Instead, a peer should be associated with a certificate signed by a group of CAs with a threshold signature, distributing, in this way, the trust.
Threshold signatures can be also used for Byzantine Consensus protocols. Let us assume a distributed system of n parties, t out of which can be faulty. The parties want to reach an agreement regarding a value. In such an algorithm, as explained in [7], there is a step where every party must gather $n - t$ messages with valid signatures. A widely-known example for an asynchronous system is the Practical Byzantine Fault Tolerance algorithm by Castro and Liskov [6]. This algorithm is implemented as an option for the ordering service of the Hyperledger Fabric. Similarly to the general case, before a peer commits a message, it must gather at least $t+1$ properly signed messages. In the current implementation, each peer signs their messages with their private key and, therefore, they must validate $t + 1$ signatures. A threshold signature scheme can be used instead. Each peer signs using their private key share and now they need to reconstruct and validate a single signature. An example of an algorithm that uses a threshold signature scheme to reach a Byzantine Agreement in an asynchronous environment is the work of Cachin et al. [4]. As a result, the validation procedure is accelerated by a factor of n.
Hyperledger Fabric should also provide threshold signatures as a service for chaincode applications. Multi-party computation, voting, distributed random number generation are examples of applications that can use threshold signatures in their core.
Last but not least, we introduce the use of threshold signatures for transaction validation in Hyperledger Fabric. Shortly, in Hyperledger Fabric version 1, before submitting their transactions for ordering, clients need to ask a set of special nodes, called *endorsers* to endorse their transaction, i.e. verify that it produces the expected result. The endorsers then sign the transaction and send it back. Only when an appropriate set of signatures is collected the transaction can be ordered. Currently, each endorser signs the endorsement with their private key. Instead, the endorsers can participate in a threshold signature scheme and produce a single signature for the endorsed transaction. This use case is discussed in detail in Section 4.3
Summarizing, there are various applications of threshold signatures for a permissioned blockchain system such as Hyperledger Fabric. To this end, the goal of this thesis is to provide a modular software framework that integrates the threshold signature functionality in the Hyperledger Fabric project. As a use case example the framework is used to replace traditional signatures with threshold signatures for the transaction endorsement.

Chapter 4

Implementing Threshold Signatures for Hyperledger Fabric

"Trust but verify."

— Ronald Reagan, *Russian proverb*

4.1 Threshold Signatures Framework Architecture

As discussed in the previous chapter, the multiple applications of threshold signatures in Hyperledger Fabric call for a modular and pluggable solution. The framework implemented as part of this work enables threshold signatures to be used anywhere in HLF. Let us now have a closer look at the implementation details.
In the core of the implementation we have two threshold signature libraries with the same interface: threshold RSA and threshold BLS. The libraries implement the algorithms described in Sections 2.2.1 and 2.2.2 respectively. Here we have a high level description of the main methods.

- `GenerateKeys`: Takes as arguments the number of signing participants, the number of signature shares required to reconstruct the signature, and the number of corrupted signature shares the system can tolerate. For the threshold RSA scheme it also takes as argument the RSA modulus size which is also the size of the signature. For the threshold BLS scheme this size is fixed. It generates the private and public keys of the scheme. In the public key structure the verification keys are also included, as well as the public information of the cryptosystem.

- `GenerateSignatureShare`: Takes as arguments a message and a secret key share and returns a serialized signature share. For the threshold RSA scheme the signature share, modeled with the `SignatureShare` structure, also includes the proof of correctness.

- `AssembleSignature`: Takes as arguments the message, an array of interpolation points, the total number of the signing participants of the system and the number of required signing participants. The interpolation point, modeled with the `InterpolationPoint` structure, includes the signature share and the ID of the signing participant. The method performs the polynomial interpolation and returns a signature.

- `VerifySignatureShare`: Given a message, the public key, which includes the verification keys, a signature share, and the ID of the signer, the method returns `true` if the signature share is valid.

- `VerifySignature`: Given a message, the public key, and a signature, the method returns `true` if the signature is valid.

For the threshold RSA scheme signature verification the standard `Go` RSA library[1] is imported. For the threshold BLS signature scheme, the non-threshold version is also implemented since there is no standard `Go` library. Regarding big integer arithmetic, for the threshold RSA library the standard `Go` library[2] is used. In threshold BLS library, big integer arithmetic and elliptic curve functionality suitable for pairing cryptography is imported from `amcl`[3] library.
For each scheme, we have an implementation of a `ThresholdSigner` interface that models the communication party. A `ThresholdSigner` could be instantiated by an endorser, a consenter, a submitter, or any other future sort of peer. This structure is initialized from a `yaml` configuration file. The `ThresholdSigner` instance is either responsible for generating signature shares, hence we will call it *signer*, or responsible for combining signature shares in a signature and/or verifying signatures, hence we will call it *verifier*. A *signer* configuration file must include the secret key share.
We want the threshold signature schemes to be imported as plugins and thus allow future integration of a different signature scheme. Also, the application that uses the threshold signature functionality should be agnostic of the scheme-specific methods that run under the hood. The `ThresholdSigner` interface serves exactly this purpose. A `controller` provides a constructor `NewThresholdSigner` that takes as parameters the name of the scheme the application wants to use and a configuration and returns the implementation of the interface that uses the corresponding cryptographic library. A `UML` class diagram of the framework architecture is presented in Figure 4.1. The `ThresholdSigner` interface exposes the following methods:

- `Sign`: Given a message, it returns a signature share signed with the secret key share.
- `Verify`: Given a message and a signature, it succeeds if the signature is valid.

The interface also provides four methods for signature reconstruction.

- `AssembleSignatureOptimistic`, `AssembleSignaturePessimistic`:
  Given an array of signature shares grouped with the IDs of the signers that generated them, they return a valid signature. Each method follows a different algorithm. We explore the reconstruction algorithms in detail in the Section 4.2.
- `AssembleSignatureOptimisticAsync`, `AssembleSignaturePessimisticAsync`:
  These methods are the asynchronous version of the methods described above. They are called upon the arrival of each signature share and they also take as argument a communication channel. When a valid reconstructed signature is available they write it in the communication channel.

To reconstruct the signature the `ThresholdSigner` must gather signature shares. To that end a dedicated data structure is implemented. We face two challenges. First, the `ThresholdSigner` must be able to store signature shares for different message. We use a hash map data structure where the key is the hash (SHA 256) of the message and the value is an array of signature shares for that message. Second, the signature shares may arrive asynchronously. However, the reads and writes in the data structure must be performed in a thread safe way. To achieve this, we use a mutex and lock before reading from or writing to the data structure. The data structure is named `SafeMap`.
The data structure also provides an efficient way to get combinations of the signature shares. As we see in the next section, a way to reconstruct a signature is to try all the possible combinations of k signature shares until we find a valid combination. Each time a new signature share arrives the combinations increase exponentially and nodes can quickly run out of memory. To avoid that and allow better scalability we save the combinations that we have already tried in the previous round. We save just the indeces of the signers that produced the signature share instead of the combinations of the signature shares themselves

[1] `https://golang.org/pkg/crypto/rsa/`
[2] `https://golang.org/pkg/math/big/`
[3] `https://github.com/manudrijvers/amcl/`

because this also requires a lot of memory. We save them as a string in a hash map so that checking for their existence is efficient.
Finally, the framework includes a key generation command line tool that generates the private and public keys for the threshold RSA and BLS signature scheme and outputs them in `yaml` configuration files, that can be used for the `ThresholdSigner` initialization.
The implementation is in `Go 1.7`[4], as the core of Hyperledger Fabric project. The software includes unit tests, benchmarks, and example configuration files.

