Jorge Filipe Ferreira Pereira

Master of Science

Decentralizing Trust with Resilient Group Signatures in Blockchains

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Henrique João Lopes Domingos, Associate Professor, NOVA University of Lisbon

Examination Committee

Chair:	Carla Maria Gonçalves Ferreira, Associate Professor, FCT-NOVA
Rapporteur:	Nuno Miguel Carvalho dos Santos, Associate Professor, IST
Member:	Henrique João Lopes Domingos, Associate Professor, FCT-NOVA

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2021

This document was created using the (pdf)LaTeX processor, based on the NOVAthesis template, developed at the Dep. Informática of FCT-NOVA by João M. Lourenço.

To my family.

Acknowledgements

First and foremost, I would like to express my sincere appreciation to my supervisor, Prof. Henrique João Domingos, for his guidance at every stage of the work and for sparking my interest in the field. Without him, this work would never have been possible. I would like to thank the NOVA School of Science and Technology for granting me a research scholarship that provided me with the financial resources to complete this work. I would also like to thank every professor who has been part of my five-year journey, during which I have learned a lot.

I would like to express my endless gratitude to my family: my parents and grandparents for their unconditional support and tremendous patience with me, my brother who, despite being so far away, was always available for the late night video games that distracted me from work, and my uncle who reminded me that there is no right path. Last but not least, I would like to thank my friends and colleagues who were always there when I needed them.

Abstract

Blockchains have the goal of promoting the decentralization of transactions in a P2P-based internetworking model that does not depend on centralized trust parties. Along with research on better scalability, performance, consistency control, and security guarantees in their service planes, other challenges aimed at better trust decentralization and fairness models on the research community's agenda today.

Asymmetric cryptography and digital signatures are key components of blockchain systems. As a common flaw in different blockchains, public keys and verification of single-signed transactions are handled under the principle of trust centralization. In this dissertation, we propose a better fairness and trust decentralization model by proposing a service plane for blockchains that provides support for collective digital signatures and allowing transactions to be collaboratively authenticated and verified with group-based witnessed guarantees. The proposed solution is achieved by using resilient group signatures from randomly and dynamically assigned groups. In our approach we use Threshold-Byzantine Fault Tolerant Digital Signatures to improve the resilience and robustness of blockchain systems while preserving their decentralization nature.

We have designed and implemented a modular and portable cryptographic provider that supports operations expressed by smart contracts. Our system is designed to be a service plane agnostic and adaptable to the base service planes of different blockchains. Therefore, we envision our solution as a portable, adaptable and reusable plugin service plane for blockchains, as a way to provide authenticated group-signed transactions with decentralized auditing, fairness, and long-term security guarantees and to leverage a better decentralized trust model. We conducted our experimental evaluations in a cloud-based testbench with at least sixteen blockchain nodes distributed across four different data centers, using two different blockchains and observing the proposed benefits.

Keywords: Blockchains, Decentralized Ledgering, Decentralized Trust, Threshold Signature Schemes, Group-Based Signatures, Certification Authority, Byzantine fault-tolerance

Resumo

As blockchains tem principal objetivo de promover a descentralização das transações numa rede P2P, baseada num modelo não dependente de uma autoridade centralizada. Em conjunto com maior escalabilidade, performance, controlos de consistência e garantias de segurança nos planos de serviço, outros desafios como a melhoria do modelo de descentralização e na equidade estão na agenda da comunidade científica.

Criptografia assimétrica e as assinaturas digitais são a componente chave dos sistemas de blockchains. Porém, as blockchains, chaves públicas e verificações de transações assinadas estão sobre o princípio de confiança centralizada. Nesta dissertação, vamos propor uma solução que inclui melhores condições de equidade e descentralização de confiança, modelado por um plano de serviços para a blockchain que fornece suporte para assinaturas coletivas e permite que as transações sejam autenticadas colaborativamente e verificadas com garantias das testemunhadas. Isto será conseguido usando assinaturas resilientes para grupos formados de forma aleatória e dinamicamente. A nossa solução para melhorar a resiliência das blockchains e preservar a sua natureza descentralizada, irá ser baseada em assinaturas *threshold* à prova de falhas Bizantinas.

Com esta finalidade, iremos desenhar e implementar um provedor criptográfico modelar e portável para suportar operações criptográficas que podem ser expressas por smart-contracts. O nosso sistema será desenhado de uma forma agnóstica e adaptável a diferentes planos de serviços. Assim, imaginamos a nossa solução como um plugin portável e adaptável para as blockchains, que oferece suporte para auditoria descentralizada, justiça, e garantias de longo termo para criar modelo melhor da descentralização da base de confiança. Iremos efetuar as avaliações experimentais na cloud, correndo o nosso plano de serviço com duas implementações de blockchain e pelo menos dezasseis nós distribuídos em quatro data centres, observando os benefícios da solução proposta.

Palavras-chave: Blockchain, ledger distribuído, descentralização da confiança, assinaturas threshold, assinaturas por grupo, Autoridade de Certificação.

Contents

List of Figures

List of Tables

Listings

Glossary

Application Specific Validation	Term used in this work to describe a business logic validation written in the smart contract.
Byzantine Fault	A fault that occurs when a node deviates from the defined protocols. This includes sending arbitrary messages to other nodes, Crash Faults and Omission Faults.
Crash Fault	A fault that occurs when a node stops stops working.
Dealer	Entity responsible for generating and distributing the Private Key Shares in a Threshold Signature protocol to the Witnesses.
Digital Signature	Signature created by a Private Key tenant using a mathematical algorithm. It guarantees integrity, pseudonymity, non-repudiation, authenticity of signed data.
Fault-Tolerance	When a node can continue to work correctly in the presence of different types of faults.
Off-Chain Smart Contracts	Smart contract where the code is stored outside the blockchain ledger.
Omission Fault	A fault that occurs when messages are not delivery to a node on time or at all.
On-Chain Smart Contracts	Smart contract where the code is stored in the blockchain ledger.
Open-Membership Blockchain	The same as Permissionless blockchain.
Permissioned	A system where the participants need to be authenticated. The number of participants is fixed.
Permissionless	A system that new participants can join without being authenticated. New participants can join at any time.
Private Key	A cryptographic key that must be kept secret and used to sign data. The result of this process is a Digital Signature that can be validated using the corresponding Public Key.

Private Key Share	A part of a Private Key in a Threshold Signature scheme. Used to sign data, producing a Signature Share.
Public Key	A cryptographic key used to validate a Digital Signature produced by the corresponding Private Key. Identifies a node unequivocally.
Random Oracle Model	Used to prove that the cryptographic protocol is correct without considering a particular hash function to be used.
Robust Threshold Scheme	A threshold scheme that can withstand against Byzantine Faults in the aggregation phase.
Security Specific Validation	Term used in this work to describe a validation that is written in a smart contract to determine how or whether to sign a transaction.
Share	Used to describe a part of something in a threshold signature. See Signature Share and Private Key Share.
Signature Share	A part of a signature in a Threshold Signature scheme. The Dealer or a player can combine multiple signature Shares to create the final Digital Signature.
Threshold Signature	A signature created by a group of Witnesses, where each group member is responsible for creating a Signature Share that is then combined with others to form the final Digital Signature. This type of signatures withstand Crash Faults and Byzantine Faults.
Witness	Participant in the validation of a message/transaction.

Acronyms

API	Application Programming Interface
BFT	Byzantine Fault Tolerance
CA	Certification Authority
CPU	Central Processing Unit
DHT	Distributed Hash Table
DKG	Distributed Key Generator
DPoS	Delegated Proof-of-Stake
EVM	Ethereum Virtual Machine
HLF	Hyperledger Framework
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
JCE	Java Cryptography Extension
LPoS	Leased Proof-of-Stake
P2P	Peer-To-Peer
PBFT	Practical Byzantine Fault Tolerance
PKI	Public Key Infrastructure
PoET	Proof-of-Elapsed-Time
PoS	Proof-of-Stake
PoW	Proof-of-Work
PPoS	Pure Proof-of-Stake
RAM	Random Access Memory
REST	Representational State Transfer

RTT	Round-Trip Time
SGX	Software Guard Extensions
SMR	State Machine Replication
TEE	Trust Execution Environment
THS	Threshold Signature
TLS	Transport Layer Security
TTL	Time to Live
UUID	Universally Unique Identifier
VM	Virtual Machine
VPS	Virtual Private Server
ZPoS	Anonymous Proof-of-Stake

Chapter 1

Introduction

1.1 Context and Motivation

In the past decade, blockchains have played an important role in society. It all started in 1991, when Stuart Haber and W. Scott Stornetta [33] introduced the concept of linking documents cryptographically to create a secure system in which the timestamp could not be forged. However, it was forgotten. Almost two decades later, in 2008, the term and concept of the blockchain was indirectly coined by Satashi Nakamoto, for the crypto-currency Bitcoin, with which Satoshi wanted to disrupt the centralized nature of the financial institutions [45].

Blockchain's are a disruptive technology, as can be seen by the architectural design of having a ledger of transactions distributed across different nodes. One of the major topics being investigated in this field is property of achieving decentralization. In 1979, Adi Shamir already understood the danger of having the trust deposited in a single centralized authority. In his paper [53] he discussed the possibility of breaking a secret into pieces to prevent nodes from compromising the system when they become Byzantine. His idea worked so well that it could avoid any loss of information in case of a system failure, by dividing the data into more pieces that were needed, it would only be necessary a threshold of pieces to rebuild the information.

Centralized Authorities have been on the eye of attackers, therefore, the high value of compromise such systems is unavoidable. Take DigiNotar and other certification authorities as an example [6, 7, 32, 60], both have been compromised, causing attackers to issue certificates from well-known companies such as *.google.com. This resulted in clients to interact with the wrong server, and after the attack was discovered, every valid certificate signed by one of these certification authorities was revoked. To prevent this problem, browser manufacturers began pinning public keys [21], which meant that public keys of

certain organizations were built into the browser. However, this solution is impractical and not scalable because thousands of certificates are issued and revoked daily, which would result in browsers to deliver hourly updates. Other solutions have been developed to mitigate the problem, such as attaching the public key when a client first sees the certificate.

It may seem that these kinds of attacks just happen with certification authorities, but that is not true. Any centralized authority can suffer similar attacks and any service that depends on those authorities can also be compromised. Similar to the secret-sharing by Shamir, one solution to decentralize trust is to introduce group-based signatures where all or a group of participants would sign the same data.

This type of signature has many applications, thus, let's take as an example the work developed by Gennaro [27], the author uses threshold signatures to improve the security of Bitcoins where several user devices would be used for signing. Another example is what we propose, which is to use group-based signatures to decentralize trust in a blockchain [56], although this work lacks dynamism when it comes to giving the user a way to configure what type of signatures to use.

1.2 Objectives and Contributions

Problem Statement: Current blockchains rely on the certification authority to sign the intervenients' public keys and, as a result, some participants nodes are assigned with too much power, leading to a high value target for attackers. There are proposals to enhance blockchain decentralization [56] by using threshold-signatures, however this proposals can only be applied in permissioned blockchains and cannot dynamically configure itself, so that the user has the possibility to choose for each transaction each type of signature he wants to use. The ability to configure the signature type is a must in a blockchain. This flexibility allows us to use the best signature type for a particular purpose, or if a particular signature type is outdated, the signature can be exchanged during operation.

This leads to the following problem:

> *How can we use group-based signatures and smart contracts to design a more reliable and dynamic blockchain, improving trustworthiness and decentralization without compromising transaction throughput?*

Main objective: In this dissertation, we design, prototype and evaluate a shared signature plane that can be integrated into different blockchains, such as permissionless and permissioned, allowing a new level of decentralization and Byzantine fault-tolerance in signatures. We use group signatures, more precisely threshold signatures in a dynamic way, which means that we will have a crypto provider with different types of signature

implementations, allowing the user to dynamically choose which scheme to use to sign the transaction, including conventional schemes. We also develop this crypto-provider so that it can be easily extended by someone who wants to integrate new schemes in the future. The smart contract in our work plays an important role in creating the configuration dynamics that we offer.

To address our objectives, we design our solution according to the following system goals:

- **Auditability** - Auditable under verifiable immutable group-signatures, recorded in a tamper-resistant log.
- **Resiliency** - Resilient byzantine-fault tolerant group witnessed signatures.
- **Fairness and Dynamic Reconfigurability** - A fair and dynamically reconfigurable cooperative signing trustee group with self-sovereign identities, reconfigurable via possible off-chain or sharded-chain membership management.
- **Pluggability and Factoring Modularity** - Reusable and pluggable modular cryptoprovider provisioning, adopting a factory-object oriented design allowing the dynamic addition of different Byzantine Fault Tolerance (BFT) threshold digital signatures' algorithms.
- **Portability** - Portable and agnostic solution, designed as a reusable service plane independent from other blockchain-based service planes, allowing to be deployed as a hybrid-usable solution for permissionless and permissioned decentralized ledgers.
- **Decentralization** - Compatible with a fully decentralized model, not compromising the inherent trust-decentralization requirements.

Contributions: In concordance with the main objective, we enumerate the relevant contributions provided in this dissertation:

- Study a range of blockchain platforms, properties and their smart contract implementation to select the most relevant permissionless and permissioned blockchain to support the development of this work;
- An analysis of various resilient group-based signature schemes and their properties, benchmarking threshold signature implementation to select some implementations to be included in our crypto-provider;
- Study and analyze which component can take advantage of being decentralized, finally designing a system model, including the chosen technologies to best accomplish the objective of decentralizing the blockchain;
- Design and implement a highly modular crypto-provider capable of supporting new, different signature schemes given our programming interface;

- Create a reliable and fully functional prototype that addresses the system goals and features defined above leverage by two different platforms: Algorand[1] and Hyperledger Sawtooth[2];
- Design and implement a smart contract extension for a permissioned blockchain and another for a permissionless that allows blockchain settings to be changed dynamically, such as the signature scheme used;
- Validation of the implemented prototype, performing different experimental evaluations (i) to analyse the performance, portability and modularity of the cryptographic provider and group signature service plane, (ii) in-depth analysis of the performance of different configurations for key sizes, group sizes and cryptographic constructs, (iii) impact of the solution in the observation of the performance, including transaction throughput and latency of operations and scale conditions. Our validation analysis will be performed by using our prototype through a permissioned ledger solution and through a permissionless ledger system.

1.3 Report Organization

The remaining chapters of this document are organized as follows:

- **Chapter 2** introduces fundamentals concepts in a top-down approach. We start the chapter by studying the different types of group signatures and understanding the key features that make them useful for decentralizing signatures. We then examine the different planes of the blockchain and show what has been done to decentralize signatures. Finally, we study smart contracts and the properties that can help us provide dynamic configurations.
- **Chapter 3** discusses the system model and architecture and the basic components to build a shared component to be integrated in the different blockchains.
- **Chapter 4** describes the implementation and technologies used to build a fully functional prototype based on the system model from the previous chapter.
- **Chapter 5** presents the experimental evaluation and a critical analysis of the results for the prototype described earlier.
- **Chapter 6** concludes the document with a series of final remarks and presents some open questions and future work.