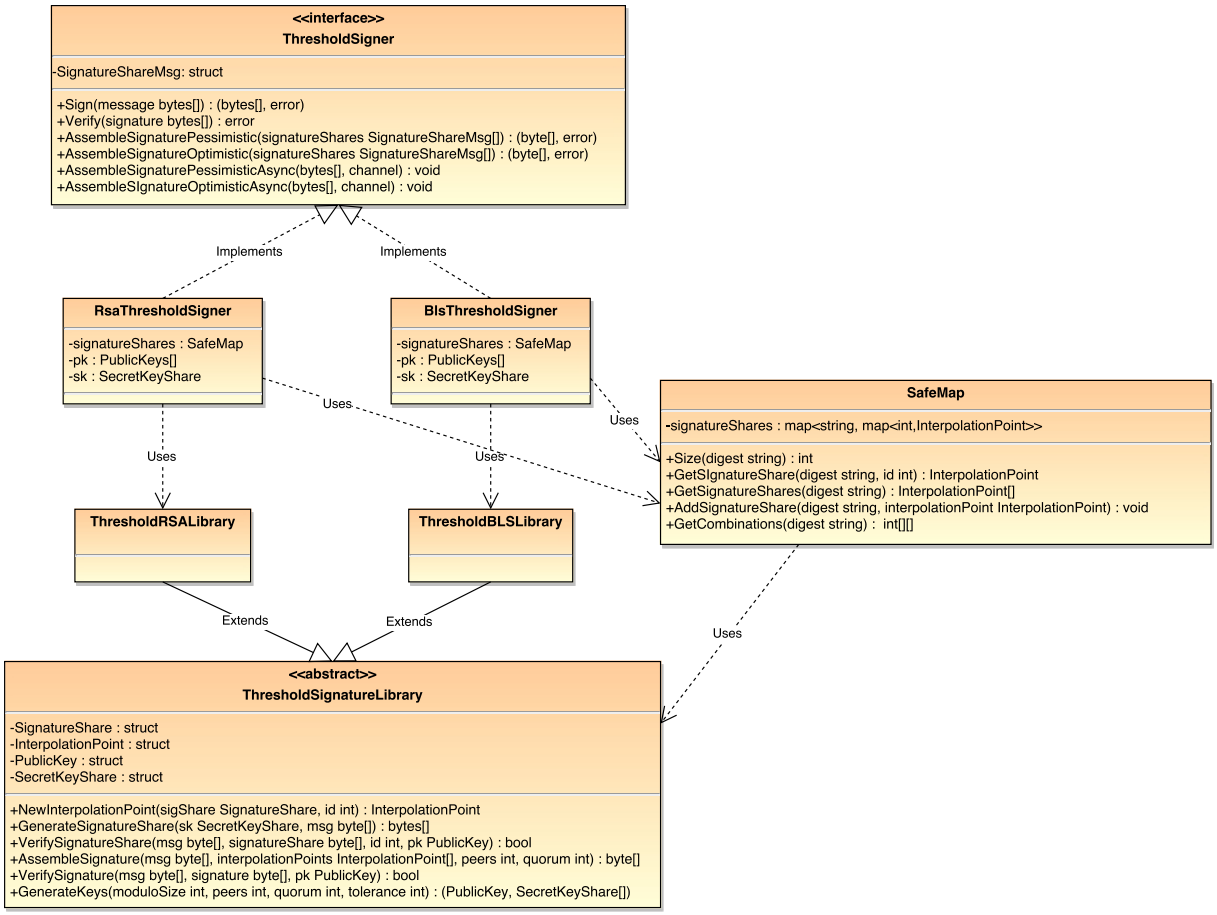

Figure 4.1. Class diagram of the Threshold Signatures framework.

4.2 Signature Share Combination Algorithms

To provide robustness a threshold signature scheme must tolerate corrupted signature shares. Therefore, part of the threshold signature schemes is a signature share verification algorithm.
Let us assume a simple scenario with a client C that wants a threshold signature on a message m and n signing servers $S_1, S_2, \cdots, S_n$ that form a (n, k, t) dual-threshold cryptosystem with n, k, t satisfying 2.1. Also, the client is equipped with an algorithm $\mathcal{A}$ that combines signature shares into a signature. C broadcasts a signature request to the servers and waits for them to reply with their signature shares.
At this point we have multiple design alternatives. The simplest and more straight forward case is the following. We assume that we have a reliable and synchronized network that will deliver all the signature shares to the client, however up to t servers are allowed to be corrupted and deliver an invalid signature share.
The client waits for n responses and calls the algorithm $\mathcal{A}$ that takes as an argument an array of the signature shares and a `ThresholdSigner` instance initialized with the public and verification keys

[4] `https://golang.org/doc/go1.7`

and the parameters of the cryptosystem. The algorithm validates the signature shares and if it finds k of them to be correct it reconstructs the signature. We call this algorithm `SynchronousPessimistic` because it assumes in a pessimistic way that some of the signature shares will be corrupted 1.

Algorithm 1 Synchronous Pessimistic Signature Share Combination

```
1: function AssembleSignaturePessimistic(signatureShares)
2:     validShares ← [ ]
3:     for s ∈ signatureShares do
4:         if VerifySignatureShare(s) then
5:             validShares.add(s)
6:     if lenght(validShares) > k then
7:         return AssembleSignature(validShares)
8:     return Error
```

However, validating all the signature shares is computationally expensive. As Cachin and Samar [5] suggest, the algorithm can be optimistic and assume that the first k signature shares are valid and use them to reconstruct the signature. If the signature shares are valid the reconstructed signature will be valid as well. If the signature is not valid the algorithm will try a different combination. We call this algorithm `SynchronousOptimistic` 2.

Algorithm 2 Synchronous Optimistic Signature Share Combination

```
1: function AssembleSignatureOptimistic(signatureShares)
2:     oldCombinations ← [ ]
3:     for i ∈ 0, ··· , n − k do
4:         combinations ← k-combinations(signatureShares[1..k + i])
5:         for c ∈ combinations do
6:             if c ∉ oldCombinations then
7:                 s ← AssembleSignature(c)
8:                 if VerifySignature(s) then
9:                     return s
10:                oldCombinations.add(c)
11:    return Error
```

In the average case, we expect this algorithm to be much faster that the pessimistic approach, especially if the signature scheme is the threshold RSA, where as we pointed out in Section 2.2.1, the signature share verification is very expensive. However, this algorithm scales badly if many shares are corrupted because the number of possible combinations increases exponentially.

Nevertheless, if the network does not guarantee the delivery of the signature shares, or, if some of the servers crash, the client will get stuck waiting for the signature shares. Instead, we can allow the servers to fail in an arbitrary way if the algorithm $\mathcal{A}$ is asynchronous. Instead of giving all the signature shares in an array as an argument, we call the algorithm every time a signature share arrives. A `Golang` communication channel is also given as an argument to the algorithm and the algorithm will write the signature to the channel when enough valid signature shares are available, for the pessimistic version 3, or when a valid signature share combination is found, for the optimistic version 4.

Finally, as an optimization for the pessimistic case, we can parallelize the signature share validation by spawning a thread for each signature share verification.

Algorithm 3 Asynchronous Pessimistic Signature Share Combination

```
1: function AssembleSignaturePessimisticAsync(sigShare, channel)
2:     validShares ← [ ]
3:     invalidShares ← 0
4:     if VerifySignatureSahre(sigShare) then
5:         validShares.add(sigShare)
6:         if lenght(validShares) = k then
7:             channel ← AssembleSignature(validShares)
8:     else
9:         invalidShares ← invalidShares + 1
10:    if invalidShares = t then
11:        channel ← Error
```

Algorithm 4 Asynchronous Optimistic Signature Share Combination

```
1: function AssembleSignatureOptimisticAsync(sigShare, channel)
2:     signatutreShares ← [ ]
3:     oldCombinations ← [ ]
4:     signatureShares.add(sigShares)
5:     if lenght(signatureShares) ≥ k then
6:         combinations ← k-combinations(signatureShares)
7:         for c ∈ combinations do
8:             if c ∉ oldCombinations then
9:                 s ← AssembleSignature(c)
10:                if ValidSignature(signature) then
11:                    channel ← s
12:                oldCombinations.add(c)
13:    if lenght(signatureShares) = n then
14:        channel ← Error
```

4.3 Threshold Signatures for Transaction Endorsements

First, let us explore in detail the transaction flow as it is currently implemented.
The blockchain network is comprised of nodes of a specific role. In the current version (`v1.0`[5]) of HLF we distinguish two different types of nodes.

- The nodes that provide the consensus service are called *consenters*. They run a consensus protocol to guarantee the ordering of the transactions.
- A *peer* is a node that maintains the ledger. A peer can also have one of the following special roles:
  - The *submitting peers* provide a client interface for submitting transactions.
  - The *endorsing peers* have the role of simulating a transaction to ensure that the outcome is deterministic and stable before it is submitted ordering service. Each chaincode is associated with an endorsement policy, defining a set of endorsers required to validate a transaction. This action is called endorsement.

In Figure 4.2 we can see an example of a blockchain network for the current version of HLF. Endorsers may belong to a certain organization.

[5] `https://github.com/hyperledger/fabric/blob/master/proposals/r1/Next-Consensus-Architecture-Proposal.md`

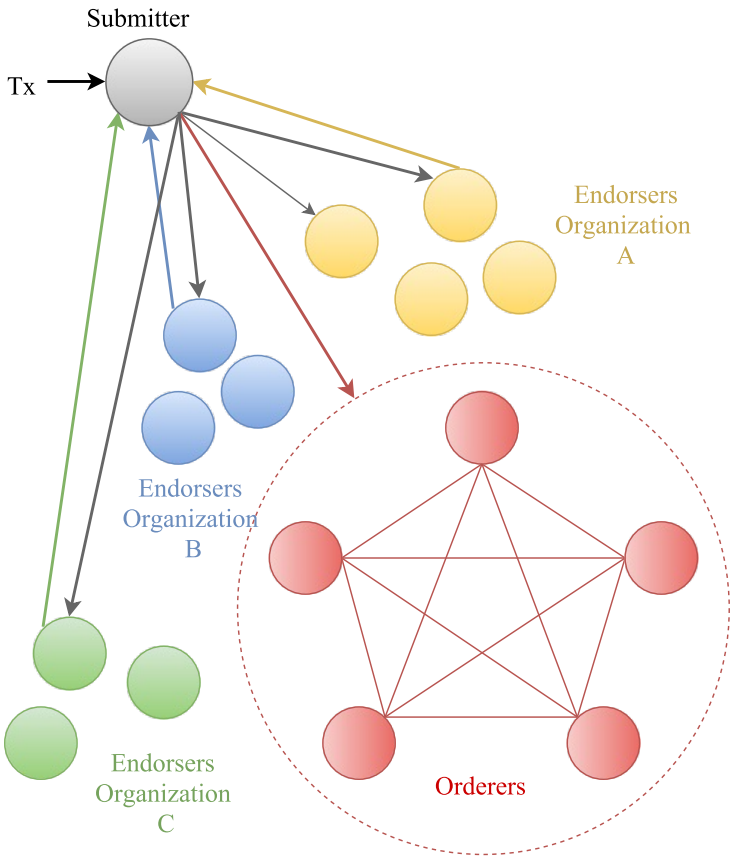

Figure 4.2. Blockchain network and transaction flow example for HLF `v1.0`.

When a client submits a transaction to a submitting peer, the submitting peer, after verifying the clientFLs signature, wraps the transaction in a `proposal` message, Figure 4.3, and sends it to a set of endorsing peers, or endorsers.

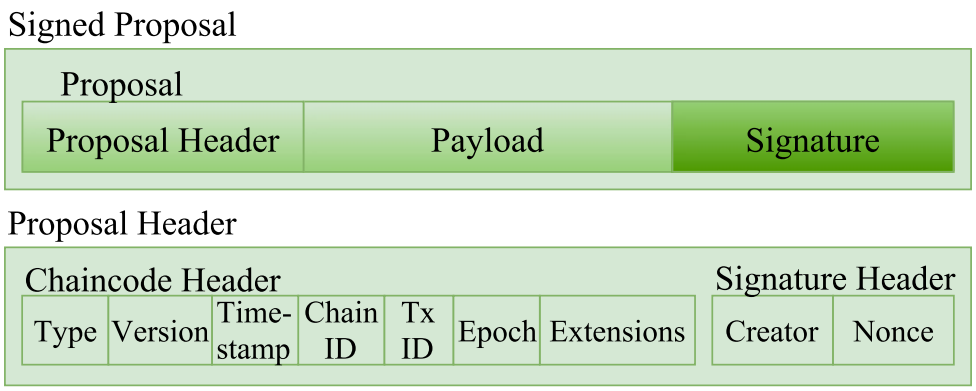

Figure 4.3. Proposal message.

The endorsers are selected according to an endorsement policy associated with the transaction chaincode. Such a policy is a logical expression that dictates which peers are suitable for endorsing transactions of the specific chaincode and how many of them must endorse the transaction. The endorsers, upon receiving the proposal, verify the signature of the client and simulate the transaction. Then, they send a `proposal response` message, Figure 4.4, to the submitting peer, signed with their private key, containing the result of the simulation, i.e. if the transaction is valid.

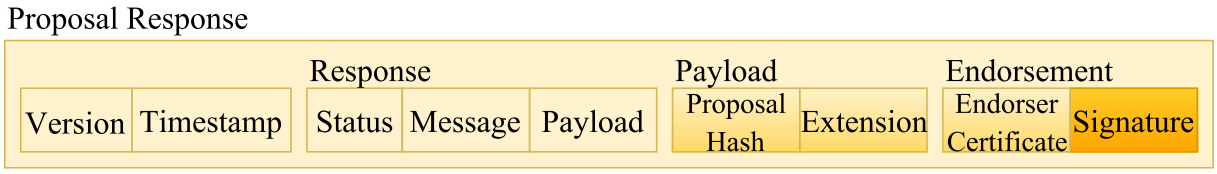

Figure 4.4. Proposal Response message.

The submitting peer waits until it receives enough valid endorsements and then wraps them in a `chaincode action payload` message. It adds this message in a `transaction envelope` message, Figure 4.5, signs it and submits it to the ordering service. Afterwards, every peer, upon receiving the transaction from the ordering service, they validate each one of the endorsements and, provided that the signatures of the endorsers are valid and that the endorsement policy for the chaincode is satisfied, they write the transaction in their ledger.

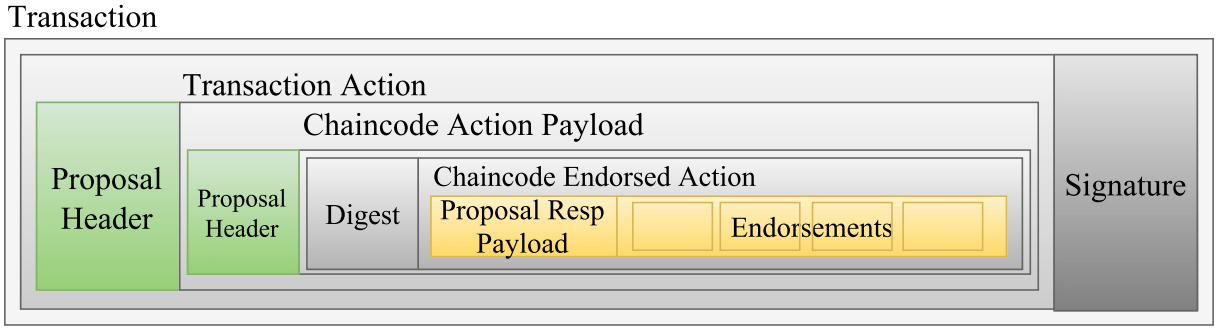

Figure 4.5. Transaction Envelope message.

The transaction endorsement and validation is summarized in Figure 4.6.

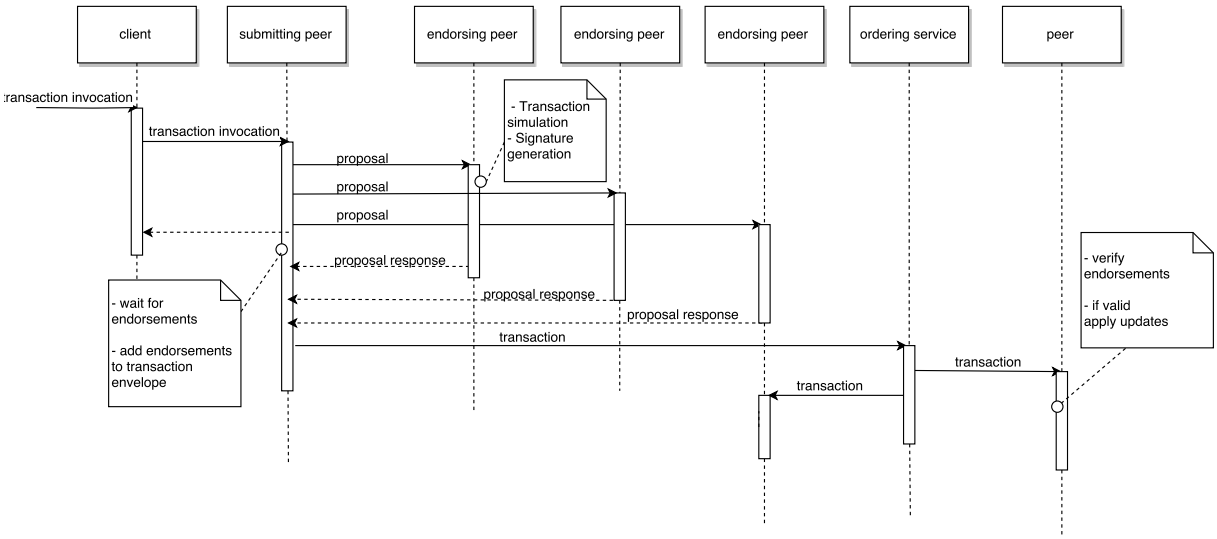

Figure 4.6. Transaction flow in HLF `v1.0`.

In the transaction flow described above, the validator needs to validate the signatures of all the endorsers. Moreover, the transaction message needs to contain all the `endorsement` messages. This approach is inefficient. To that end, as mentioned in Section 3.3, we modify the transaction endorsement procedure so that the endorsers participate in a threshold signature scheme. In this way, the submitting peer after collecting the required number of endorsements, signed by the endorsers with their secret key share, reconstructs the signature and adds only one `endorsement` message to the transaction that contains

the reconstructed signature, as we can see in Figure 4.7. Also, the validator needs now to verify only one signature, thus accelerating the transaction validation.

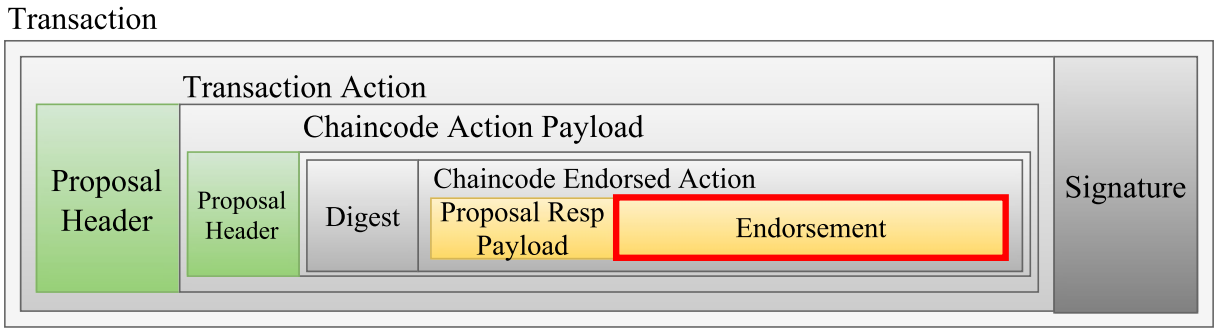

Figure 4.7. The transaction envelope with a single endorsement message containing the reconstracted signature.

To integrate the threshold signatures in the endorsement procedure a new membership service provider (MSP) needed to be implemented. The MSP provides a signing identity object to the peer. The new membership service implements the `ThesholdSigner` interface. Therefore, when the endorser calls the sign method of the signing identity instead of creating a signature with a private key, it creates a signature share with a private key share.
Minimal changes needed to be done in HLF core. First, some modifications needed to be done in the submitting peer. The submitting peer is responsible for collecting the endorsements. Therefore, in the case of a threshold signature scheme, the submitting peer uses one of the algorithms described in Section 4.2 to combine the signature shares of the collected endorsements to a signature. Also, for the Lagrange interpolation in the signature reconstruction, the signature share must be associated with the ID of the signer. To achieve this an extra field was added to the `endorsement` message, as we can see in Figure 4.8. The endorser certificate, already existing in the message, could not be used for identification because it is the same for all the endorsers, since there is only one public key for the threshold signature scheme.