[1] **Algorand**: https://github.com/algorand/go-algorand
[2] **Sawtooth:** https://sawtooth.hyperledger.org/docs/core/releases/1.2.6/introduction.html

Chapter 2

Related Work

In this chapter, we present the required related work to our dissertation. We begin by familiarizing the reader with group signatures and their benefits, introducing the different types of group signatures schemes, and explaining how we can use threshold signatures to decentralize trust in a blockchain. We also benchmark threshold signatures to show that decentralizing trust has a cost. Next, we introduce the concept of the blockchain, explaining some of the different planes that exist and then we analyze the properties of some available blockchain platforms. Finally, we introduce the concept of a smart contract and how it appears in the different blockchain platforms. At the end of this chapter, we conduct a critical analysis of the different topics addressed.

2.1 Group-Based Digital Signatures

In this section we first explain what digital signatures are and why we need group-based digital signatures, then we show examples of different group schemes leading to our scheme of choice. We conclude this section by showing some threshold signatures schemes and some preliminary benchmarks.

2.1.1 Group Signatures

Digital Signature is, as the name suggests, a mathematical scheme that allows participants to sign messages and verify the authenticity of signatures. This scheme consists of three primitives, one for generating a key pair (private key, public key), another for signing messages using the private key and, finally, a primitive for verifying the authenticity of signatures. However, not every scheme containing the preceding primitives can be considered a valid digital signature scheme, since signatures must be forgery-proof if

the private key is not known, must be easily verifiable, and must also have the following properties: (i) integrity, (ii) pseudonymity, (iii) non-repudiation, (iv) authenticity.

The use cases for this type of signatures are enormous, they are used in PKI's, Certification Authority (CA)'s and many other places, although each of these use cases is considered centralized, which is not good, since the proper functioning and security of digital signatures is stored in a unique private key. At this point we might wonder what happens if the private key is disclosed, but unfortunately we already know the result [6, 7, 32, 60].

By now we all agree that one of the shortcomings of digital signatures is the lack of decentralization, because a single point of failure can easily compromise any system. To distribute trust among the different peers, David Chaum and Eugene van Heyst in 1991 [10] proposed group signatures. In their proposal, only members of the group would be able to sign, and the receiver could verify that the message was signed, but could not verify that a particular member of the group signed. If the group consists of more than one element, it is easy to verify that the attacker must compromise more than one participant node to compromise the security of the system, although we cannot determine the exact number of nodes because it depends on used group signature scheme.

Since then, various group schemes have been proposed, such as ring signature [52], multisignature [3],and threshold signature [16].

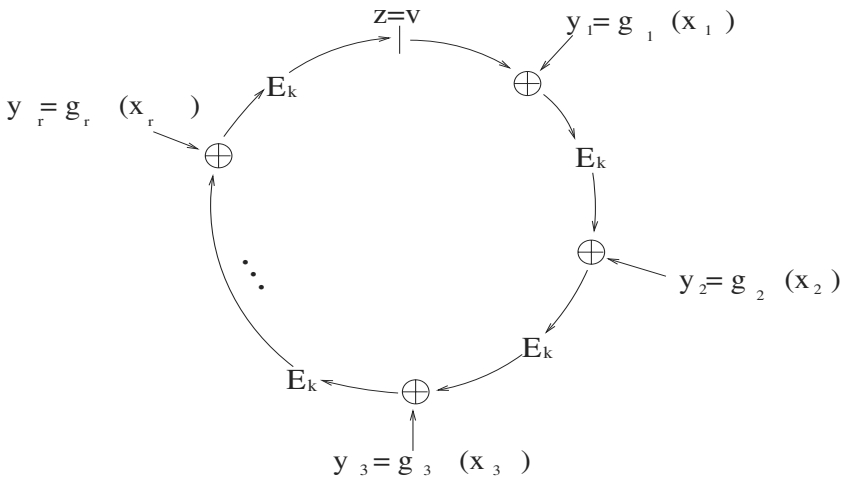


Figure 2.1: Ring Signature (from [52])

Ring Signature was first formalized in 2001 by Rivet, Shamir and Tauman. In this signature scheme [52], the signer of the message chooses other nodes to be part of the signature, although to include other members in the signature, the signer does not need to ask for approval, as only the node's public key is required. This type of signature is used when the nodes do not want to cooperate with each other and the signer of the message wants to be anonymous. To generate a signature in this scheme, the signer will generate the entire signature himself using any number of public keys. Figure 2.1 shows the ring equation that the signer must calculate to create a y_s that is used to calculate the signer's

x_s, the other x_i are all randomly calculated and used in the g trapdoor function. The generated final signature consists of all public keys, a glue value v and all x_i including the signer x_s , $(P_1, P_2, \ldots, P_r; v; x_1, x_2, \ldots, x_r)$.

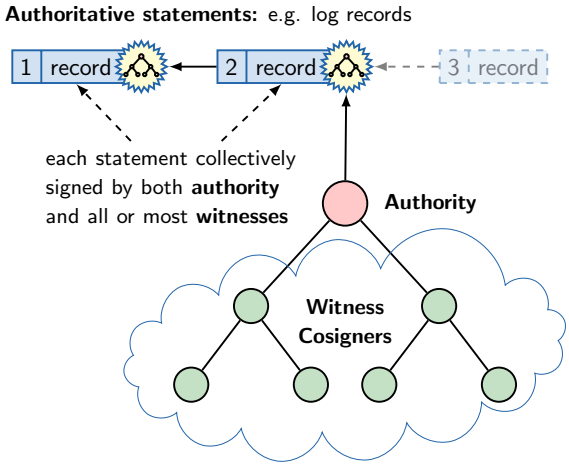


Figure 2.2: CoSi architecture (from [58])

CoSi [58] is an optimized **multi-signature** protocol that scales in the presence of many signatories. To achieve this scalability, CoSi implements Schnorr multi-signatures [48] with the combination of multicast tree based dissemination protocol. The CoSi consists of an authority/leader and witnesses who are committed to validate and co-sign the message, as shown in figure 2.2. The protocol starts when the leader broadcasts an announcement to the tree of co-signers, then each node i will select a random secret v_i and calculate a Schnorr commitment. In the same phase nodes from the tree will get the values from their children and calculate the aggregated commitment. Next, the leader computes a collective Schnorr challenge with the statement S and sends it down the tree. Finally, using the collective challenge, the nodes calculate their own response and compute an aggregated response from the bottom up. The result of this protocol will be a Schnorr signature of 64 bytes.

THS [16] also called (t,n)-threshold, is a cryptographic scheme that allows the computation of digital signatures within a group. By distributing this task among different participants, we also distribute trust among the participants. In THS, only t members of a group of size n need to sign, this means that Byzantine members need t to forge the signature, therefore $t-1$ cannot sign a message. The THS scheme is initialized by splitting the private key (pk) into n pieces called secret shares (pk_i), then by distributing somehow the secret shares among all the members of the group. The protocol starts when the group decides to sign a message M. First, each participant creates a signature share $(Sig_{pk_i}(M) = ss_i)$ and, finally, the shares have to be aggregated to build the complete signature $(SigShare = \{ss_1, ss_2, ss_{n-1}, ss_n\} If \#SigShare < t$, no signature is produced, else $signature = \sum_{i=1}^{\#SigShare} ss_i$). The verification of the signature is transparent to the verifier because there is only one public key and we do not know that multiple participants have

signed this message.

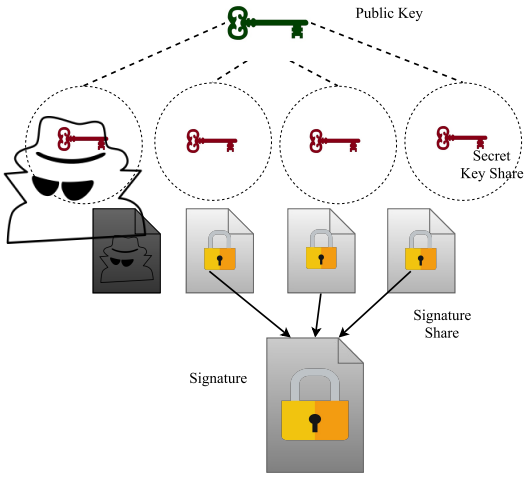


Figure 2.3: Threshold Signature Architecture with a Byzantine member (from [56])

Figure 2.3 represents a group of members trying to sign a message, but only 3 out of 4 members have to sign. In this case a Byzantine member refused to sign, but the signature is done anyway. From this we can conclude that one of the most important features of THS is its robustness, because when t Byzantine members try to collude with each other, they cannot prevent the signature from being created. Another important property is proactiveness, the members of the group update their private key shares to prevent an attacker from building up information about the system (e.g., if the key shares are updated every 24 hours, an attacker must collect more than t shares to control the group in 24 hours, which makes it difficult).

So far, we have studied various group signatures, but for the purpose of this work, not all of them have the properties we were looking for. We began this section by stating that we need a group of signers that covers the flaw of a unique signer, although ring signatures are not suitable, since only one member of the group will create the signature using the public keys of other forced members. This type of signature is more likely to be used when a signer wants to hide the identity but wants to prove that the message is authentic. A naive solution to our problem would be to use multi-signatures, in this case the trust is distributed among the various signers. However, this solution is not scalable, since validation requires a signature and a public key for each new member, and instead of a single point of failure, we now have a point of failure for each member of the group; if one of them fails, no signature can be produced. CoSi is a variant of the multi-signature, but is carried out in rounds that lead to latency due to communication costs. CoSI does not specify the threshold of signers required for the protocol, and the authority is considered a single point of failure. With this, the threshold signature will be used in this work, due to its properties and the choice to distribute trust for our work. In the remaining of this chapter, we will discuss different schemes for THS.

2.1.2 Threshold Based Signatures Schemes

There are types of THS schemes that rely on a trusted dealer [54] to distribute key shares, although not every system is capable of tolerating such a threat model. So there are THS schemes that rely on a distributed key generation [5, 28, 57]. A Distributed Key Generator (DKG) is a protocol in which each participant contributes to the generation of private key shares, and, at the end of the protocol, each participant holds only the share that belongs to him and a public key for the group share. In a THS scheme that does not use DKG, only the dealer needs to be compromised in order to compromise the entire THS signature, but when using DKG, more than a certain threshold number of participants must be compromised to defeat the system security. A DKG was present by Pedersen [49], although in the presence of Byzantine participants the resulting private shares may be biased. To solve this problem, Gennaro [29] presented another DKG, but it is twice as expensive to generate the secret shares, since Peterson's DKG is not interactive in the absence of errors and, for Gennaro DKG, two round trips are required. DKG's have some advantages, although they incur communication costs. For some use cases it is necessary not to have a trusted dealer, but for others, who in their threat model tolerate not having a trusted dealer, it is more efficient not to have DKG.

The schemes we are looking for must be robust. A robust threshold scheme ensures that if t members of the group follow the protocol, a signature will be produced even if the other $n-t$ members do not follow the protocol and contribute with erroneous signatures shares. On the other hand, a non-robust threshold scheme aborts when a faulty share is detected. In this work we are not interested in non-robust schemes [14, 26], but if necessary, we can process the shares with an algorithm to select only valid shares. We will now focus on RSA [54], BLS [29], DSA [28],and Schnorr [57].

The RSA threshold signature scheme proposed by Victor Shoup [54] is a scheme that is considered robust and forgery-proof in the random oracle model. One of its major advantages is that it is not iterative and takes advantage of the RSA modulus problem, which has already proven to be hard and robust.

The BLS [5] signature scheme was proposed by Bohen, Lynn and Shacham with the aim of creating a signature of small size that can be typed by humans or to use low resources in a computer. It is considered forgery-proof in a random oracle model. BLS has some useful and interesting properties such as being aggregatable, which gives the possibility to aggregate any primitive (secret keys, public keys, signatures), e.g., to aggregate n private keys and sign a message, it can be verified with the appropriate aggregation of n public keys. When BLS threshold signature is applied, no trusted dealer is required to generate the private key share, since n participants can work together to generate the shares [29].

The threshold DSA scheme [28] proposed by Gennaro, Jarecki, Krawczyk and Rabin does not rely on a trusted dealer to distribute the key shares among the participants. The security of this scheme is not proven by a random oracle. The robustness of this signature

scheme is also achieved by using an error correction technique developed by the authors Berlekamp and Welch [42].

The threshold Schnorr scheme [57] proposed by Stinson and Strobl transforms Schnorr scheme into secure and robust Distributed Schnorr scheme where n players would participate to generate the private key shares and the public key.

2.1.3 Summary

Table 2.1 shows the results of a preliminary benchmark performed with some threshold signatures found in Golang and for signatures without threshold. The benchmark was performed in a MacBook Pro with MacOS 11.0, Darwin 20.1.0 kernel, 16GB 2400 MHz DDR4 and 2,6 GHz Intel Core i7 with 6 cores. The table is divided into 4 phases of the THS: first the phase of the key generation; second the signing phase (for THS the signing consists only of signing with a share); third and only for THS the phase of aggregation signature share and finally the phase of verification of the signature. Each of these phases is divided into 3 benchmark values, one for the time for each operation in milliseconds , another for the number of bytes allocated in the heap for each operation (B/op) and finally the number of allocations in the heap (allocs/op). We decided to present these values to better choose the cryptographic scheme we will use during the implementation and to show the impact of switching from a conventional signature to an THS. We must note, however, that these tests have been performed in a single machine with the above specification, which means that the latency to a dealer distributing shares and, in the case of DKG, the generation of the shares is ignored.

Analyzing the results, it is obvious that the complexity of any THS scheme depends primarily on the key share generation, since some parameters are generated on the fly and it is necessary to generate not only one key, but n keys (e.g., RSA generates a 1024-byte safe prime pair, which takes a long time to generate). We can also see that when using THS schemes, we have to pay the cost of decentralized trust and have more points of failure, e.g., THS RSA 3072 takes 36 ms to sign a message, while a non-THS scheme with the same key size takes only 3.7 ms. For BLS and Schnorr, the time required for signing is almost the same, but for key generation the times are larger. It is also easy to see that for the THS schemes we compared, the non-THS scheme is used to verify the signature of an THS scheme, this means that after signature aggregation, the verification is transparent, e.g., we can use an RSA scheme to verify a t-RSA signature.

The impact of THS key shares generation together with the number of generated shares and distribution of such shares by a group of signers in the context of a Blockchain Co-signing environment is one of the relevant challenges that must be addressed in our design model. This will be an important evaluation factor for the validation of our proposed solution.