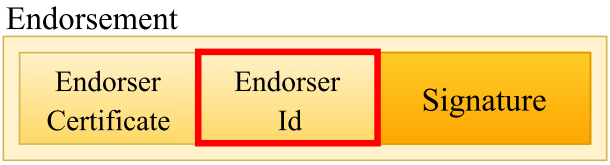

Figure 4.8. The modified endorsement message with the new `ID` field.

Finally, for the validator, the signature validation is the same as if the signature was from a non-threshold scheme. The modified transaction endorsement and validation flow is summarized in Figure 4.9.

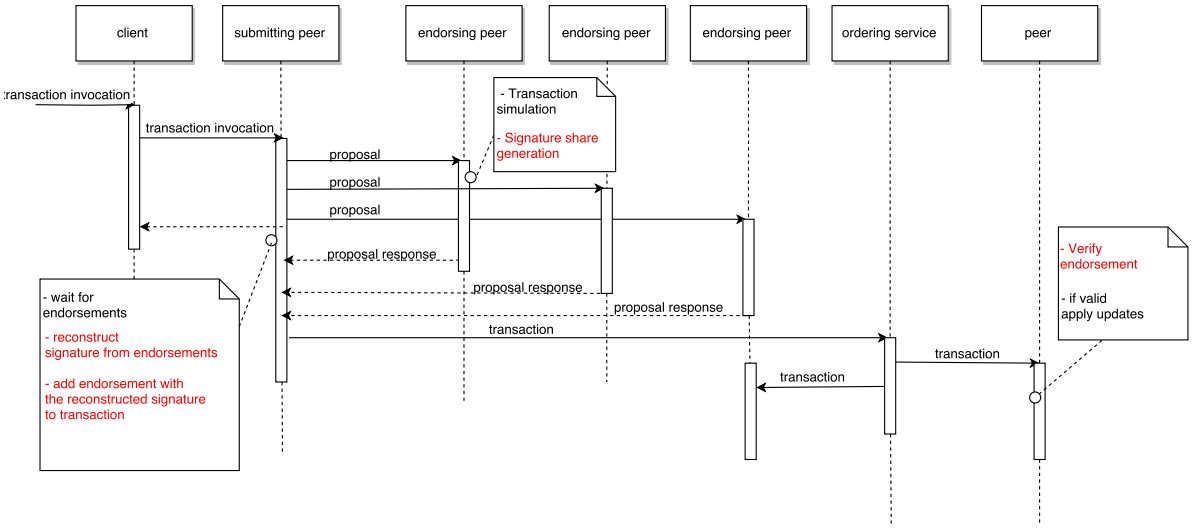

Figure 4.9. Transaction flow with threshold signatures.

Chapter 5

Evaluation

In this chapter, we evaluate the performance of the threshold signature schemes we implemented. We perform benchmarks to examine how they scale while increasing the number of participating nodes. We compare threshold RSA to threshold BLS and we also compare them to multisignature schemes.
First, we examine the performance of the signature reconstruction operation. We are comparing the performance of the two different approaches we introduced in Chapter 4, the *pessimistic* and the *optimistic* approach. We assume a network of n signing participants f out of which might be faulty. For our benchmarks, we set $n = 3f + 1$ and we request $k = f + 1$ valid signature shares for the signature reconstruction. We evaluate two scenarios. In the first scenario all the signature shares we receive are valid. In the second, we have a maximum number of f corrupted signature shares that are distributed within the n signature shares in a uniformly random way. For the benchmarks, we use the synchronous methods because we want to measure the time required per signature reconstruction operation and we do not want to introduce further delay assuming asynchronous network delivery. In the Figures 5.1, 5.2, 5.3 we can see the performance of the threshold RSA signature scheme for increasing values of the RSA modulus N=1024, 2048, 3072. On the horizontal axis we have the number of participants $n = 3f + 1$ and on the vertical axis we can see the required time per operation in milliseconds. Today, for a secure RSA scheme, keys of length 2048 bits are reasonable, while in the future we expect that 3072 bits will be required[1].

[1] `https://www.keylength.com/en/3/`

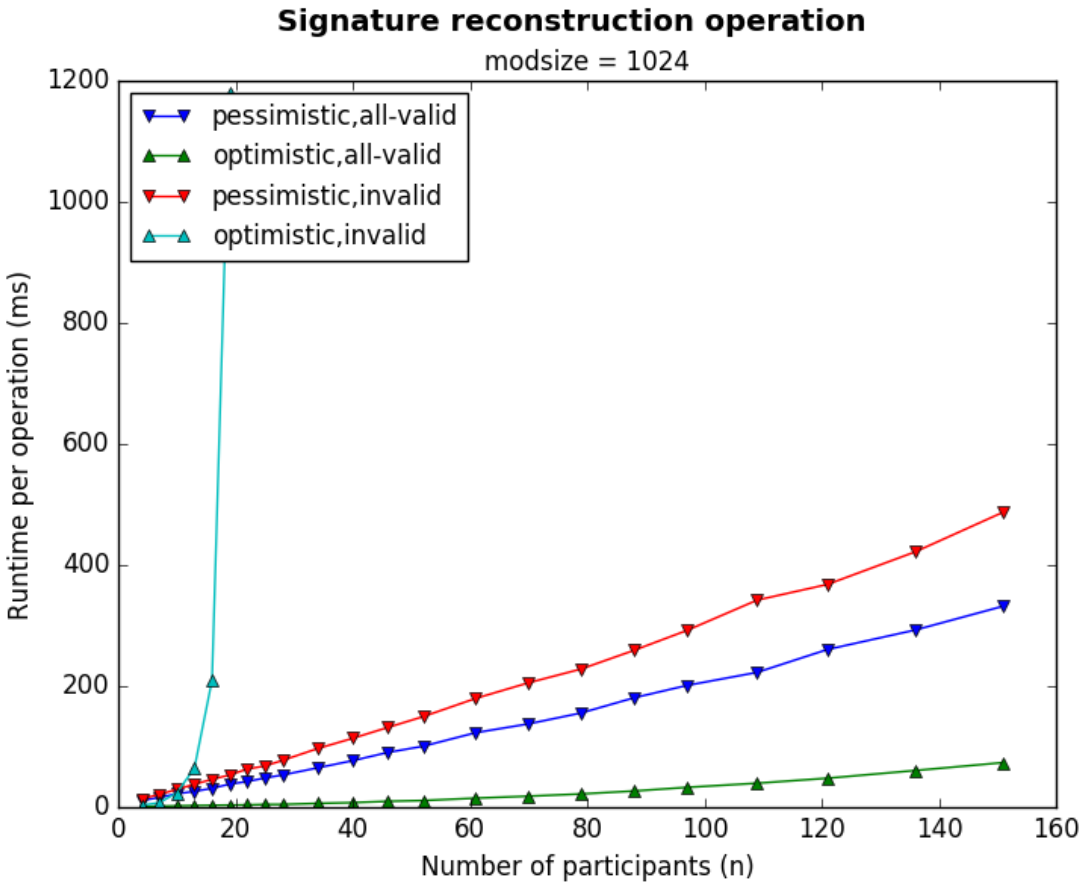

Figure 5.1. Threshold RSA signature reconstruction benchmark for modulus size $N = 1024$

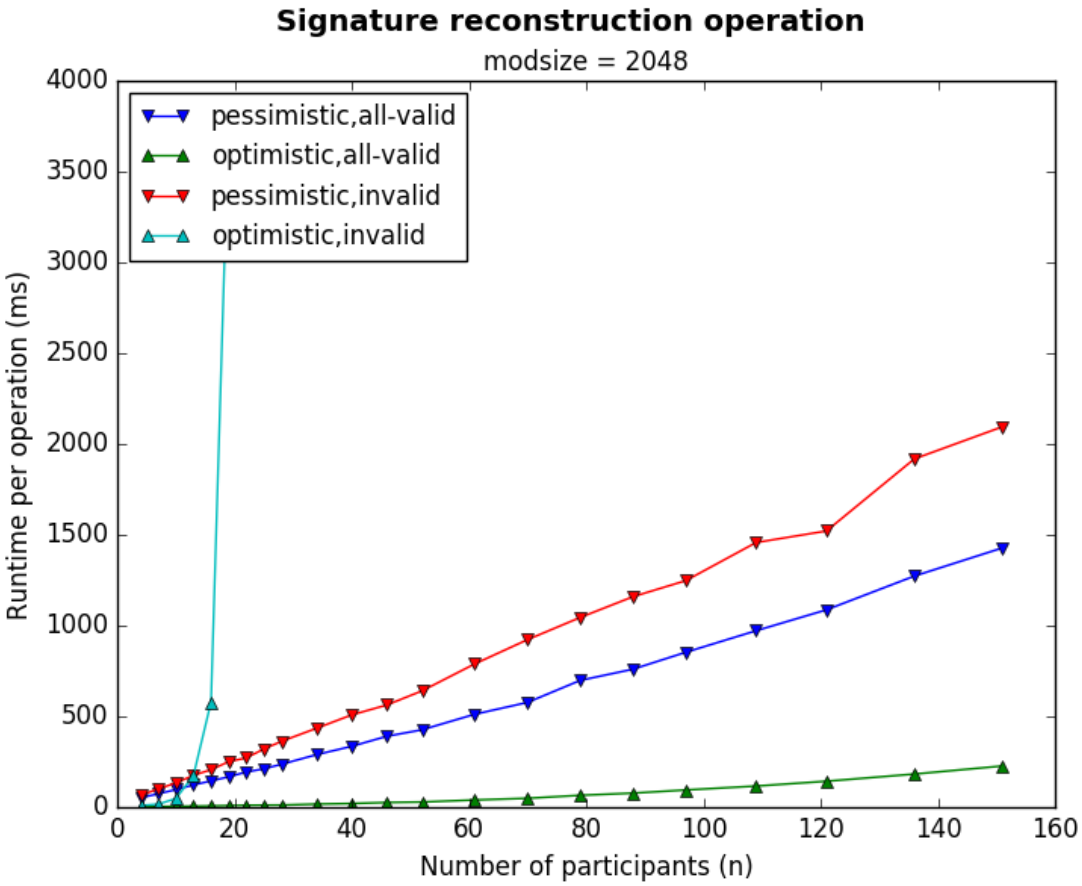

Figure 5.2. Threshold RSA signature reconstruction benchmark for modulus size $N = 2048$