Signature Schemes	Key Gen [1]			Sign [2]			Aggr. Sig. Shares [3]			Verify Signature		
	Time ms/op	Allocated Bytes bytes/op	Allocation Operations allocs/op	Time ms/op	Allocated Bytes bytes/op	Allocation Operations allocs/op	Time ms/op	Allocated Bytes bytes/op	Allocation Operations allocs/op	Time ms/op	Allocated Bytes bytes/op	Allocation Operations allocs/op
t-RSA 1024 [46]	3340	216834878	820406	2.4	27510	477	0.25	24677	541	0.02	2633	11
RSA 1024	18	1045203	3990	0.30	16134	103	-	-	-	0.02	2633	11
t-RSA 2048 [46]	102186	1939336100	5058227	12	58524	866	0.6	48791	935	0.06	5195	11
RSA 2048	147	2695898	7060	1.37	30654	109	-	-	-	0.06	5195	11
t-RSA 3072 [46]	132014	1197593400	2439053	36	81061	1228	0.9	103102	1292	0.11	11856	12
RSA 3072	564	4815592	9846	3.7	57631	117	-	-	-	0.1	11856	12
t-BLS 256 [19]	1.21	8654	107	0.19	6427	248	8.5	335964	3610	2.7	108803	1062
BLS 256	0.42	2305	15	0.23	7435	357	-	-	-	2.81	2814143	1163
t-Schorr 256 [15]	257.3	2019614	25939	0.21	5237	72	37	79825	1142	0.35	1600	16
Schorr 256	0.07	1376	15	0.21	3814	47	-	-	-	0.33	1664	18
t-ECDSA P-256 [47]	1472	5737157	11745	3388	10240823	15332	[4]	[4]	[4]	0.07	928	16
ECDSA P-256	0.003	608	12	0.005	2639	32	-	-	-	0.02	825	16
t-ECDSA P-384 [47]	4446	234255153	1876417	9634	54981570	337937	[4]	[4]	[4]	8.5	3476539	28736
ECDSA P-384	1.1	1742716	14389	1.2	1747066	14429	-	-	-	2.2	3480334	28752
t-ECDSA P-521 [47]	9941	389220026	2456671	22755	86319572	437160	[4]	[4]	[4]	13	5930354	38397
ECDSA P-521	1.8	3021289	19542	1.9	3027865	19589	-	-	-	3.7	6121331	39618

[1] For threshold signature is the generation of the private shares [2] For threshold signature is signing with a share [3] Only applicable to threshold signatures
[4] Not applicable. ECDSA is composed of four rounds in which will result an aggregate signature

Table 2.1: Benchmark to different schemes. The benchmark Time is in ms/op, *Allocated* Bytes is the number of bytes allocated on the heap per operation, *Allocation Operation* is the number of allocations on the heap per operations.

Table 2.2 summarize the properties and time complexity for the different Threshold Signature Schemes.

Table 2.2: Characteristics for different THS schemes

	RSA	**Schnorr**	**BLS**	**ECDSA**
Recommended key size	2048 bits	256 bit	256 bits	256 bits
Non-threshold variant	✓	✓	✗	✓
Sig. Size (bytes)	128	64	33	64
Signing Time Complexity	High	Low	Low	High
Verifying Time Complexity	Low	Low	High	High
Key Generation	Trusted Dealer	DKG	Membership	DKG
Setup Time Complexity	High	High	Medium	High

We can deduct from the table 2.2 that most of these threshold signatures can be used in our work to decentralize trust, although two THS schemes stand out. First, despite the high time complexity of signing and setup, RSA is the fastest one for signature verification and is not interactive in generating key and signature shares, but the proposed key size is currently 2048 bits, and for some use cases no trusted dealer can be used. Finally, BLS has small key sizes (recommended 256 for the same security as RSA) and supports distributed key generation [29].

2.2 Blockchains and Group-Based Signatures

In this section we first explain the blockchain technology, consensus protocols and relevant concepts, then we show what has been done so far to apply group signatures to permissionless and permissioned blockchains. We will start with Bitcoin as an example to illustrate, but in this work we will focus on blockchain technologies.

2.2.1 Blockchains

The concept of the blockchain was first introduced in 1991 by Stuart Haber and W. Scott Stornetta [33]. Their vision was to use this technology as a way to time-stamp digital documents so that documents could not be forged. But it was not until 2008 that blockchains began to become popular, as Satoshi Nakamoto introduced Bitcoin, a revolutionary "electronic payment system based on cryptographic proof rather than trust..."[45] with which Satoshi wanted to decentralize the financial institution. Since then, the use cases of blockchains have spread to a variety of domains [63], such as supply chain management, digital identity healthcare, notaries and many others. With so many use cases yet to be discovered, high expectations are placed on blockchains, leading to a race for new use cases with the development of new blockchain platforms.

Blockchains were invented to create a distributed database in which no central authority was required for the system to function properly. We can describe blockchains

more scientifically as a distributed database that contains a list of blocks in which the blocks are cryptographically linked together, meaning that once the blocks are created, they cannot be changed or forged making blockchain an immutable distributed ledger. As can be seen in the definition we will have participant nodes executing well established protocols in order to this Peer-To-Peer (P2P) technology work. Consensus protocol and validation protocols are some of the protocols the nodes need to run, thus, we will discuss more about this later.

2.2.1.1 Blockchain Foundation

A blockchain can be divided into different components that must exist. Three of the most important components are (i) immutable ledger, (ii) consensus algorithm, (iii) transaction validation algorithms. Although we will not focus on validation algorithms in this subsection, since different validations are performed depending on the blockchain platform and business logic. Smart contracts are discussed later in the section 2.3.

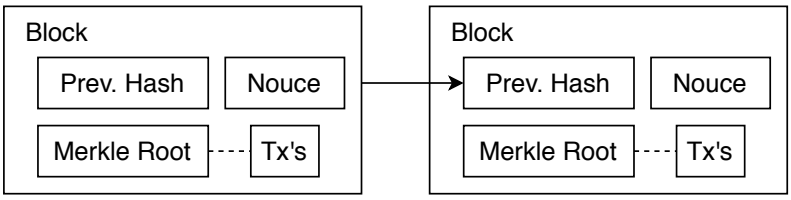


Figure 2.4: Blockchain block structure (adapted from [45])

Blockchains ledger: To explain this component of a blockchain, it is easy to start by explaining the life cycle of a transaction. When the clients are not directly involved in the blockchain and want to submit a transaction, they first start by sending the request to a blockchain node. Then, using a gossip protocol in a P2P network, the node disseminates the transaction to reach a majority of the participating nodes. There are some blockchains in which the submitted transactions remain in a transaction pool until they are selected for integration into a block. Considering that we are in a blockchain with a proof-of-work consensus, the miners select the transactions with a higher fee to integrate them into a block, but before they do, the nodes must perform some validation of the transaction. A block as can be seen in figure 2.4 is built by a set of transactions, the previous block hash, and by a nonce. The genesis block is the unique block in the chain that does not contain a previous hash. In Proof-of-Work (PoW) consensus, the nonce is calculated by finding a number in which leads to the current block hash having a pre-fixed number of leading zeros. After finding the golden nonce, the block is considered to be part of the chain. If someone later tries to temper with a particular block, he must temper with the entire blockchain that follows the block to be tempered. This is a difficult task, because when using PoW all nonce's have to be recalculated. There is another inherent complication: the chain is distributed, so the attacker must corrupt a high percentage of the total peers, in the case of PoW, 51%.

Blockchain consensus protocol: Consensus protocols have long been studied and are a fundamental part of the blockchain, because to achieve State Machine Replication (SMR), replicas must agree on an order in which operations are performed. The operations in a blockchain are the transactions. We want to guarantee that a set of transactions are applied in the same way in all nodes to maintain a consistent transaction ledger. Consensus protocols must demonstrably adhere to certain properties to be considered valid [13], these properties are: a) termination - every correct process eventually decides a value; b) validity - if a process decides v, then v was proposed by some process; c) integrity - no process decides twice; d) agreement - no two correct processes decide differently.

In presence of a bounded number of faults, consensus protocols must ensure consistency in the ledger, so there are different types of consensus for different types of faults such as crash-fault, Byzantine faults [41] and message omission. However, the properties above only work when faults occur in a synchronous system and these types of systems don't exist. When applying the consensus in an asynchronous system, the FLP impossibility [24] emerges, FLP states that there is no deterministic protocol that solves the consensus in an asynchronous system where a single process may fail. To circumvent FLP consensus protocols, the specifications have been relaxed, e.g., Paxos [40] relaxed the termination property based on probabilistic calculations and in certain circumstances consensus may not be reached.

There are thousands of consensus protocols that are ready to be used in blockchains, although when choosing a consensus algorithm, several considerations must be made, such as the type of the underlining network, what fault-tolerance assumptions the system makes, throughput, power consumption, etc. Some of the most used consensus algorithms in blockchains today are [23]:

- **Proof-of-Work (PoW):** Proof-of-Work was proposed by Dwork and Naor in 1993 [18] to combat email spam by introducing computational effort on the part of the sender. To send an email, the sender would have to compute a resource-intensive puzzle to which the proof would have to be attached. Upon receiving the email, it would be necessary to verify the solution. The idea behind their proof-of-work later led to the most used consensus in blockchain, in which two different participants would be required: the miners, who would solve the puzzle to find the golden nonce, and the verifiers, who would check that the nonce respected the number of leading zeros. PoW now comes in two variants, computer-bound PoW, which requires only a processing unit to solve the puzzle, and memory-bound PoW, which requires access to memory, which introduces an upper limit due to the latency when accessing RAM. Usually this type of consensus is used in a permissionless blockchain and the miners are given an incentive to continue mining and contribute to the creation of new blocks, e.g., Bitcoin uses a CPU-bound PoW, and for every golden nonce found by the miners, 6.25 Bitcoins [4] are released to them.

- **Proof-of-Stake (PoS):** The PoW consensus has some limitations due to low scalability, high computing power and power requirements. To mitigate some of these limitations, proof of stake was introduced in a forum in 2011 [50]. PoS is a consensus protocol in which the block creators and validators have to block part of their wealth (so-called stake). Once they have deposited some of their coins in an escrow account, they are ready to be considered stakeholders and can be selected to participate in the creation of blocks. In case of misconduct, a stakeholder may lose all or part of the deposited coins. The purpose of this consensus protocol is to prevent participants from not following the rules. PoS also comes in different variations like, Delegated Proof-of-Stake (DPoS), Leased Proof-of-Stake (LPoS), Anonymous Proof-of-Stake (ZPoS) and Pure Proof-of-Stake (PPoS).

- **Proof-of-Elapsed-Time (PoET):** Another approach to reach a consensus without consuming too much computing power and energy is to use PoET. This consensus was invented by Intel to take advantage of its Trust Execution Environment (TEE), the Software Guard Extensions (SGX). At a high level, this protocol requires a validator to execute a lottery function within a TEE and wait for the time returned by that function. PoET is considered secure because everything takes place within a secure enclave and the lottery function must guarantee fairness and be easy for any participant to verify that a validator is respecting.

- **Practical Byzantine Fault Tolerance (PBFT):** In 1999, Miguel Castro and Barbara Liskov introduced the PBFT [9] consensus protocol which can withstand Byzantine faults. A client starts the normal use case of the protocol by sending a request to the primary. Then, the primary multicast requests (pre-prepare) to all participants in the PBFT, at this moment the replicas will exchange 2 rounds of message (the prepare and commit) to guarantee that no Byzantine replica exists. Finally, the protocol terminates when the client receives f + 1 replicas results. This protocol improves the performance of the blockchains comparing to the PoW although it contains some scalability problems.

Types of Blockchains: The literature divides blockchains in three groups [8]:

- **Permissionless**: In a permissionless or public blockchain, anyone who wishes to participate can take part either as a miner, or a user, or both. Typically, these types of blockchains attempt to maintain anonymity and use a consensus protocol that requires a certain amount of a resource such as currency or computing power.

- **Permissioned**: In a permissioned or private blockchain, only authenticated users can participate in the creation of the chain, although in some cases not all nodes in the blockchain are involved because of different roles. One of the consensus protocols used is PBFT.

- **Federated**: A federated blockchain is a combination between permissionless and permissioned blockchains. The trust in federated blockchains is not deposited in all nodes, nor in a single organization, instead the trust is divided between multiple organizations or multiple nodes.

2.2.2 Permissionless Blockchains with Group Signatures

As we saw in section 2.1, group signatures are a powerful tool widely used, even in blockchain technologies, to completely decentralize the distributed ledger. However, to achieve this in a blockchain, we must have a balance between security, scalability, and performance. In permissionless blockchain this is a more difficult task than in permissioned blockchains, because when group signatures are used for transaction validation, a group of validators must be selected without bias so that no one can take control of the ledger. ByzCoin [37] and Omniledger [39] solve this problem in the same way.

ByzCoin[37] is an example of a blockchain that brings together three different concepts: Bitcoin- NG Eyal2016, PBFT [9] and CoSi [58]. ByzCoin uses a group signature scheme called CoSi with the aim of improving the security and performance of Bitcoin. The confirmation rate of Bitcoin is about 1 hour (6 blocks). By changing the consensus protocol from PoW to PBFT, ByzCoin achieves less than one minute confirmation time. However, PBFT does not scale well, so ByzCoin changes the prepare and the commit phase with some rounds of CoSi group signature.

Another attempt to improve the distributed ledger systems was made by Omniledger [39]. Omniledger is a distributed ledger architecture that offers all the features provided by ByzCoin plus "scale-out". Again, both ByzCoin and Omniledger are trying to improve security, transaction throughput and transaction confirmation time to compete with centralized payment processing systems such as Visa. Omniledger uses a sliding window with the latest proof-of-work miners to select the representatives to participate in the validation of transactions. These validators then use a version of PBFT with group-based signatures (CoSi) to validate transactions without intensive computations, reducing confirmation time and increasing throughput. To achieve scale-out, Omniledger uses shards of validators that are responsible for their own distributed ledger.

Another approach to the introduction of group signatures for permissionless blockchains, in this case Bitcoin, was made by Goldfeder *et al* [30] in their work, which began with the argument that Bitcoin is subject to attacks that result in the loss of cryptocurrency for businesses and individuals, demonstrating that this is usually done by infecting a machine with a virus that contains the credentials for a Bitcoin wallet. They solved this problem by first introducing a threshold signature scheme compatible with Bitcoin's ECDSA wallet and explained that multiple participants are required to conduct Bitcoin transactions in order to be secure. Bitcoin already supports multi-signature, but the threshold signature brings some advantages, such as flexibility, confidentiality, anonymity and scalability. Their latest contribution was a threshold signature based two-factor secure wallet, so

users do not have to store their private keys in a single device.

Calypso [38] is an example of another project that uses threshold cryptography not for signatures, but to encrypt data so they can provide secure, verifiable data sharing over any blockchain. To achieve this, they use threshold cryptography to prevent any node from reconstructing the secret without having the authorization and without having logged the intent to open the secret. Calypso develops two ways for data confidentiality: i) Long-term secrets are used in permissioned blockchain and are composed as follows: the secret data m is encrypted under a symmetric key k which is then encrypted under a public threshold key of the secret-management committee; ii) One-time secrets are similar to long-term secrets but for permissionless blockchain. Here, to send a secret, it is necessary to select a set of members to be the secret-management committee, then the shares are generated and delivered to the respective committee member.

2.2.3 Permissioned Blockchains with Group Signatures

In permissioned blockchains, group signatures for the validation of transactions can be easily applied because a fixed number of validators is preselected and there is little change in this group of validators.

In a research report published by Stathakopoulou and Cachin [56] they described a study implementation that manages to use threshold signatures for transaction validation in the Hyperledger Fabric. The authors chose to use RSA [54] because it is non-interactive, i.e., no interaction is required to produce a signature requiring only the exchange of signature shares. THS RSA is also deterministic and equivalent to RSA signatures generated by a non-threshold. The author also used BLS [5] with some of the features of RSA THS, although BLS has a smaller key size than RSA, and the authors say that in the future they may implement DKG for BLS. With these two THS schemes, the authors concluded that there is no negative impact on performance when the cryptography is changed from a non-threshold to a threshold, but there is a security gain from the distribution of trust.

In Godinho's dissertation [25], he also used THS and applied RSA threshold signatures to Hyperledger Fabric, although he extended the smart contracts so that the consensus protocol and type of signature could be dynamically changed per transaction.

2.2.4 Summary

As we have seen in this section, we can have different blockchain platforms with different combinations of consensus planes, network planes, storage planes and view planes. In Table 2.3 and Table 2.4, we summarize the key characteristics of some blockchain platforms that we have been able to find. It is important to note that the different tradeoffs between consensus mechanisms, decentralization, and throughput. Consensus algorithms similar to proof-of-work suffer from low throughput but a high degree of decentralization, whereas Consortium and PBFT consensus suffer from low decentralization and high throughput.

				Consensus Plane				Storage plane	View Plane			
	Descentr.	**Mult. chain**	**Smart Contracts**	**Mechanism**	**Plug.**	**BFT**	**Throughput scalability**	**Ledger replication**	**View computation**	**Tx. privacy**	**Peer anonym.**	**Sign. Validation** [7]
Quorum	Partial	✗	✓	Consortium	✓	3f+1 [1]	Presumably high (> 1000 tps)	Partial (private smart contracts are only replicated between authorized peers)	Strongly consistent, totally ordered SMR	✓	✓	Conventional
Hydrachain	Partial	✗	✓	Consortium	✗	3f+1	Presumably high (> 1000 tps)	Global	Strongly consistent, totally ordered SMR	✗	✗	Conventional
Hyperledger Fabric	Partial	✓	✓	Off-chain service[2]	✓	3f+1[2]	High (> 1000 tps)	Partial (each channel holds a ledger between a subset of nodes)	Strongly consistent, totally ordered SMR	✓	✗	Multisignatures
Hyperledger Fabric with Group Signatures[6]	Partial	✓	✓	Off-chain service[2]	✓	3f+1[2]	High (> 1000 tps)	Partial (each channel holds a ledger between a subset of nodes)	Strongly consistent, totally ordered SMR	✓	✗	Multi and Threshold Signatures
Hyperledger Sawtooth	Full[3]	✗	✓	PoET / PBFT / Other's [2]	✓	51% [4]	Presumably high (> 1000 tps)	Global	Eventually consistent, causally ordered SMR	✗	✗	Conventional
Hyperledger Burrow	Partial	✓	✓	Consortium	✗	3f + 1	High (> 1000 tps)	Partial (each chain holds a ledger between a subset of nodes)	Strongly consistent, totally ordered SMR	✓	✗	Conventional
Hyperledger Iroha	Partial	✗	✗	Consortium	✗	3f + 1	Presumably high (> 1000 tps)	Global	Strongly consistent, totally ordered SMR	✗	✗	Multisignatures
Hyperledger Indy	Full	✗	✗	RBFT	✗	3f + 1	Presumably high (> 1000 tps)	Global	Eventually consistent, causally ordered SMR	✓	✗	Multisignatures
Openchain	Partial	✓	✓	Partitioned Consensus [5]	✓	n/2 + 1	High	Global	Strongly consistent, totally ordered SMR	✗	✓	Multisignatures
Multichain	Partial	✓	✓	Consortium	✗	Not BFT	High (> 1000 tps)	Partial (each chain holds a ledger between a subset of nodes)	Eventually consistent, causally ordered SMR	✓	✓	Multisignatures

[1] With Istanbul BFT; [2] With the unofficial BFT-SMaRT ordering service presented in [55];
[3] Assuming every node is equipped with Intel's SGX technology. Otherwise decentralization is partial; [4] If using PoET ;
[5] Transactions are validated by different authorities depending on the assets being exchanged. Each organization controls their own instance and each instance has only one authority validating transactions; [6] Version implemented in [56] [7] All blockchains support the normal traditional signature validation.

Table 2.3: Comparison Of Permissioned Blockchain Platforms

				Consensus Plane				Storage plane	View Plane			
	Descentr.	**Mult. chain**	**Smart Contracts**	**Mechanism**	**Plug.**	**BFT**	**Throughput scalability**	**Ledger replication**	**View computation**	**Tx. privacy**	**Peer anonym.**	**Sign. Validation** [5]
Ethereum	Full	✗	✓	PoS	✗	51%	Low (< 100 tps)	Global	Eventually consistent, causally ordered SMR	✗	✗	Conventional
Bitcoin	Full	✗	✓[1]	PoW	✗	51%	Low (< 10 tps)	Global	Eventually consistent, causally ordered SMR	✗	✗	Conventional
Omniledger	Full	✓	✗	PBFT [2]	✗	3f + 1 [3]	Presumably High (> 1000 tps) [4]	Partial (each chain holds a ledger between a subset of nodes)	Strongly consistent, totally ordered SMR	✗	✗	Multisignatures
Tezos	Full	✗	✓	PoS	✗	51%	Presumably Low (< 80 tps)	Global	Eventually consistent, causally ordered SMR	✗	✗	Multisignatures
Byzcoin	Full	✗	✗	PBFT [2]	✗	3f + 1	Presumably High (> 900 tps)	Global	Strongly consistent, totally ordered SMR	✗	✗	Multisignatures
Algorand	Full	✗	✓	PPoS	✗	51%	Presumably High (>900 tps)	Global	Strongly consistent, totally ordered SMR	✗	✗	Multisignature

[1] Not Turing-Complete [2] PBFT integrated with CoSi group signatures [3] For each shard [4] With 25 shards can achieve throughput of 13,000 tps
[5] All blockchains support the normal traditional signature validation.

Table 2.4: Comparison Of Permissionless Blockchain Platforms

2.3 Smart Contracts

In 1990, Nick Szabo first proposed the term smart contract to refer to "a set of promises, specified in digital form, including protocols within which the parties perform on these promises"[59]. Essentially, a smart contract in a blockchain is a piece of code stored in the chain allowing different peers to do general-purpose distributed computations. With this, it's possible to have different peers interacting with each other, deterministically, by using the defined smart-contract [11].

For example, one of the most cited examples to illustrate this topic in the blockchain is that Bob defines a smart contract in the blockchain. This contract contains the code with three functions (a deposit function, a withdrawal function, and a trade function) that allows anyone to trade with it. Once Bob has deployed the smart contract, he can perform a deposit function to increase the inventory of item B available in the smart contract. The trading function is defined so that for every two items of type A traded by a peer, the smart contract automatically returns a type B item. Therefore, if Alice tries to trade only one type A item, she will not receive a type B item, but if Alice trades the right amount of type A items, the code in the smart contract will automatically return the earned type B item to Alice. As you can see from this example, some validations have prevented Alice from committing fraud and enforcing an agreed law (or code law) between two peers without the need for a trusted intermediary, and we must note that the deposit function and the withdrawal function are tied to Bob's private key.

2.3.1 Smart Contracts in Blockchain and Cryptocurrency Domains

Smart contracts to be used in a blockchain need to be digitally signed programs like any cryptocurrency transaction performed in a blockchain using digital signatures schemes based on asymmetric cryptography (e.g., DSA [34], RSA [51], ECDSA , etc.).

Different blockchains have different consensus plane services and mechanisms with different levels of consistency. However, when a blockchain implements a smart contract, it must allow digital security by decentralizing it into three different phases (deployment, installation and execution). Once deployed, a smart contract cannot be changed during the lifecycle of any transactions executed with this smart contract and, since they are stored in the chain, a unique address is provided where the latter clients can interact by addressing transactions to it. A smart contract is usually defined so that any arbitrary state can be stored and any arbitrary computation can be performed. In a permissionless blockchain in a crypto-currency system, client interaction can lead to changes of state and the exchange of cryptocoins from one smart contract to another or even from one account to another. In some cases, these smart contracts can also be used as the first component for setting up a blockchain. In a permissioned and consortium blockchain, smart contracts are agreed by all participants before they are executed. This means before a transaction with this smart contract is executed, all participants are aware of the contract content. In

many applications, however, this does not mean that is being stablished legal agreement recognized by law, but that code agreed upon in advance is executed. For example, a smart contract can be used to switch from an unpredictable human to algorithms so that we can automate payment methods, event notifications [44] or even the exchange of pre-agreed transactions (like our first example of this section) [12, 44].

2.3.2 Smart Contracts and Programming Support

Smart contracts can be defined in many different programming languages, some of these languages are well known (like Java, Golang, Javascript, etc.), but others like Solidity [20] were created simply to solve this domain problem, and there are even smart contract specific languages that are not Turing-Complete like Bitcoin [45] smart contracts. But what all these smart contract programming languages have in common is that they appear as a built-in feature of some blockchains to design custom, sophisticated logic. In the following subsection we will discuss in more detail the types of smart contracts supported by the different blockchain platforms.

What we need to keep in mind is that smart contracts consist of code that must be visible to all participating nodes, and most often they contain valuable assets, which can cause attackers to be tempted to explore existing vulnerabilities such as security holes that are difficult to fix. For example, in June 2016, a reentrancy attack on smart contracts was conducted and called DAO attack [17]. This attack had enormous consequences, such as the draining of 50 million US dollars in ether coins and reduced the trust and value of ether coins.

2.3.3 Smart Contracts in the different blockchains

Ethereum: Ethereum blockchain supports the execution of smart contracts in a Virtual Machine (VM). The Ethereum Virtual Machine (EVM) is the virtual machine supported by Ethereum smart contracts, which is able to execute a Turing-Complete language called EVM bytecode [62]. A smart contract programmer normally does not program directly in EVM bytecode. Instead, Solidity is used as a high-level programming language, as shown in listing 2.1, but it is necessary to compile Solidity to EVM bytecode before execution.

Solidity based smart contracts allow to specify rules for regular expressions, recursions and loops (Turing-Complete). An important concept in the execution of Ethereum smart contracts is the control of the so-called halting problem [35] - the halting problem is a problem of trying to determine whether a program ends or runs forever. To solve this problem, Ethereum smart contracts introduced the concept of gas. For Ethereum, gas is a way to give computing time to a certain smart contract, the caller of that smart contract has to buy gas first, a certain amount of gas is consumed during the execution of each EVM operation, in case of gas shortage the smart contract is stopped, but if there is still gas left, it is returned to the caller. In this way, denial of service attacks can be prevented, as the cost of operating a smart contract is not insignificant.

Listing 2.1: Simple Solidity smart contract [20] that is able to store a unsigned int in the ledger of Ethereum.

```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Listing 2.2: A Bitcoin script that checks for SHA1 hash colisions [61].