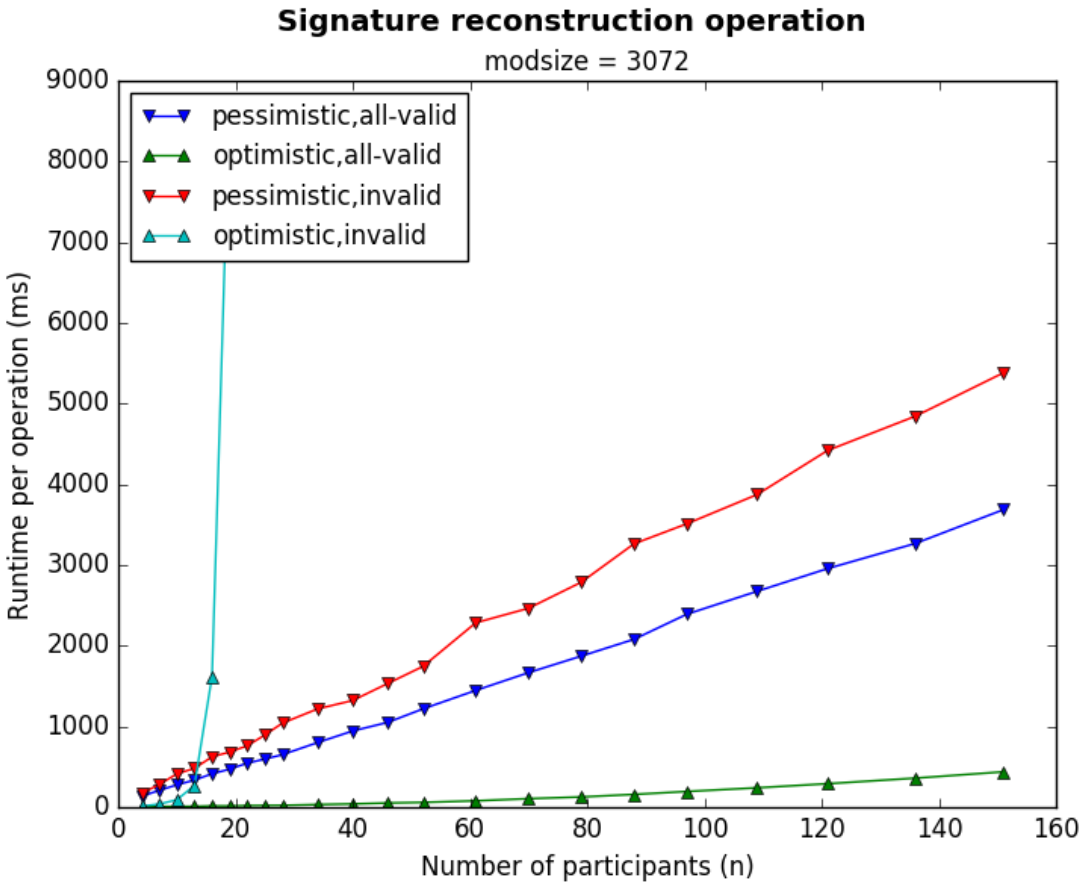

Figure 5.3. Threshold RSA signature reconstruction benchmark for modulus size $N = 3072$

As we can see in Figures 5.1, 5.2, 5.3 the optimistic approach is more efficient when we do not have corrupted signature shares. To understand this, we need to have a look in the core of the reconstruction operation for each approach. In the optimistic approach, we have a polynomial interpolation followed by a signature verification. For the pessimistic approach, we have k signature share verifications followed by a polynomial interpolation. An important factor here is that, as we see in Table 5.2, the signature share verification is more expensive than the signature verification. Also, in the pessimistic approach as the number of participants increases, the signature share verification operations that must be performed increase as well. However, when we have corrupted signature shares, in the optimistic approach, the time per signature reconstruction operation increases exponentially and, after a threshold, is slower than the pessimistic approach. This happens because for the optimistic approach if the reconstructed signature from the first k signature shares is found invalid we search for a valid signature of a k-combination out of $k+i$ signature shares where $i = 1, \cdots, n-k$. The number of k-out-of-n combinations is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ which increases exponentially.

Moreover, in the Figures 5.1, 5.2, 5.3 we can observe that the time per signature reconstruction operation increases as the RSA modulus N size increases. This is also illustrated in Figure 5.4. This is expected, since for both signature verification and signature share verification the most expensive operation is an exponentiation to N and $2N + L1$ respectively, where $L1$ a secondary security parameter respectively, which depends on the modulus size N. Also, the signature share verification is more expensive than the signature verification because $N < 2N + L1$. Finally, regarding the scenario with corrupted signature shares, we can observe that for an increasing value of N the number of nodes for which the optimistic approach is better than the pessimistic also increases. In Table 5.1 we can see the number of participants up to which the optimistic approach is faster even for the scenario with the corrupted signature shares.

Modulus size	Number of participants
1024 bits	10
2048 bits	12
3072 bits	14

Table 5.1. Number of participants up to which the optimistic approach is faster than the pessimistic approach for threshold RSA even with corrupted signature shares.

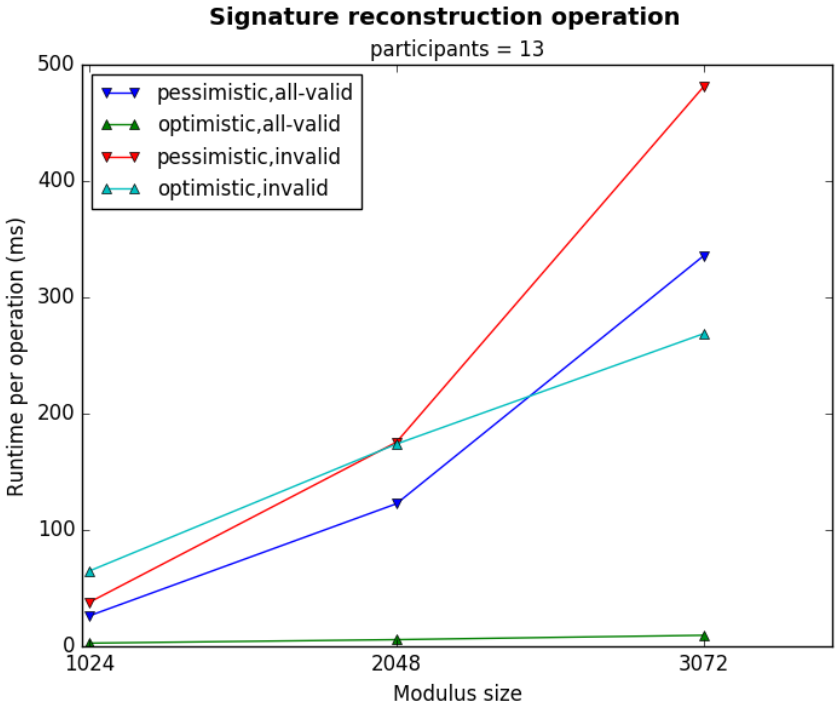

Figure 5.4. Threshold RSA signature reconstruction benchmark for increasing modulus size.

We now evaluate the performance of the threshold BLS signature scheme and compare it to threshold RSA. For the threshold BLS scheme of modulus size 254 bits the corresponding RSA modulus size with equivalent security is $N = 3072$ bits[2]. In Figure 5.5 we observe that, as for threshold RSA, also for threshold BLS, when we do not have corrupted parties the optimistic approach is more efficient. For the pessimistic approach k signature share verifications are required, while for the optimistic approach only one. In the scenario where we have corrupted signature shares again the optimistic approach scales exponentially. In fact, because for threshold BLS the signature verification and the signature share verification are the same, as we see in Table 5.2, the optimistic approach with corrupted signature shares is always worse.