```
OP_2DUP OP_EQUAL OP_NOT OP_VERIFY OP_SHA1 OP_SWAP OP_SHA1 OP_EQUAL
```

Bitcoin: Initial Bitcoin infra-structure had no notion of smart contracts, but latter it was introduced Turing-Incomplete scripting language (the Bitcoin Script Language). Despite being Turing-Incomplete, it is possible to create different smart contracts with the Bitcoin scripting language, such as multisignature accounts, lotteries, payment systems, trading systems, freezing funds and many other smart contracts [2].

The scripting language is a list of instructions composed of operands and operations. When an operand is declared, it is placed in a stack, after operators will pop out and put the result in the next stack. A smart contract is considered to be executed successfully if a non-zero value is at the top of the stack at the end, otherwise it is considered failed [61]. Listing 2.2 shows a smart contract (script) that verifies SHA1 hash collisions. The decision not to give the smart contract programmer a Turing-Complete language may help to overcome security problems that can be seen in the smart contract of Ethereum [2].

Tezos: The Tezos blockchain platform supports its own set of rules with minimal network disruption through an on-chain governance model. Tezos smart contracts are supported by a Turing-Complete stack language similar to Ethereum. As we have seen, Ethereum uses a fee model to limit the number of steps a smart contract can take. However, it is possible to perform an attack that exploits the fact that all validators must validate a transaction. For example, a malicious miner could forge a transaction that would run in an infinite loop and pays himself to validate it, meaning that other validators will waste a lot of computer resources trying to validate the transaction [31]. To mitigate this problem, Tezos, in addition to using fees, set a cap on the number of instructions that can be executed with a single transaction, but this limit may increase in the future.

	Turing Complete	Solve Halting Problem	Type	Language
Ethereum	✓	✓	on-chain	Solidity
Bitcoin Script	✗	✗	on-chain	BitCoin Scripting Language
Sawtooth				
TF	✓	✗	installed	GO, Java,...
seth	✓	✓	on-chain	Solidity
Algorand				
ASC1	✗	✓	on-chain	TEAL
ASC2	✓	✓	off-chain	–

Table 2.5: key characteristics of the different smart contracts

Hyperledger Sawtooth: Hyperledger Sawtooth is a blockchain platform that supports two types of smart contracts: installed and on-chain contracts. The installed (or transaction family) smart contracts were created with the aim of limiting the risks associated with programming a smart contract using an Turing-Complete language. This is achieved by limiting the number of operations that a transaction can perform. Hyperledger Sawtooth provides developers with several examples, one of which is the IntegerKey transaction family, which allows three operations to be performed on integers (increment, decrement, set), and another is the Settings transaction family, which allows blockchain settings to be exchanged "on the fly", e.g., the consensus protocol can be swapped when the blockchain is running. Transaction families can be written in a variety of programming languages such as Golang, Java, Javascript, Python, Rust, and so on.

Hyperledger Sawtooth also supports on-chain smart contracts when using Burrow Ethereum. This component makes it easy to execute smart contracts written with Solidity. As we have seen in Ethereum, Solidity is a Turing-Complete language that allows you to program any business logic.

Algorand: Algorand Smart-Contracts are divided into two parts, the first of which have already been released: the Layer 1 on-chain smart contracts (ASC1) and in the future the Layer 2 smart contracts will be developed. An ASC1 can offer security, efficiency, and atomicity as a single-payment transaction [43]. These smart contracts are known to be part of the protocol and are able to communicate with Layer 1 features such as atomic transfers, which provides a way to guarantee that all transactions in the smart contract are executed or none of them is executed. ASC1 can be written in Transaction Execution Approval Language (TEAL), which is an assembly like Turing-Incomplete language that is processed with a stack- machine [1], but is usually written using Python with the PyTeal library. Layer 1 smart contracts are good for everyday needs, but Algorand has layer 2 off-chain smart contracts when there is a need for more complex contracts.

2.3.4 Summary

In Table 2.5, we present some of the characteristics of blockchain-enabled smart contracts. Smart contracts for blockchains have received a lot of attention in the research community. Many issues have arisen, such as how to prevent smart contracts from running forever, damage in executions with exacerbated resources, concerns about language support and expressiveness, among other issues. Different blockchains have developed different smart contracts with different features to address such concerns in different ways.

2.4 Critical Analysis

We truly believe that blockchains need to be revised to better improve resilience and decentralization. Currently, blockchains rely on centralized authorities to sign participants' public keys. Only then can nodes participate in the various phases of a blockchain protocol. Relying on a centralized authority can lead to very lucrative attacks, as it is only necessary to simply compromise a node to gain an advantage over the system. As stated initially (chapter 1) we will design a cryptographic co-signature service with the following requirements: a) Auditability; b) Resiliency; c) Fairness and Dynamic Reconfigurability; d) Pluggability and Factoring Modularity; e) Portability; f) Decentralization.

Blockchains using digital signatures in a single root of trust fail to address a decentralized trust model. The ability to easily apply our portable solution to any blockchain typology, neutral to other specific service planes, can provide tremendous benefits, for near-zero cost implementation. Portability and trust decentralization have already been addressed by other authors [38]. In contrast, the work presented by [37, 39, 56] relies on heavy changes to the blockchain consensus protocol to use group signatures, which leads to poor portability as no modular solutions are used. Another interesting property that we did not find in related work is pluggability and factoring modularity. The ability to quickly instantiate new cryptographic primitives or plug a new, faster primitive into a crypto-provider at runtime can provide a great qualitative solution. Solutions such as [37–39] provide cosigning auditability over a tamper-resistant log. The use of BFT THS cosigning models are an important feature already addressed in [38, 56]. We will pursue the mechanisms of such solutions as baseline mechanisms to create more fair and dynamically reconfigurable cooperative co-signing groups in a more generic way. We must mention that the approach in [38] assumes a centralized trust model, even though the solution can be adopted for permissionless or permissioned models. In [56], the authors use decentralized BFT THS, but specifically targeting the HLF permissioned blockchain transaction processing plane. As a novel dimension, our solution is targeted to be agnostic, not depending on the base service planes and mechanisms of specific blockchains - using two different blockchains (HLF and Algorand as our neutrality proof-of-concept).

Next chapter we will present our elaboration approach for an agnostic, scalable, and highly configurable co-signing component.

Chapter 3

System Model and Architecture

In this chapter, we present our system model and architecture for an agnostic, scalable, and configurable component that can be integrated with different blockchain typologies. The solution aims to address the single point of failure problem introduced in blockchains by conventional signatures, where PKIs and CAs are part of this centralized trust model.

We first describe an application scenario, building the final solution for the application scenario step by step. Next, we present our system goals and a layered view of our system model, then we discuss our reference architecture and each component that is part of our architecture.

3.1 Application Scenario

To better illustrate our system model, we can imagine a simple application scenario inspired by [36]. In today's cities, towns and villages, more and more people are looking for sustainable energy sources and ways to save money. Currently, solar energy plants and eolic turbines are the only two viable options of sustainable energy sources that can be generated on premises to reduce a household's overall energy consumption from the grid, thereby reducing electricity bills. Solar and Eolic energy, as we know, are intermittent energy sources, as the period of time we can generate energy with either of these energy sources is limited and varies depending on the length of the day and weather conditions. One way to solve this problem is to store the energy for pos-consumption, however, compared to simply storing and using fossil fuels from the grid, batteries have a high green premium that hardly any household is willing to pay. A fictitious company called GreenTrader knew of this marked gap and decided to create a disruptive innovation that would allow household energy producers to sell their excess energy production to their community at a lower price. In order to sell the excess energy produced, one party

can issue a transfer transaction to another party, who receives the right to consume that energy during a specific time (e.g., household A sells 20% of the excess energy produced from 10:00 a.m. to 11:00 a.m.).

To illustrate the challenges for such a solution, we first present two strawman solution implementations for this application scenario under a Byzantine environment. Based on the lessons learned, we then summarize the system goals and a representation of this application scenario, shown in Figure 3.1.

3.1.1 Strawman I: The Unique Central Database

The first *strawman* assumes that the data involved in selling and buying electricity is stored in a single trusted entity. The obvious way to store an application's data is to use a single server instance with a database engine installed. Here, all operations to sell electricity to a neighbor would be performed by a user application and sent to the single database instance.

The *strawman I* architecture is completely centralized, and in order to implement such solutions, all participants in the buying and selling of electricity must trust each other or assign a trusted party to deploy and manage the database instance. This means that if this database is compromised, the application built on top of it will also be compromised.

3.1.2 Strawman II: The Apparently Decentralized Blockchain

The obvious way to decentralize the *strawman I* architecture and provide some level of auditability over data changes is to replicate and provide a tamper-proof log. To replicate their system, GreenTrader decided to enter the blockchain world. To do this, GreenTrader deployed a blockchain node in each household and energy provider that adhered to their service to decentralize their legacy database and to improve auditability of the system.

The *strawman II* solution seemingly brings full decentralization to the application, as no two households need to trust each other, nor do they need to trust an energy provider, however digital signatures are also centralized and a key part to the *strawman II* . GreenTrader did not realize that when they deployed the blockchain platform, they also needed to deploy a PKI to certify the clients and nodes. The PKI, as we already know, is centralized and a valuable target for attackers. When a node's private key is compromised, it is necessary to revoke the byzantine key. The window of opportunity between discovering that a private key has been compromised and revoking the key can give attackers an advantage, and that is not the worst problem. What happens if the PKI's private key is compromised? In this case, certificates can be issued on the names of individual nodes to mask the true identity and attack the system.

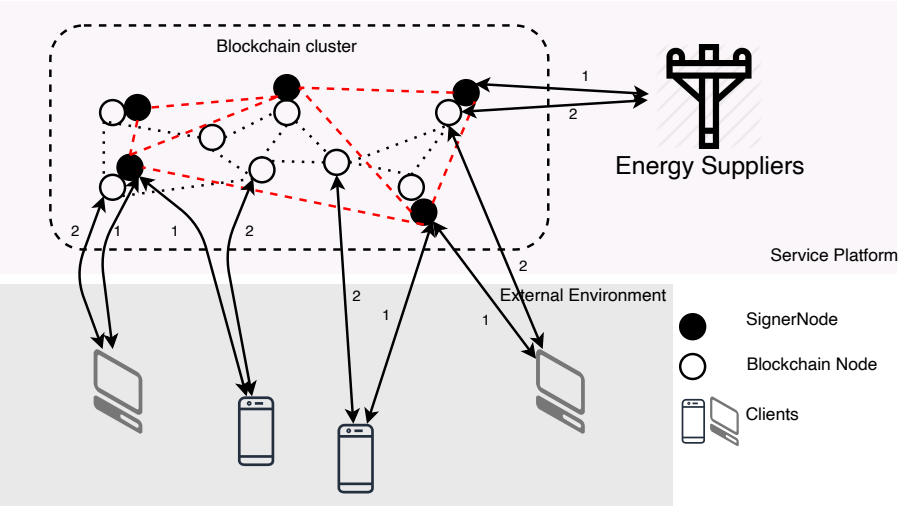


Figure 3.1: Application Scenario illustration

3.1.3 Our Solution to The Application Scenario

To address all the issues in strawman I and II, we propose a modular layer implemented agnostically of the used blockchain that eliminates the centralization problems and the single point of failure present in traditional signatures. The application scenario for our proposal can be seen in figure 3.1.

In this architecture, household clients are represented by a signer node, and a blockchain node, and energy providers are represented by an electricity pole. To sell and buy electricity, an external application represented by a computer and a mobile phone communicates with GreenTrader services in two steps. First, before a transaction (e.g., selling energy) is sent to the blockchain node, each client transaction must be co-signed to ensure that at least t nodes agree and witness the transaction. To do this, clients contact a signer node (1) to obtain a co-signature of the transaction, which is created by a group of witnesses. Signer node's can represent neighbors who will witness the transaction and endorse it after validation. Only then can clients submit the transaction (2) along with the co-signature to the blockchain node to be processed and accepted by all peers after verifying the correctness of the transaction and group signature.

The actual interaction is somewhat more complex than shown in figure 3.1. What needs to be retained is that transactions must be co-signed to remove the conventional PKI, CA, or even the ad hoc certification found in open memberships blockchains.

3.2 System Goals

Our solution sets out to address the following primary system goals.

- **Fairness and Dynamic Reconfigurability**: Fair and dynamically reconfigurable cooperative signing trustee groups with self-sovereign identities, reconfigurable via possible off-chain or sharded-chain membership management.

- **Resiliency**: Support to resilient Byzantine fault-tolerant group witnessed signatures to protect against attacks on the conventional signature. All transactions must be co-signed by a group of witnesses before being processed by a blockchain node.
- **Auditability**: Auditable under verifiable immutable group-signatures, recorded in a tamper-resistant log. All transactions are verifiable and recorded in a tamper-resistant log with the associated co-signature.
- **Decentralization**: Compatibility with a fully decentralized model, not compromising the inherent trust-decentralization requirements.
- **Portability**: Portable and agnostic solution, designed as a reusable service plane independent from other Blockchain-based service planes.
- **Pluggability and Factoring Modularity**: Reusable and pluggable modular cryptographic provisioning, adopting a factory-object oriented design allowing the dynamic addition of different BFT threshold digital signatures' algorithms.

3.3 System Model

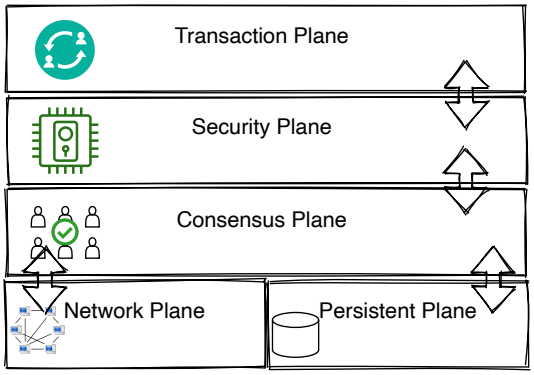


Figure 3.2: Node Layered Model

To better understand our System Model, let's first step back and understand the primary layers our solution must have. Simply put, our system model consists of a set of nodes that work together in a P2P network. Each of these nodes implements a software stack with different layers of services. To represent a node, we first use a simple stacked model consisting of 5 important layers, as shown in Figure 3.2. As in all blockchains, we find a transaction layer, a security layer, a consensus layer, a network layer, and a persistence layer in our system model. If all these components work together, a secure blockchain can emerge.

In Figure 3.3, we introduce several components in our layered model that help us describe the critical roles each plane plays to build a fully functional node. In the following subsection, we briefly describe the purpose of each plane.

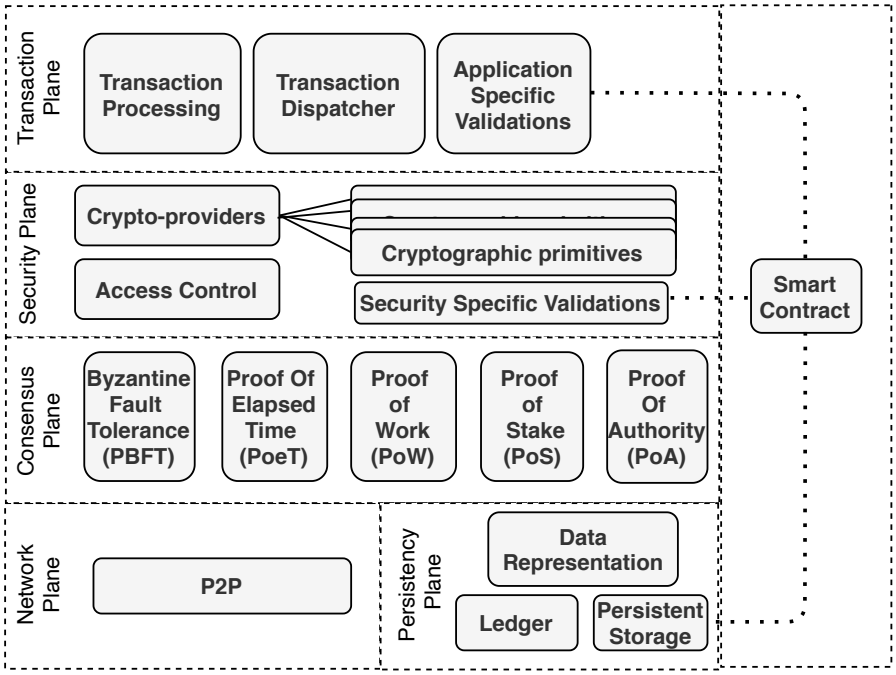


Figure 3.3: Node Layered Model With Componenets

3.3.1 Planes

a. The **Transaction plane** is responsible for transaction processing and transaction dispatching. When a client submits a transaction to a blockchain node, this is the first component that will make a preparatory processing, validations, and then dispatches the transaction to the security plane. The smart contract in this plane plays an important role of providing some of the validations. For example, in case of our application scenario, the smart contract may have a condition that limits the amount of removed inventory items in a single transaction. These validations are called Application Specific Validations.

b. The **Security plane** is responsible for the access control and the use of crypto-providers to sign and validate signatures. The crypto-providers contain different cryptographic primitives, such as THS and conventional signatures schemes. They ensures integrity, non-repudiation, and authenticity of any transaction. Some conditions for choosing which primitives to use are also expressed from roles provided from the extended smart contract expressiveness, to these validations, we call Security Specific Validation.

c. The **Consensus plane** is responsible for ensuring a correct SMR, therefore, protocols such as PBFT, PoET, PoW, PoS guarantee that each transaction is applied in the same way to the different participant nodes. by doing so, the ledger is considered distributed between the nodes. For example, if the order of transactions A, B, C is being decided, this plane must ensure, using other available service planes, that A,

B, C are applied to the different nodes in the same order.

d. The **Network plane** is responsible for exchanging messages between the nodes during the different protocol phases, which means that when data arrives at this plane, it must propagate to reach all nodes in the blockchain using a P2P protocol. For example, the consensus plane uses this layer to propagate the messages necessary to reach a consensus between the different participants, and the security layer, if needed, can also take advantage to propagate signatures related data.

e. The **Persistent plane** is responsible for storing different types of persistent data and the respective representation in a node. For example, if a client deploys a smart contract, it must be stored so that later a client can address transactions to it, and the transaction ledger must persist in every participant node. Another example is the signature state persistency while there aren't enough signature shares to build the signature.

3.4 Threat Model

We assume that adversary nodes have limited computational power, this means secure hash functions, Diffie-Hellman, and signature schemes can hold their assumptions even under attack. We assume that each correct participant verifies the different signatures present in the transaction and accepts only those that were correctly signed.

The transaction witnesses consist of n signer nodes, t of which may fail or behave maliciously by producing incorrect protocol messages or simply going offline. The number t is determined by the previously defined smart contract, we require n >= t. We do not impose any further restriction on t.

We can assume that clients submitting transactions are trustworthy and adhere to the defined protocol and do not behave maliciously, since it is the client the first principle who wishes to submit his transaction correctly without any obstacles. We assume that attacks on the availability of the node's resources, such as Denial of Service (DoS) and Smart Contract DoS, and attacks on peer-to-peer communication, are outside the scope of our system model.

3.5 Mapping to our Architecture

In our system model, we have presented our solution as a single component, but we cannot develop an agnostic solution to decentralize conventional signatures by utterly changing a blockchain node to integrate the group signature and verification process. Instead, our solution is divided into two main components. A signer node, which is responsible for signing and validating the transaction, and a blockchain node, which is responsible for processing and validating the transactions, as shown in figure 3.4. This allows us to achieve our portability system goal with a high degree of agnosticism, enabling the

adaptation of our prototype by any blockchain with any consensus protocol or access control policy.

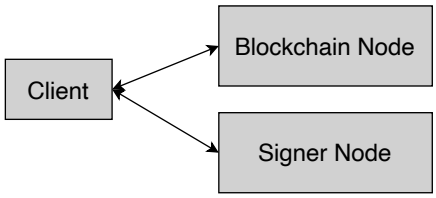


Figure 3.4: Component Division: Signer Node + Blockchain Node

3.6 Interactions

The interaction in our system occurs between three entities, a client that wants to securely submit transactions to a blockchain platform, a signer node that helps the client achieve its goal, and a blockchain node that receives, verifies, and applies the transactions sent by the client. Our system provides two types of interaction, one for a closed-membership blockchain and another for an open-membership blockchain.

Permissioned Model We start with a permissioned model because it is the simplest and contains some common flow fragments with the permissionless model. For our permissioned model, illustrated in Figure 3.5, we consider blockchain with a closed-membership, this means that an administrator has previously configured the nodes with the private key shares and the group public key, and at this point a client can start sending transactions to the nodes knowing that the keys have been previously agreed upon by each signer node.

The permissioned model can be seen on Figure 3.5.

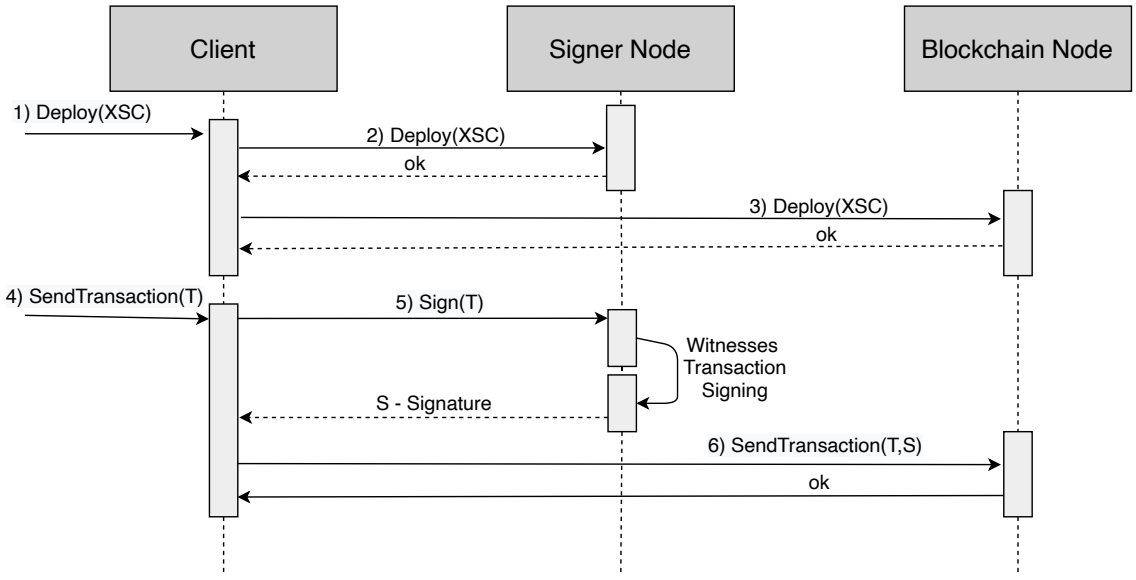


Figure 3.5: Permissioned interaction model

Following on Figure 3.5, the interaction flow of the permissioned model is as follows:

1. Before a client starts sending requests to the blockchain, it must first provide a smart contract, which is responsible for processing transactions and providing the properties used to sign a transaction. Depending on the blockchain platform, this smart contract can be accessed in three different ways: a) a client can piggyback the smart contract along with the transaction and then send it to the signer node; b) the smart contracts can be provided by an administrator before the blockchain is launched; c) the REST API provides a special operation to deploy smart contracts in the blockchain. In Figure 3.5, we have a client that explicitly deploying an extended smart contract to both the signer node and the blockchain. It is important that the extended smart contract is deployed in both nodes.

    2. The signer node must have access to the smart contract to execute the client transaction and extract the crypto-provider configurations to dynamically choose how to sign a transaction.

    3. The blockchain node must have access to the smart contract in order to execute the contract logic and process the transactions as required by the smart contract code.

4. Once the smart contract is deployed, a client can start sending transactions to the blockchain.

    5. Before a transaction is sent to the blockchain, it must be co-signed by the different signer node witnesses. To do this, a client sends a transaction to a signer node, which produces a co-signature share using the extended smart contract and group primitives available in our crypto-provider. Then, if successful, the signer node asks other witnesses to co-sign the transaction, combining all co-signature shares into one and returning the final co-signature to the client.

    6. Once the transaction is signed, the client can now send the transaction plus the group co-signature to the blockchain. When the transaction reaches the blockchain node, some validations are performed, including signature verification, to guarantee that the transaction was successfully co-signed. After these validations, the transaction can be applied to the ledger, including the co-signature envelope, so that the co-signature can then be audited.

Permissionless Model Not every blockchain has a closed-membership and the configurations can be made before execution. To integrate our prototype into a blockchain platform with open-membership, we introduce the permissionless model. This model does not care about where a particular (signer or blockchain) node is located, but simply assumes that configuration can be performed by a client interested in ensuring that its transaction runs smoothly and securely.

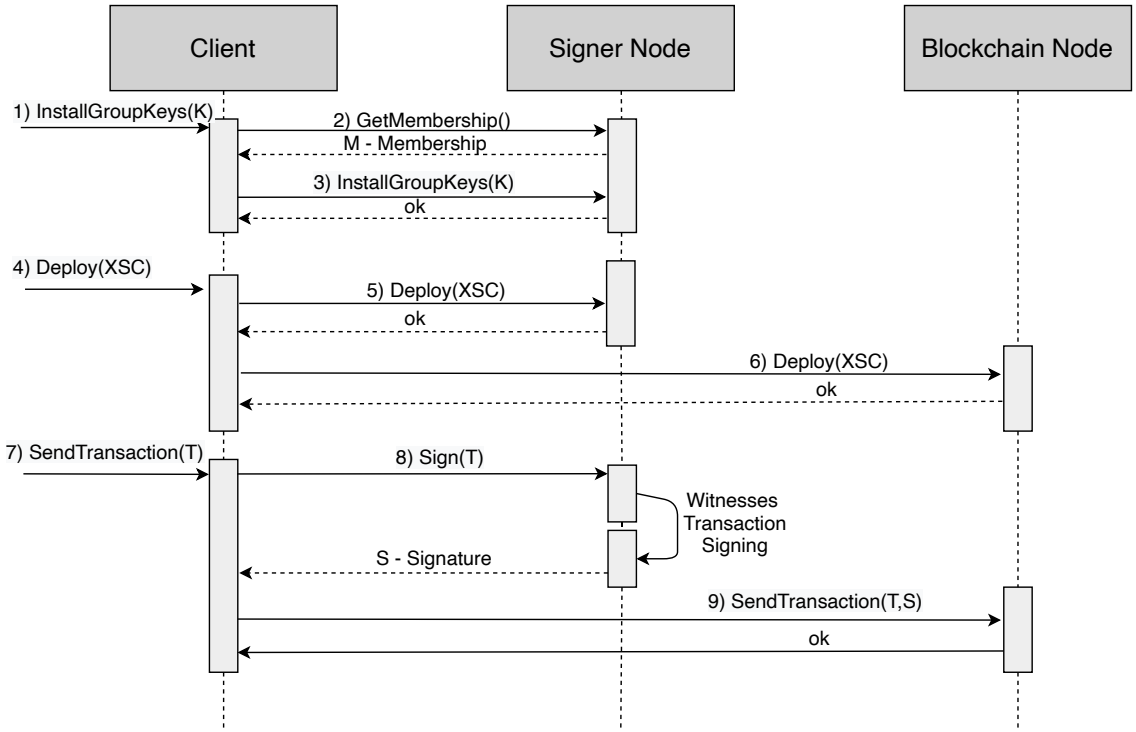


Figure 3.6: Permissionless interaction model

The permissionless model can be seen on Figure 3.6.
Following on Figure 3.6, the interaction flow of the permissionless model is as follows:

1–3. Before a client sends transactions and installs the smart contract, it must first select a group of signer nodes that will validate its transactions. To do this, the client first contacts the signer node to obtain a subset of the membership, and then uses this subset to select some in different ways, such as selecting the most geographically distant witnesses or the most regionally distant, but importantly, the selection must be random to reduce the possibility of finding Byzantine signer nodes and to increase the probability that different nodes will always be selected to sign the transactions (fairness). The selection and generation of key shares can be done offline without affecting performance, then the client installs all required keys in each selected signer node. During this installation process, it can choose how long the keys can be used or if it is a one-time use to increase security.

4–6. Similar to the permissioned model, a smart contract must be deployed by the client in both the blockchain and the signer node to subsequently enable transaction validation and parameterization selection for our crypto provider.

7–9. The difference between the permissioned model and this model is that in this model the client cannot send the transaction to any node in the membership of the signer nodes. The client must send its transaction to one of the signer nodes that it has previously installed the keys. After the signer node receives the signing request, it

verifies that it has the appropriate key for the transaction and signs it according to the logic specified in the extended smart contract, and it asks other witnesses that are in the same witness group to sign the transaction. Then, the client sends the transaction to the blockchain along with the co-signature produced by the transaction witnesses.

3.7 Reference Architecture

Our proposed solution allows clients to submit co-signed transactions witnessed by a set of signer nodes to improve decentralization, auditability, and non-repudiation by each of the involved parties via a generic blockchain. To achieve this, we divided our solution into two main components: i) a signer node, which is responsible for co-signing and validating client transactions; ii) and a validator, which is responsible for processing and validating transactions after they have been co-signed by a set of signer nodes. These two main components, shown in Figure 3.7, work together with other equivalent components in a distributed peer-to-peer network to build a Byzantine, fault-tolerant and decentralized architecture.

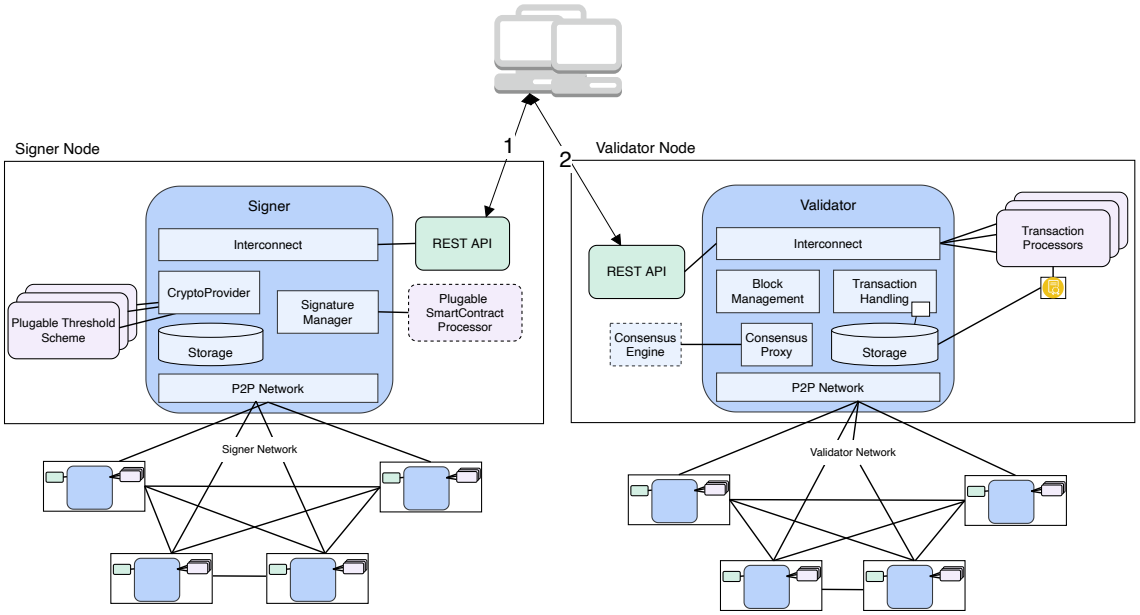


Figure 3.7: Extended Architecture

Signer Node: Our signer node is designed as an agnostic and portable component to be compatible with any blockchain and extensible for future upgrades, such as integrating new cryptographic primitives, modifying the running core protocol, and other extensibility features. The signer node is principally mapped to three System Model layers, the security plane, the transaction plane and a network plane to allow signer nodes to collaboratively build a group co-signature. This main component is subdivided into other smaller components, such as:

- **Signer Node API:** enables communication between the client and the signer node. For example, request a transaction co-signature, request a signature validation, etc.
- **Cryptoprovider:** allows the creation of signatures with different signature schemes. It is designed to primarily address our system goal of *Pluggability and Factoring Modularity* and *Resiliency*.
- **Smart Contract Processor:** allow us to validate a transaction and dynamically choose how to sign a particular transaction.
- **Signature Manager:** is one of the most important components, as it is responsible for joining all the components we have talked about up to this point and executing a well-defined protocol. It is designed to primarily respect our *Fairness and Dynamic Reconfigurability* System Goal.
- **P2P Network:** allows different signer nodes to exchange messages to collaborate in a defined protocol to build a co-signature over a client transaction.

To better understand the role of each component, let us briefly describe a transaction flow. A client starts by sending a request to the Signer Node API with the transaction to be signed. Next, the Signer Node API forwards the transaction using the interconnect to the signature manager component, which selects the correct smart contract to execute that was previously deployed. When the smart contract is executed, the transaction is validated, resulting in different properties that help correctly parameterize the crypto provider to sign that transaction. After the correct parameterization of the crypto provider, the transaction is processed by one of the protocols which will be discussed later. There is one protocol for permissionless blockchains and another for permissioned blockchains. The result of running one of these protocols is a correct co-signature over a transaction that is sent to the client. Since some cryptographic primitives need to communicate with other signer nodes to cooperate in signature generation, it is necessary to use a P2P network to allow different signer nodes to send their signature shares.

Validator Node: The validator node aims to represent a generic node found in current blockchain platforms, with different characteristics, for example, we can find a validator node with PBFT consensus protocol with closed membership setting and other with PoS consensus protocol with open membership setting. However, for integration with our signer node, we only need to implement the pluggable smart contract processor to run an extended smart contract, since our Signer Node is portable and agnostic.

The transaction flow in the validator node depends on the blockchain platform used, but some general things happen: (i) the client submits a co-signed transaction to the node; (ii) the node submits a co-signed transaction to the consensus protocol; (iii) the node executes the smart contract and verifies that the signature built by the signer node is correct; (iv) the transaction is attached to the ledger along with the co-signature to ensure a high level of auditability.

3.8 Software Architecture Components

In this section, we address the architectural components of our signer node that enable us to achieve our system goals and requirements. We do not address blockchain-specific components, as our solution is built agnostically, regardless of which blockchain platform is used.

3.8.1 Signer Node API

A client to invoke operations over our signer node must have access to an API that promises different operations to achieve the defined protocols. The signer node API is the entry-point that allows communication with our signer node, in order to execute well-defined protocols (permissioned, permissionless) and to invoke other functionalities that help the client make decisions during the different protocols. Let us now define the methods that can be expected in our API.

- *installSmartContract(smartcontract) boolean*: In blockchains where it is necessary to install smart contracts on the fly, this operation installs an extended smart contract in the signer node, which is then executed when a transaction is signed.
- *signTransaction(smartcontracAddress,uuid,data) SignatureSpecs*: This operation sends a request to sign the transaction data. Given the smart contract address, the UUID of a previously defined group (for a permissionless setting), and the data to be signed, a signature specification is returned containing the group signature produced by signer nodes and the scheme used to sign the data.
- *getMembership() []Nodes*: In a permissionless setting it is necessary to obtain the membership of signer nodes in order for a client to select a set of signers and deploy the keys. This operation supports retrieving this membership, and returns a subset of the membership.
- *installKeys(uuid,ttl,isOneTimeKey,keys) boolean*: This operation allows the installation of the public key and private key share to a particular signer node, given the UUID of the group, TTL to limit the usage time of a certain private key, a boolean to specify whether the key should be used only once, and the key material. This operation may need to be performed once for each signer node belonging to the group specified by the UUID.

3.8.2 CryptoProvider

Our crypto provider is a key component to co-sign transactions and verify that signatures are formed correctly. With our system goals in mind, we set out to decentralize signatures and provide a modular crypto provider that can integrate new primitives in any language at runtime. Threshold signatures can help us achieve decentralization by eliminating the

single point of failure of conventional signatures. Conventional signatures require each peer to sign a transaction individually, which, if we think of attacks on a CA, a PKI, or even ad hoc certification, can lead to Byzantine blocks and transactions. To accomplish this modular crypto provider, we first need to define an API and some algorithms to be supported by the cryptographic primitives we want to integrate into our crypto provider. We then show how a cryptographic primitive communicates with the crypto provider client by exchanging messages.

Let us now define the API that a primitive must respect to be integrated with our crypto-provider:

- **Gen(n,t) (GroupPublicKey,[]PrivateKeyShare)**: Given the number of participants n and a minimum number of correct signatures t, generates a group public key and a set of private key shares. The group public key is used to verify correct signature formation and the private key share is used for signing. In the method stub only n and the t can be parameterized, other parameterizations are done when instantiating the scheme context (e.g., RSA mod size, hash function, etc.).

- **Sign(b []byte, k PrivateKey) (s []byte, e error)**: Specify a message digest (in bytes) b and a private key share k, and generate a signature share s, over b if it is possible to sign, or an error e if something went wrong during the signing phase.

- **Aggregate(s [][]byte, d []byte, k PublicKey, t int, n int) (sig []byte, err error)**: Given a set of signature shares s, digest d, group public key k, minimum number of correct signatures t, and size of group n, combine each signature share in s into a group signature sig. If there are not enough correct signature shares, an error is issued.

- **Verify(s []byte, d []byte, k PublicKey) err error**: Given a signature s, a digest d, and a public key k, verify the signature s over d, and generate an error if the signature is malformed.

Algorithms for non-resilient threshold schemes: In order to provide the robustness guarantees for the different threshold schemes, we need to build algorithms into our crypto-provider that are able to select correct signatures shares to build the final signature. Recall that a robust (t-n)-threshold scheme is one that can generate the final signature in the presence of Byzantine signature shares if we have at least t correct signature shares. In other words, we can tolerate $n - t$ incorrect signature shares, that is, $n - t$ signer nodes can yield invalid signature shares without affecting the liveness properties of our system.

To help us achieve robustness, we will use two algorithms (1, 2) proposed by [56]. Algorithm 1 and 2 are respectively classified into optimistic, meaning that it assumes that it will be easy to find a set of correct signature shares, and pessimistic, meaning that it assumes that more incorrect signatures will be obtained than correct signature shares.

Both algorithms use the framework API described earlier, with *aggregatePessimistic* and *aggregateOptimistic* mapped to the Aggregate methods in our API. We also defined two auxiliary methods: (i) *buildSignature/3* is given a valid signature shares, a digest, and a public key, which it uses to generate a correct aggregate signature from the valid shares or an error; (ii) *generateNextCombination/2* is given a set of signatures greater than or equal to t, a number with the size of the permutation , and the permutation ID, thus generating a deterministic static permutation with t signature shares; (iii) *verifySignatureShare/1* given a signature share returns error if the signature share is not built properly.

The pessimistic algorithm, as the name implies, assumes that some of the signature shares will be corrupted. Therefore, it starts by validating each signature share until it finds t correct signature shares. Then it reconstructs the final group signature. We see that the complexity is $\mathcal{O}(t)$ in the best-case because the algorithm can find t correct signatures after the t initial validations, and $\mathcal{O}(n)$ in the worst case because it is possible that the final signature share is the t correct one or there are not enough correct signature shares. It is easy to see that in an environment with many Byzantine signer nodes, this algorithm behaves with linear complexity and better than the optimistic algorithm we will see next.

Algorithm 1: Aggregate Pessimistic