[2] `https://www.keylength.com/en/3/`

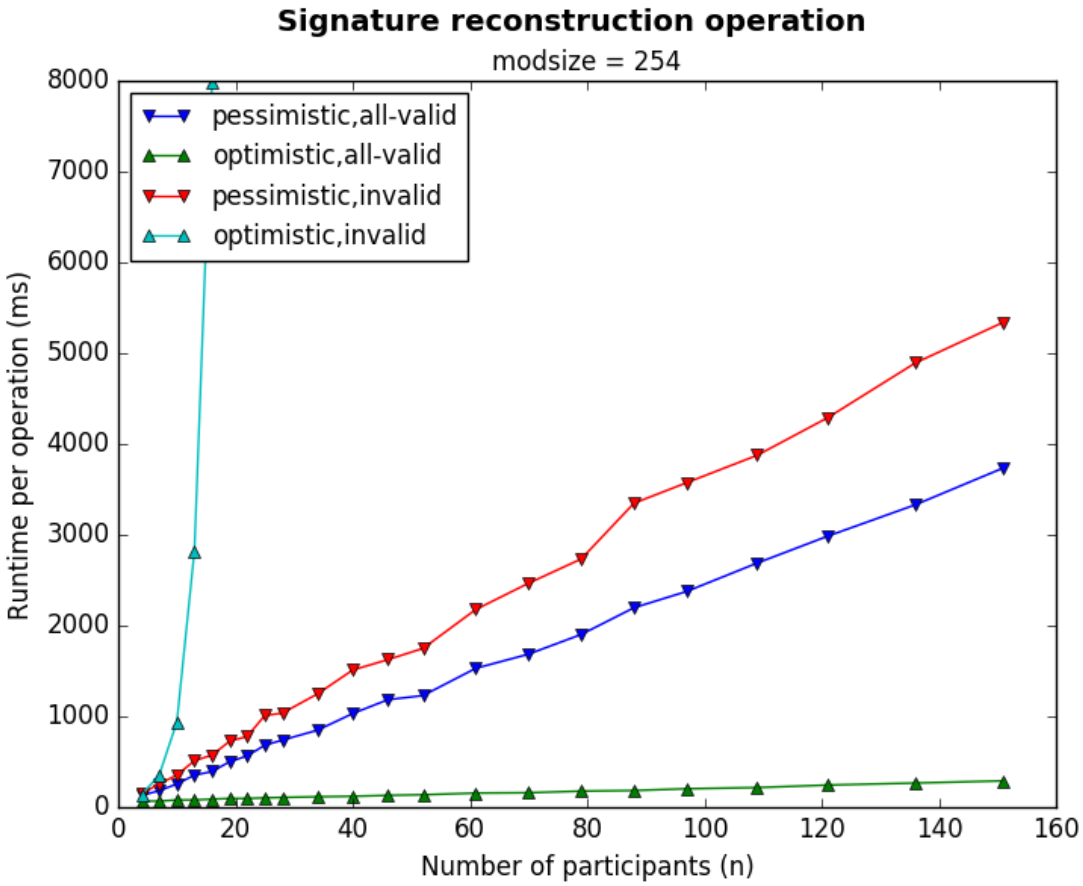

Figure 5.5. Threshold BLS signature reconstruction benchmark.

In Figures Figures 5.7, 5.8 we can see the threshold BLS and the corresponding threshold RSA scheme together, for the two scenarios, with all valid signature shares and f invalid signature shares respectively. We observe that in the pessimistic approach the two algorithms have similar performance, since the cost for the signature share verification is similar. For the optimistic approach, when we do not have invalid signature shares, we can see that threshold RSA performs faster for approximately up to 100 nodes. To understand this we must take two factors into consideration. As we see in Table 5.2 the signature verification is much faster for threshold RSA which results in a faster signature reconstruction for up to 100 nodes. However, the internal reconstruction of the signature from the signature shares with polynomial interpolation, as we see in Figure 5.6 is faster for threshold BLS. This happens because in the case of threshold RSA, as we saw in Section 2.2.1 we have some additional operations which for large numbers are not negligible. Therefore, as the number of participants increases it affects the cost of the signature reconstruction .

Threshold RSA		Threshold BLS	ECDSA
Signature Verification	Signature Share Verification	Signature & Signature Share Verification	Signature
0.022 ms/op	69.878 ms/op	53.807 ms/op	0.123 ms/op

Table 5.2. Signature and signature share verification comparison.

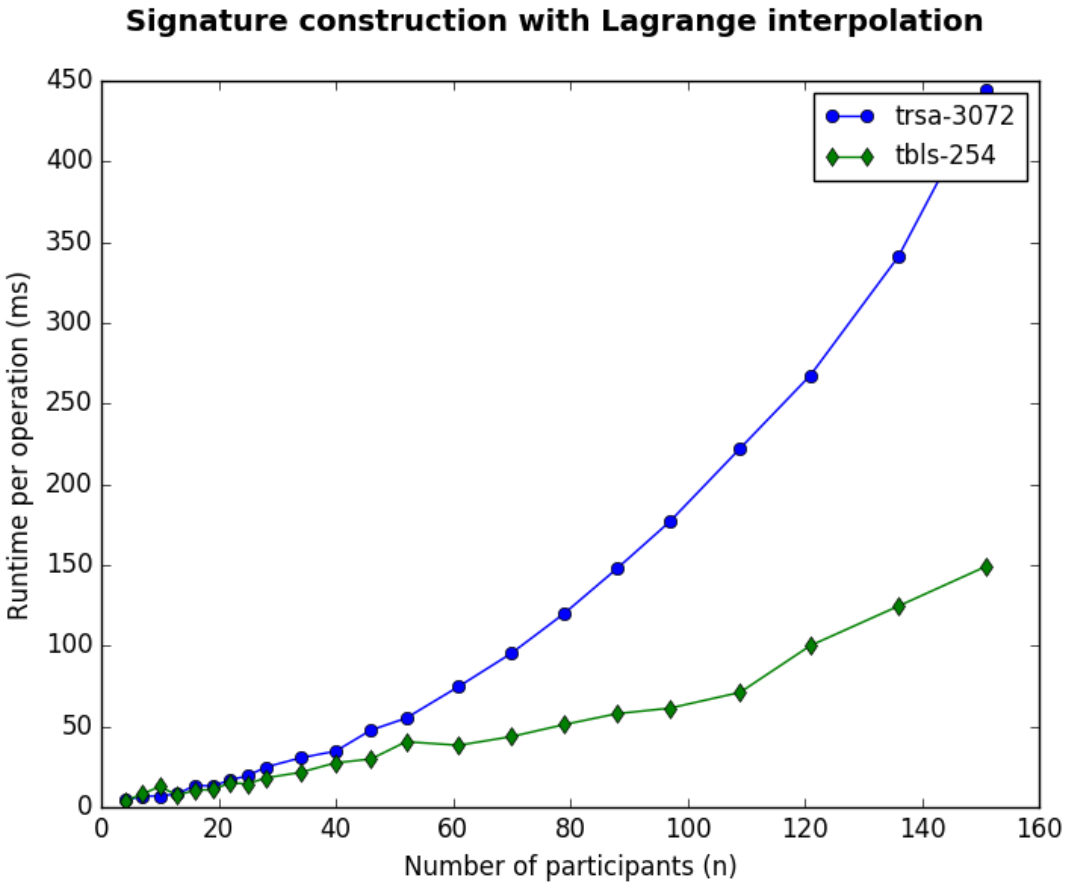

Figure 5.6. Internal signature reconstruction from signature shares with polynomial interpolation.

From our evaluation of the signature reconstruction operation, we conclude that the choice between the optimistic and the pessimistic approach is relevant to the application. When we expect that the corrupted signature shares are very sporadic, or that all the nodes of the network can be trusted, the optimistic approach is suggested. For threshold RSA the optimistic approach is also suggested for an application that requires only a small set of participating nodes. Such applications are realistic. An example is the two-factor authentication in [10] or an endorsement policy for HLF that requires a small number of endorsements. Also, HLF is a permissioned blockchain and therefore we do not expect corrupted parties to be the norm. For a big network where we would expect a lot of corrupted nodes, such as the Bitcoin network, the pessimistic approach is suggested. However, a combination of the two algorithms is also possible. For example, the client that wants to reconstruct the signature can run both algorithms in parallel and get the signature from the algorithm that finishes first.

We now compare the two threshold signature schemes to multisignatures. Multisignatures is a different distributed signature scheme. As mentioned in Section 2.3 multisignatures are already implemented as an option for the Bitcoin protocol and as we saw in Section 4.3 are also used in HLF for the endorsement policies and in the core of the PBFT protocol. We implemented two multisignature schemes, one with RSA and one with ECDSA signatures. For the multisignature schemes each participant has a public-private key pair. The equivalent of a signature share is a signature by the private key of the participant and the equivalent of the signature share verification is the verification of the signature by the public key of the signer. The equivalent of the pessimistic signature reconstruction of a threshold signature scheme is to verify each signature until k valid signatures are collected and output an array of all the k valid signatures. An equivalent for the optimistic approach does not exist; if we construct a set of the first k signatures we cannot know if the set is valid without validating each signature. Finally, the signature verification of the reconstructed signature equivalent is the verification of the set of the k signatures.

We first compare the signature reconstruction operation for an increasing number of participants. As

before $n = 3f + 1$, $k = f + 1$. We compare a threshold RSA (trsa) and a multisignature RSA scheme (mrsa) both of public exponent $E = 2^{16} + 1$ and modulus size $N = 3072$ bits, the threshold BLS signature scheme (tbls) with modulus size 254 bits and a multisignature ECDSA (mecdsa) scheme with modulus size 256 bits. As we can see in Figures 5.7, 5.8 the multisignature reconstruction is much more efficient. This is explained if we take into consideration that the verification of a single signature for RSA and ECDSA , which we call signature share verification for the multisignature scheme, is much faster than the threshold RSA and threshold BLS signature share verification. Also in the case of multisignatures we don't have the additional cost of the polynomial interpolation.