```
// s is the set of signature shares to be aggregated
// d is the digest of the message
// k is the group public key
// t and n are respectively the lower threshold of current shares and group size
function aggregatePessimistic(s, d, k, t, n)
    valid ⟵ []
    for sig ⟵ s do
        if verifySignatureShare(sig) != error then
            valid ⟵ valid ∪ sig
        if len(valid) ≥ t then
            return buildSignature(valid,d,k) //returns error if the signature is not valid
    return error
```

Validating each signature share is very computationally expensive, so the optimistic algorithm takes a different approach. If we create a subset with t signatures and reconstruct the group signature with success, we can assume that the shares used are all valid. In case of unsuccessful reconstruction, we go back and choose a new subset with t signatures. It is easy to see that in the best-case scenario our complexity is linear ($\mathcal{O}(1)$) because we can obtain all correct signature shares in our first combination, but in the worst case the complexity is $\mathcal{O}(n!/(t!(n-k)))$. For example, if we have a group of 15 signer nodes and a threshold of 8, in the worst case we need to reconstruct 6435 incorrect group signatures until we can obtain 1 correct group signature. Compared to the pessimistic algorithm, the optimistic one behaves poorly in the presence of many Byzantine signer nodes.

Algorithm 2: Aggregate Optimistic

```
// s is the set of signature shares to be aggregated
// d is the digest of the message
// k is the group public key
// t and n are respectively the lower threshold of current shares and group size
function aggregateOptimistic(s, d, k, t, n)
    comb ⟵ generateNextCombination(s,t)
    repeat
        sig ⟵ buildSignature(comb,d,k)
        if sig ! = error then
            return sig
        comb ⟵ generateNextCombination(s,t)
    until len(comb) == 0
    return error
```

Primitive to cryptoprovider communication: To ensure the property of pluggability and factory modularity, our cryptoprovider must be structurally divided into two prominent components that communicate with each other by exchanging messages. The components are the cryptographic client, which is responsible for invoking the various primitives, and the handler servers, which contains various primitives bundled together like a Cryptographic Service Provider in Java Cryptography Extension (JCE).

To achieve such communication, we opt for a communication pattern known as Router-Dealer, where the router keeps track of a connection list for the various handlers that have previously connected to the router. This means that a router can send messages to n dealers but the dealers can only send messages to one router. This pattern is particularly useful because our cryptographic client can take on the router role to invoke the various handler servers that take on the dealers' role.

Figure 3.8 shows a cryptographic client (router) talking to three different handler servers. A handler server is nothing more than an abstraction layer that contains communication protocols to enable communication between the router and the dealer, an API that defines the interface that each primitive must implement in order to integrate with our cryptoprovider, and the primitive that contains the necessary code to generate our signatures. To create a new provider (Handler Server), it is only necessary to follow the framework API that we discussed earlier.

Before we can use our cryptoprovider, each handler server must register each primitive it manages so that the cryptographic client has knowledge of the primitives that can be used and where to find them. After registering the primitive and invoking the smart contract, the client is ready to send a transaction to be signed, along with the specification for such a signature, so that any parameter used to sign a transaction can subsequently be seen in the blockchain node.

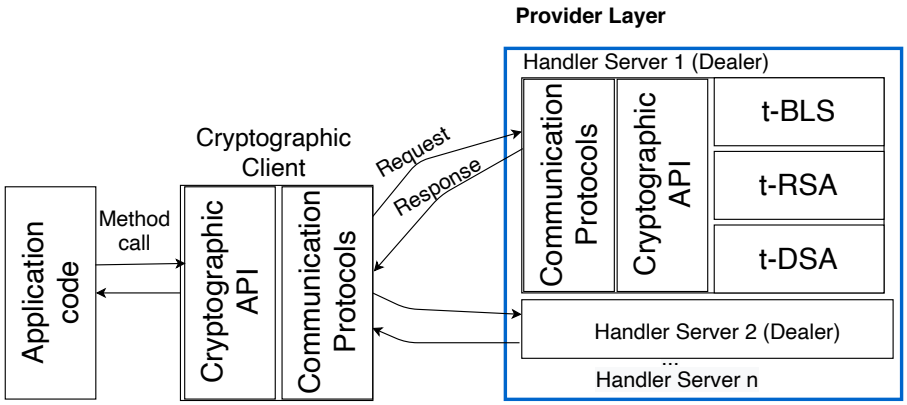


Figure 3.8: Cryptoprovider Architecture

Figure 3.9 illustrates the co-signature of a client transaction. A client transaction is considered a black box that can only be interpreted by the smartcontract engine. The co-signature header consists of the protocol version, the group size n, the threshold t, the type of signature (conventional RSA, threshold RSA, etc.), and the signer's public key, which in our case will be the group's public key.

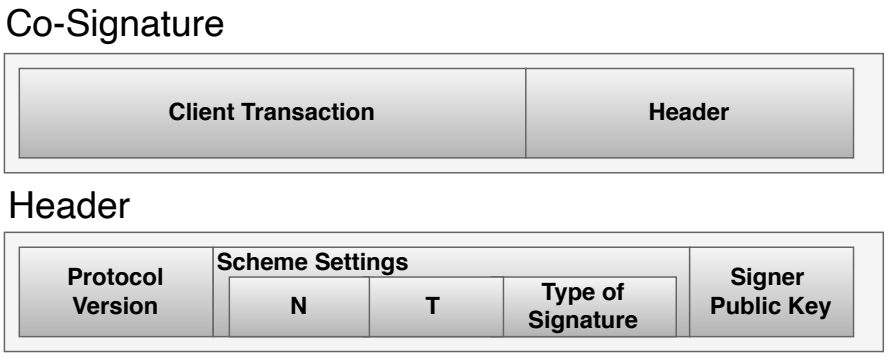


Figure 3.9: Co-Signature Message Structure

3.8.3 Smart Contract Processor

Our extended smart contract engine is a key component to dynamically choose some parameterization used to sign a particular transaction with our crypto provider. In order to dynamically determine the parameterization and verify the validity of a particular transaction, a transaction must be executed by an extended smart contract that contains security and application-specific validations defined by the smart contract owner.

To better illustrate what needs to be implemented in an extended smart contract to be executed in our signer node, we first need to specify an extended smart contract in pseudocode that is used as a model to be instantiated by a partial blockchain smart contract. Succinctly, an extended smart contract can be divided into two distinct sections:

i. a section with properties and functions required to fulfill the requirements of a base smart contract present in current blockchains. Typically, the base smart contract contains properties and logic associated with the business requirements;

ii. a section with properties and functions required to extend a smart contract. This includes ways to perform validations on a transaction, determine if it is valid, and define how a transaction should be subsequently signed.

Listing 3.1: PseudoCode

```
contract extended_contract {
    /*Define Structs
    TYPE SECURITY_PROPERTIES = STRUCT {
                                SignatureType STRING,
                                SignatureScheme STRING,
                                N INTEGER,
                                T INTEGER,
                                PubKeyTTL INTEGER,
                                ValidFrom DATE,
                                ExpiresOn DATE,
                                IsValidTrans BOOLEAN
                            }

    /*Define Application State Struct */
    TYPE STATE = STRUCT {
                    ...
                }