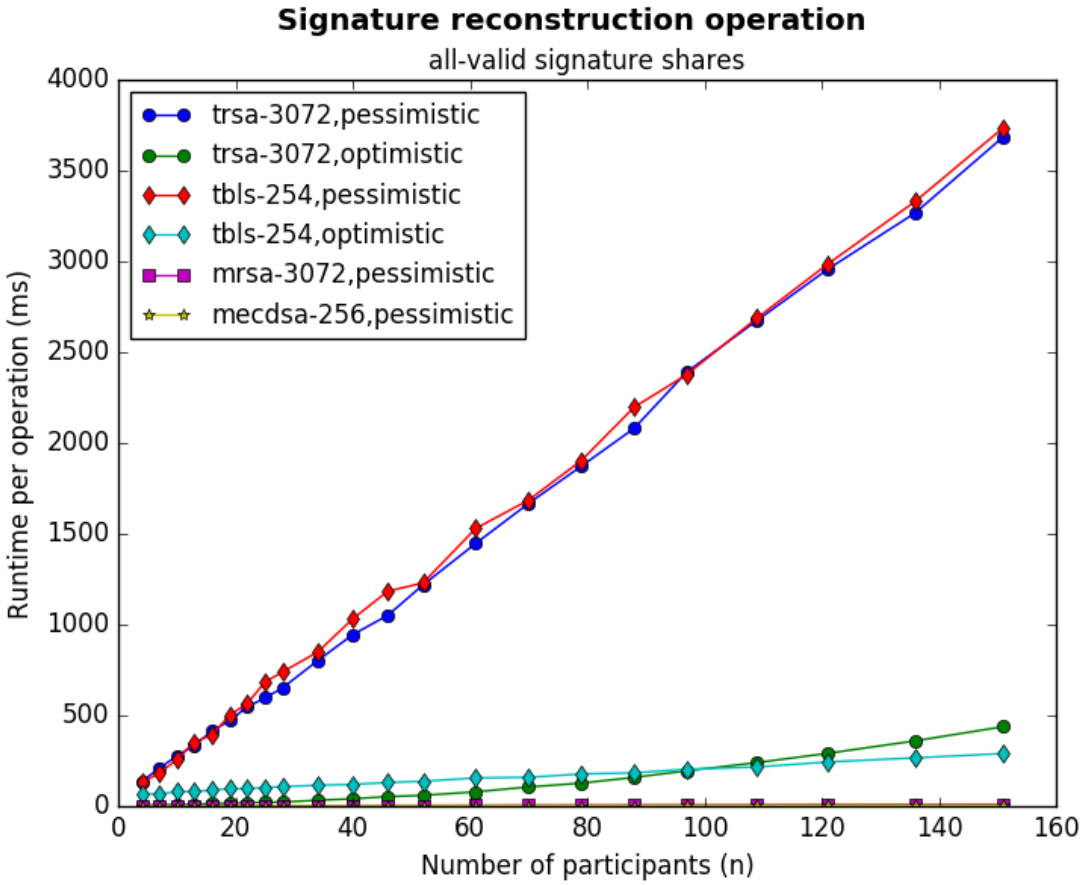

Figure 5.7. Comparison of signature reconstruction operation for threshold signatures and multisignatures with all the signature shares valid. For the multisignature schemes the operation costs less than 20 ms.

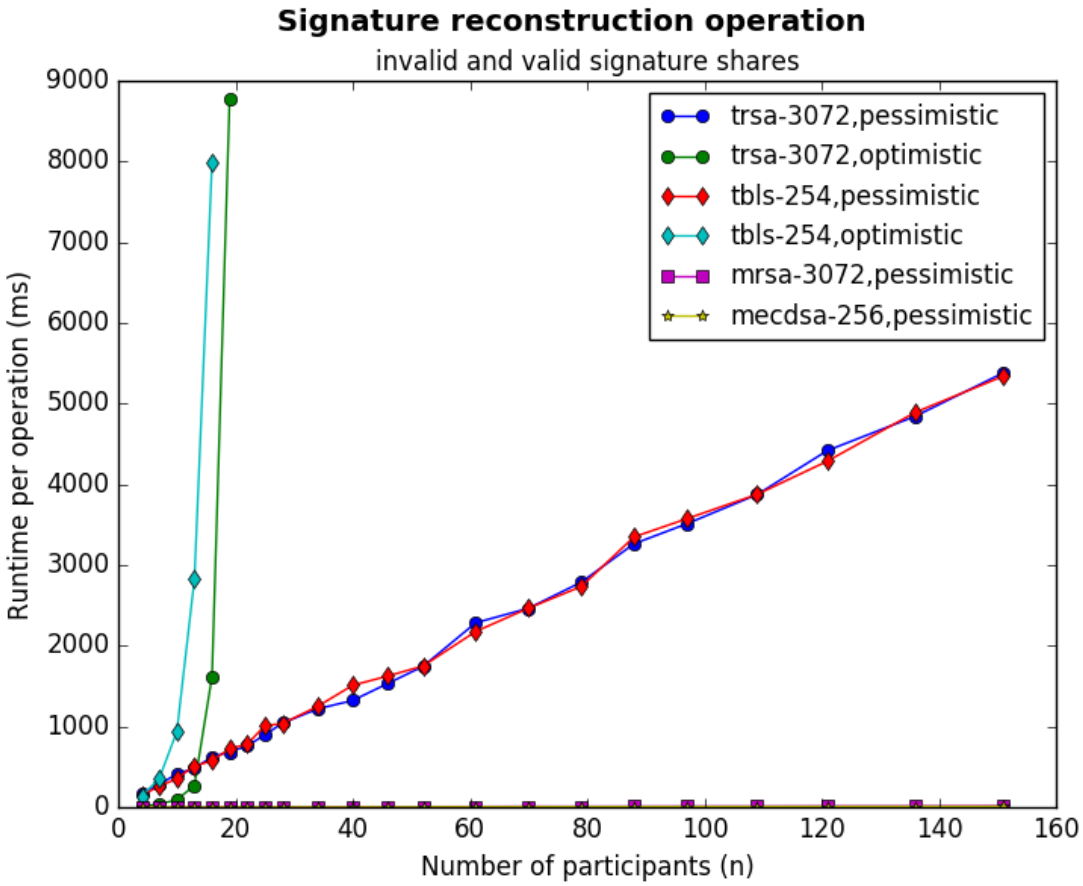

Figure 5.8. Comparison of signature reconstruction operation for threshold signatures and multisignatures with f random signature shares invalid. For the multisignature schemes the operation costs less than 20 ms.

We also compare the verification of the reconstructed signature. As we can see in Figure 5.9 for the threshold schemes the required time per operation is always the same since we always verify only one signature. For the multisignature schemes the verification time increases linearly since we have to perform $i \in k, \cdots, n$ signature verification operations.

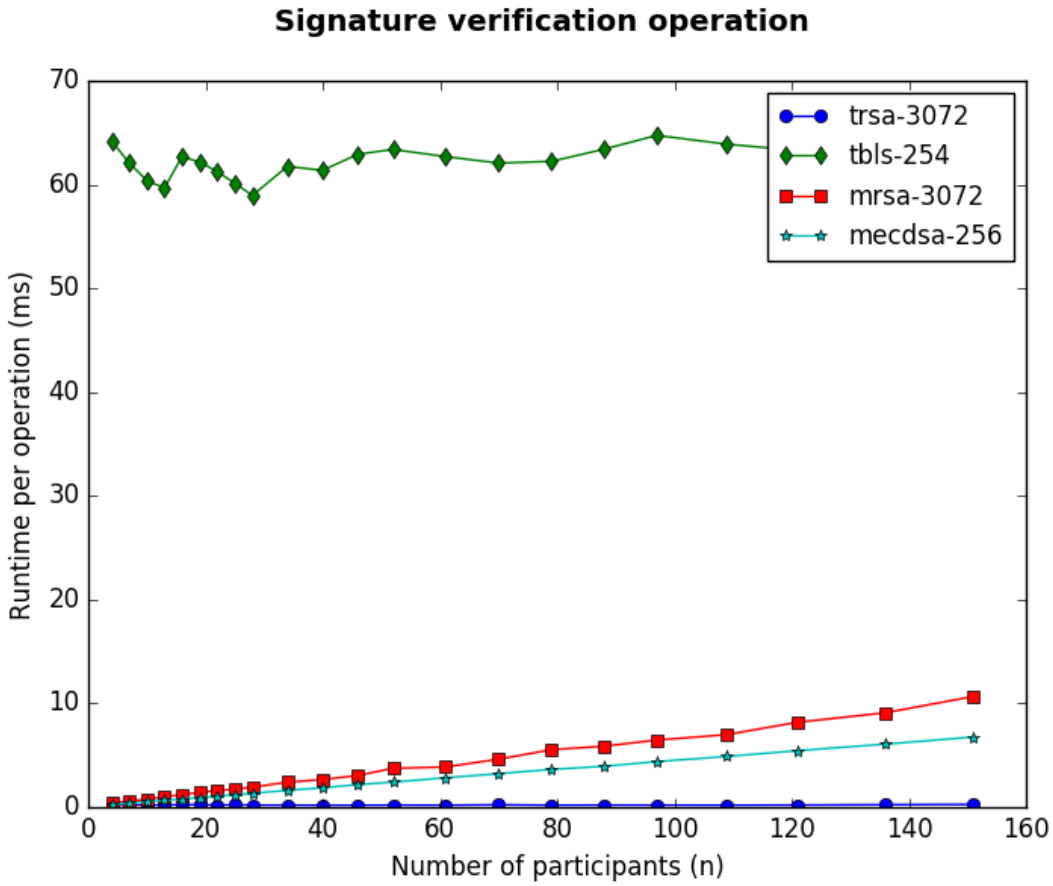

Figure 5.9. Comparison of signature verification operation for threshold signatures and multisignatures.

Finally, we must also consider the size of the reconstructed signature, since the size of the signature affects the network propagation time. In Table 5.3 we can see the size of the assembled signature form k signature shares. For the threshold signature schemes the size of the signature is fixed whereas for the multisignatures the assembled signature is an array of k signatures. In Figure 5.10 we can see how the size of the signature is affected by the number of nodes in the network. Again we assume $n = 3f + 1$ and $k = f + 1$.

	Signature size
Threshold RSA	384 bytes
Threshold BLS	64 bytes
ECDSA multisignatures	$64 * k$ bytes
RSA multisignatures	$384 * k$ bytes

Table 5.3. Reconstructed signature size in bytes.

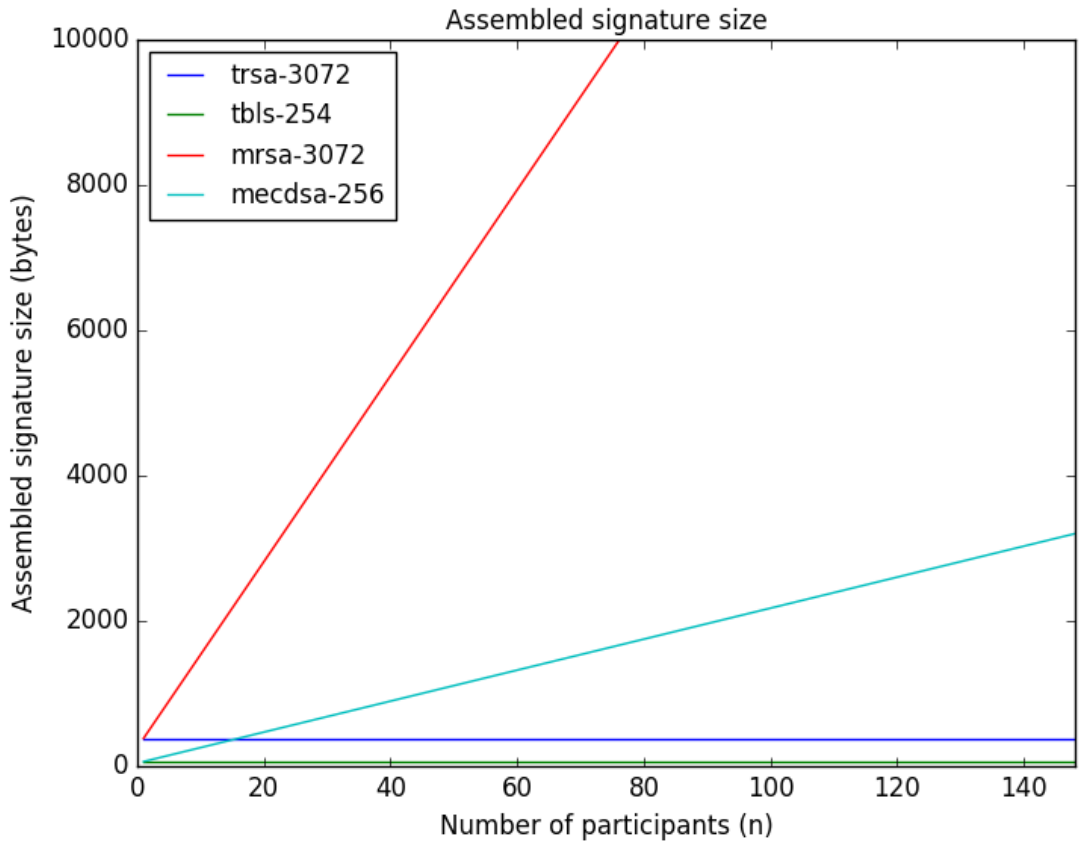

Figure 5.10. Comparison of signature verification operation for threshold signatures and multisignatures.

From the evaluation we can conclude that the signature reconstruction for the threshold signature schemes is more expensive than in the case of multisignature schemes. However, the signature verification of the threshold signature schemes requires a fixed amount of time and thus scales better for a large number of participants. Moreover, the signature size is fixed for the threshold signature schemes and therefore it does not increase the network traffic nor the delay as the number of participants increases.
The evaluation was performed on a Linux virtual machine, run by `Virtualbox` version `5.1.6 r110634 (Qt5.5.1)` in a Windows 7 laptop. The technical characteristics are presented in Tables 5.4, 5.5

Operating System	Windows 7 Professional, 64-bit
Memory	16.0 GB (15.6 GB usable)
CPU	Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz 2.70GHz
Model	ThinkPad w530

Table 5.4. Technical characteristics of the host.

Operating System	Ubuntu 16.04.2 LTS, 64-bit
Memory	12.0 GB
Number of CPUs	4

Table 5.5. Technical characteristics of the virtual environment.

Chapter 6

Conclusion

In this work we explored the advantages of threshold signatures for permissioned blockchain systems and introduced threshold signatures for HLF. Two threshold signature schemes: (1) threshold RSA and (2) threshold BLS, suitable to the properties and needs of HLF, were implemented and evaluated. Furthermore, two different algorithms for the threshold signature reconstruction were implemented and evaluated. Additionally, a framework was designed and implemented to make available the threshold signature schemes to HLF. As an application, threshold signatures were integrated to the transaction endorsement procedure. Finally, further potential use cases of threshold signatures in HLF were suggested.

We chose to implement threshold RSA and BLS because they are non-interactive and deterministic. Also, the resulting signature is equivalent to the non-threshold version of the scheme, allowing thus a seamless integration into HLF. From the performance evaluation, we concluded that threshold RSA is more efficient. On the other hand, threshold BLS has shorter signatures and also is suitable for distributed key generation and proactive security.

Regarding the suggested algorithms for signature reconstruction, we concluded that the choice has to do with the characteristics of the network. For a small and/or reliable network the optimistic approach is more efficient. The optimistic approach does not scale well if we have a lot of corrupted signature shares. However, a system can combine the two methods.

Finally, we compared RSA and BLS threshold signatures to RSA and ECDSA multisignatures. The signature assembling for multisignatures scales better for an increasing number of nodes. However, for the threshold signature schemes a validator needs to validate only one signature whereas for the multisignatures the number of the signatures that must be validated increases linearly with the number of participants. Finally, a threshold signature has a fixed size whereas for multisignatures the size again increases linearly with the number of participants.

Our framework will allow any potential future application to use a threshold signature scheme. Moreover, it allowed a minimalistic integration with the transaction endorsements; only a few lines of code of the core HLF implementation needed to be changed. Also, future threshold signature implementations can be easily added as pluggins. The source code will be made public under the repository of HLF [1].

Concluding, threshold signatures are a powerful tool introduced more than two decades ago, when blockchain systems did not exist. Today we can leverage it to enhance the security and resilience of blockchain systems.

[1] `https://gerrit.hyperledger.org/r/#/admin/projects/fabric`

Bibliography

[1] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[2] C. Cachin. Distributing trust on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN 2001) (formerly: FTCS), 1-4 July 2001, Göteborg, Sweden, Proceedings*, pages 183–192. IEEE Computer Society, 2001.

[3] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proc. 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 88–97. ACM, 2002.

[4] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[5] C. Cachin and A. Samar. Secure distributed DNS. In *Proc. International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 423–432. IEEE Computer Society, 2004.

[6] M. Castro, B. Liskov, et al. Practical Byzantine fault tolerance. In *Proc. Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999.

[7] B. Charron-Bost, F. Pedone, and A. Schiper. *Replication: theory and practice*, volume 5959. Springer, 2010.

[8] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *Proc. Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1992.

[9] Y. Desmedt. Threshold cryptosystems. In *Proc. Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 1989.

[10] R. Gennaro, S. Goldfeder, and A. Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *Proc. Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016*, volume 9696 of *Lecture Notes in Computer Science*, pages 156–174. Springer, 2016.

[11] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[12] S. Goldfeder, J. Bonneau, E. W. Felten, J. A. Kroll, and A. Narayanan. Securing bitcoin wallets via threshold signatures. `http://www.cs.princeton.edu/~stevenag/bitcointhresholdsignatures.pdf`, 2014.

[13] A. Kate and I. Goldberg. Distributed key generation for the Internet. In *Proc. 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*, pages 119–128, 2009.

[14] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *Proc. 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 279–296. USENIX Association, 2016.

[15] L. Lamport. Paxos made simple. `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`, December 2001.

[16] A. J. Menezes. *Elliptic curve public key cryptosystems*, volume 234. Springer Science & Business Media, 2012.

[17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[18] M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):986–1009, 1994.

[19] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[20] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[21] V. Shoup. Practical threshold signatures. In *Proc. Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000.

[22] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *Proc. IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 526–545. IEEE Computer Society, 2016.

[23] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[24] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/paper.pdf`, 2014.

[25] L. Zhou, F. B. Schneider, and R. Van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)*, 20(4):329–368, 2002.

[26] X. Zhou, Q. Wu, B. Qin, X. Huang, and J. Liu. Distributed Bitcoin account management. In *Proc. Trustcom/BigDataSE/ISPA, 2016 IEEE*, pages 105–112. IEEE, 2016.