    FUNCTION Invoke(msg BYTES, old STATE) STATE{
        See if the invoker can execute the smartcontract
        Parse msg and run application specific validations
        Run application business logic to calculate new State

        return STATE{...}
    }

    FUNCTION Validate(msg BYTES) SECURITY_PROPERTIES{
        Run Security specific validations
        Run Application specific validations
        Calculate Security Properties

        return SECURITY_PROPERTIES{...}
    }
}
```

Listing 3.1 shows an extended smart-contract structure in pseudocode with different required properties and functions. In the listing, the two sections mentioned earlier are physically separated. Section i) is mapped to the type STATE and the Invoke function. Section ii) is mapped to type SECURITY _PROPERTIES and the Validate function. If

a particular blockchain supports smart contracts, then an Invoke function and its corresponding properties are found in these smart contracts, allowing application-specific validations to be performed and business logic code to be executed for a particular application, transforming the old STATE of the blockchain into the new STATE. To execute a smart contract in our signer node solution, it is mandatory to have a function to validate transactions. This validate function takes a transaction and performs security and application specific validations and finally computes SECURITY _PROPERTIES needed to sign a transaction. The SECURITY _PROPERTIES are nothing but parameterization and properties that help in signing a transaction. There are several fields in the SECURITY _PROPERTIES, such as:

- **SignatureType** - Defines the type of signature used, can take the value of *multisignature*, *threshold signature*, *conventional signature*, etc.
- **SignatureScheme** - Defines the scheme used to sign a certain transaction, can take the value of RSA, BLS, DSA, etc.
- **N** - Defines the size of the group of witnesses to sign a certain transaction in case of group signature.
- **T** - Defines the low threshold of signatures needed to form a valid transaction in case of using a group signature.
- **PubKeyTTL** - Defines the maximum lifetime a public key can have under the assumptions of this smart contract. After *x* seconds, a public key should no longer be used for security reasons.
- **ValidFrom** - Defines the point in time when it is valid to use a certain private key to sign a transaction
- **ExpiresOn** - Defines the point in time when a private key expires.
- **IsValidTrans** - Determines whether a particular transaction is valid and can be signed. When the smart contract executes a transaction, it performs application-specific and security-specific validations that determine whether or not a particular transaction is valid.

Client to Engine communication: Like all our other components that are blockchain dependent, the smart contracts will need to be modified and extended to a specific blockchain implementation to maintain the high level of agnosticism, reconfigurability, and portability for our solution.

To achieve the system goals, we decided to separate the smart contract processor into two parts: a client and a smart contract engine. The client interacts with the smart-contract engine via message passing. For this purpose, we could have simply used a client-server communication pattern, since all extended smart contracts could run in the

same engine. However, for better flexibility, we choose to use the same communication pattern that we integrated in the crypto-provider because it allows us to run multiple smart contract engines simultaneously.

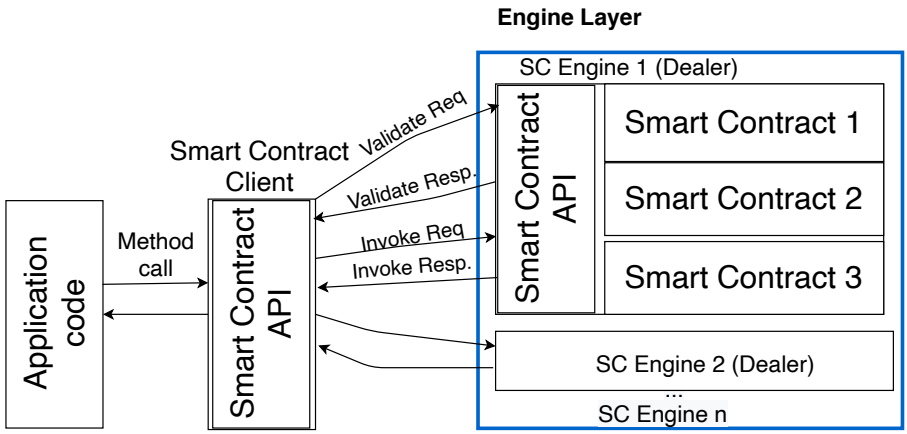


Figure 3.10: Smart Contract component communication.

Figure 3.10 shows a smart contract client talking to an extended smart contract processor engine. An extended smart contract engine contains all the code needed to interpret/compile/execute an extended smart contract in a given language. To create a new extended smart contract processor engine, it is necessary to implement a protocol that contains the following messages:

a. **Register Message**: A register message must be sent from the engine to the client to inform the client of the existence of the engine.

b. **Validate Message**: When a client wants to validate a particular transaction, it sends a validation message and receives the security properties in response.

c. **Execute Message**: When a client wants to execute the normal flow of a smart contract, it sends an execute message and receives in response whether the smart contract was executed correctly.

3.8.4 Signature Manager

One of the most important components in our prototype is the Signature Manager, because it is responsible for gluing together all the components we have discussed up to this point, such as Smart Contract Processor, Cryptoprovider, REST API and the Storage. The Signature Manager achieves this unification by running one of the implemented protocols, one for a permissioned blockchain and another for a permissionless blockchain. These protocols have the ability to talk to the different components.

Permissioned Protocol: The simplest protocol we have in our solution is the permissioned protocol. It is designed for a closed membership blockchain, where the administrator configures the different peers before starting the blockchain or runs an update command for the existing nodes. This administrative configuration makes it easy to create groups and distribute shares because apart from having everything configured, the roles for each peer are well defined which makes it easy to create a group that witnesses incoming transactions, we just need to select n signers from that membership that belong to the same witness group.

To better explain the permissioned protocol, the Algorithm 3 represents the respective pseudocode of the protocol based on events.

The permissioned protocol or closed membership signature protocol has defined in the interface an asynchronous request called *Sign* that is invoked by a client communicating with the REST API, and an indication that allows the protocol to return the final aggregated group signature to the client that requested it.

To temporary store the transactions that are being processed we have two map data structures in the protocol state, the *signState* and the *signRequest*. The *signState* contains the signature shares already sent by other witnesses, and the *signRequest* contains important information about the request to be fulfilled, such as the scheme to be used, the minimum threshold of signatures required, the size of the witness group, the data to be signed, and the UUID of this transaction.

Upon a signing request from the client, the signature manager starts executing a specific smart contract, provided by the client, against the data to be signed. The smart contract engine executes a smart contract and returns a tuple with useful properties. For simplicity, we have not specified in the protocol algorithm all the properties returned by the smart contract engine after the execution of a smart contract. Based on the properties returned by the engine, the protocol first cheques whether the transaction is valid. If it is valid, transaction processing can continue and the request is stored in the signRequest. After that, it is necessary to broadcast the transaction to all other witnesses so that they can produce a signature share. Then, the protocol selects the correct private key share previously installed by an administrator and uses it to call a sign function in our cryptoprovider module, which produces a signature share that is stored in the *signState* along with other signature shares.

Concurrently, after the *RequestToSign* is triggered, other witnesses will produce their signature share, as we explained earlier. The difference is that after producing the share, the witness broadcast its share to the group as a *SignResponse*. Upon sign responses, a witness collects the signature shares, and eventually, if there are at least t non-byzantine nodes, the signature can be aggregated. To aggregate a signature, a witness searches for the matching public key and asks the cryptoprovider to aggregate the transaction. After the signature is aggregated, if it is valid, it is returned to the client that requested it via the *SignDelivery* indication. If it is not valid and all shares have been delivered, an error message is sent to the client.

Algorithm 3: Closed Membership Sign Protocol

Interface:
Requests:
Sign (*info*,*data***)**
Indications:
SignDelivery (*signInfo*,*signature* **)**

State:
signState //map with a list of shares identified by the request id.
signRequest //map with request information identified by the request id.

Upon Init () do:
signState[] ⟵ ⊥;
signRequest[] ⟵ ⊥;

Upon Sign(*info*,*data***) do:**
(scheme,t,n,valid) ⟵ invokeSmartContract(info.SCAddress,data);
If valid **then**
signRequest[info.uuid] ⟵ (scheme,t,n,data)
Trigger BroadcastP2P(RequestToSign,*info*,*data***)**;
privKey ⟵ getPrivKey(scheme,t,n)
signature ⟵ sign(privKey,scheme,t,n)
signState[info.uuid] ⟵ signState[info.uuid] ∪ {signature}

Upon RequestToSign(*info*,*data***) do:**
(scheme,t,n,valid) ⟵ invokeSmartContract(info.SCAddress,data);
If valid **then**
signRequest[info.uuid] ⟵ (scheme,t,n,data)
signature ⟵ sign(privKey,scheme,t,n)
Trigger BroadcastP2P(SignResponse,*info*,*signature***)**;

Upon SignResponse(*info*, *signature***) do:**
signState[info.uuid] ⟵ signState[info.uuid] ∪ {signature}
r ⟵ signRequest[info.uuid]
If #signState[info.uuid] ≥ r.t **then**
pubkey ⟵ getPubKey(scheme,t,n)
(aggregateSig,valid) ⟵ aggregate(scheme,r)
If valid **then**
Trigger SignDelivery((*pubkey*, *scheme*)**,***aggregateSig***)**;
Else If r.t = r.n **then**
Trigger SignDelivery((*pubkey*, *scheme*)**,***error***)**;

Permissionless Protocol: Removing closed membership assumptions from our permissioned protocol presents several challenges that conflict with the open nature of a permissionless blockchain. To this end, we present several solutions to this protocol and explain why we chose one in favor of the others. To this end, we present several solutions to this protocol and explain why we chose one in favor of the others. We also build step by step the protocol present in the Algorithm 4 and 5.

There are a few questions that need to be answered to create a permissionless protocol:

- How can we select a group of peers to be involved in signing transactions?
- How to distribute/generate the shares?
- How to receive signature shares and aggregate them?

How can we select a group of peers to be involved in signing transactions?

A naïve way to solve this problem would be to select the entire network of peers as a group. However, this solution introduces some scalability issues, as there is an upper bound where the latency to wait for the signature is unacceptable.

A second approach and more realistic for group election would be similar to Byzcoin [37]. With PoW, nodes with computational power can participate by mining participation shares, allowing them to cooperate in the transaction witness in a proof-of-membership mechanism. To revoke shares or accept new participants, this mechanism builds on a sliding window. Shares outside the window expire and become obsolete, and shares inside the window give a signer the opportunity to participate in the signer's group to witness transactions. This solution has the advantage of being fully decentralized and following the nature of the blockchain, however, it is a bit too complex for our problem and introduces too much latency, since in addition to election a group, we also need to elect a leader to coordinate the signing session, and this protocol for choosing a leader must have the property of being provably fair and fully decentralized.

The second approach removes fairness from the solution, as groups would become biased since only peers with higher computational power would be chosen as witnesses. To mitigate this problem, we decided to create a third approach.

The third approach would allow the client that sends the transaction to choose which signer nodes to sign the transaction, since clients want to select the correct nodes to witness their transactions. A client can obtain a subset of the signer nodes from the blockchain at any time. After obtaining a subset, the client can perform group generation in a random manner offline and autonomously without consuming bandwidth. We must note at this point that the client is the interested party in obtaining a correct signer node group, and not doing so in a random manner would harm its transactions due to the possibility of selecting a biased group containing Byzantine nodes.

How to distribute/generate the shares?

Having defined the witness group, we then need to decide how to divide the shares between the different members of the group. This problem does not arise in a permissioned

blockchain, since we can simply set up the shares when we first start the system up. In a permissionless blockchain, we cannot set up the shares in the same way because of the large combinations. For example, take a permissionless blockchain with 10000 signer nodes available, if we want to set up a BLS threshold signature with n out of 5 nodes, we would get $8,325 * 10^{17}$ possible combinations. Of course, there are a lot of other issues, such as churning and who is responsible for setting up everything. For all these reasons, we need to find another way to distribute the shares:

1. The first solution to this problem can be mapped to the possibility of generating groups with PoW by choosing a leader with a provably fair algorithm that would be responsible for generating the shares and distributing them. This solution, as described, ends up centralizing trust in the leader. In this way, we would have to introduce a notion of epochs in which the leader would be elected and would be responsible for that epoch. This solution is slow because we would have to generate different key shares for each transaction and elect new leaders for different epochs.

2. We chose a second approach and let the client distribute the shares, since the clients themselves are the ones most interested in the protocol. As we discussed earlier, a client can generate a group offline and autonomously without consuming any blockchain resources. To do so, the client would only need to obtain a subset of signer nodes, and then it could generate the group. To distribute the shares, the client could take advantage of this group generation to allocate a private key share for each member of the witness group. Later, the client could use a special operation to inform each participant about the groups it has integrated and the corresponding private key shares for signing transactions from that client. We must note here, the more careful the client is, the more groups it can generate for the different transactions to be processed, however, the signer would have knowledge of the (Time To Live) TTL, so the signer can discard stale groups and private key shares.

How to receive signature shares and aggregate them?

In a permissioned blockchain, our signer node can easily broadcast the signature shares generated for each node because the membership will be small, but the same does not happen in a permissionless blockchain. Thus, we intend to solve this problem in one of the following ways:

1. The easiest way to disseminate the signature shares to the interested party is to organize the witnesses into a list. When the first member of the group finishes co-signing a transaction, he will pass the signature share to the next witness. The other witnesses will do the same but with all those previously produced. When the required threshold of signature shares is reached, the signature is produced by that witness. This solution is not fault-tolerant, as a single error in the list can cause the protocol to stop.

2. To improve the robustness of the aggregation, we can organize it in a tree instead of a list. Like this, a failure of a branch does not mean that the protocol stops.

3. Finally, we could set up a smaller broadcast domain containing only the peers that belong to a particular witness group. Here, the peers could broadcast the signature shares and the first one to collect the minimum number of shares would generate the signature. We will pursue this last option since it allows all peers to get all shares, and in case of failure of a leader it is easy to contact other peers to obtain the aggregated signature. However, we take a small performance hit to broadcast the shares to all group members.

We can summarize the chosen approaches as follows: a) the client selects a subset of the membership and generates groups offline; b) the client generates shares offline for the groups and then installs them in each signer node belonging to the same group; c) the signer nodes broadcast the shares between the group, aggregating into the final signature.

Algorithm 4: Open Membership Sign Protocol Part 1

Interface:
Requests:
Sign ($info$**,**$data$**)**
InstallShare ($info$**,**$priv$**,**pub**)**
Indications:
SignDelivery ($signInfo$**,**$signature$ **)**

State:
signState //map with a list of shares identified by the request id.
signRequest //map with request information identified by the request id.
keyShares //map with key shares identified by an unique id

Upon Init () do:
signState[] ⟵ ⊥;
signRequest[] ⟵ ⊥;
keyShares[] ⟵ ⊥;
Setup Periodic Timer Clean (T) //T is the clean period in seconds

Upon InstallShare($info, pub, priv$**) do:**
keyShares[hash(pubKey)] ⟵ (info,pub,priv)
Trigger JoinBroadcastGroup($hash(pub)$**)**

Algorithm 5: Open Membership Sign Protocol Part 2

```
Upon Sign( info,data) do:
  scResult ⟵ invokeSmartContract(info.SCAddress,data);
  (valid,shareKey) ⟵ checkValidityAndGetShareKey(info,scResult);
  If valid then
    (scheme,t,n) ⟵ (cResult.scheme,scResult.t,scResult.n)
    signRequest[info.uuid] ⟵ (scheme,t,n,data,info)
    Trigger BroadcastToGroup(RequestToSign,hash(shareKey.pub),info,data);
    signature ⟵ sign(shareKey.priv,scheme,t,n)
    signState[info.uuid] ⟵ signState[info.uuid] ∪ {signature}

Upon RequestToSign(info,data) do:
  (scheme,t,n,valid) ⟵ invokeSmartContract(info.SCAddress,data);
  (valid,shareKey) ⟵ checkValidityAndGetShareKey(info,scResult);
  If valid then
    (scheme,t,n) ⟵ (cResult.scheme,scResult.t,scResult.n)
    signRequest[info.uuid] ⟵ (scheme,t,n,data)
    signature ⟵ sign(shareKey.priv,scheme,t,n)
    Trigger BroadcastToGroup(SignResponse,hash(shareKey.pub),info,signature);

Upon SignResponse(info, signature) do:
  signState[info.uuid] ⟵ signState[info.uuid] ∪ {signature}
  r ⟵ signRequest[info.uuid]
  If #signState[info.uuid] ≥ r.t then
    shareKey ⟵ keyShare[r.groupId]
    (aggregateSig,valid) ⟵ aggregate(shareKey.pub,scheme,r)
    If valid then
      Trigger SignDelivery((pubkey,scheme),aggregateSig);
    Else If r.t = r.n then
      Trigger SignDelivery((pubkey,scheme),error);

Upon Clean() do:
  Foreach shareEntry ∈ KeyShares do
    toDelete ⟵ checkKeyValidity(shareEntry)
    If toDelete then
      delete(KeyShares,hash(shareEntry.pub))
```

3.9 Summary

In this chapter, we have presented our system model and architecture for an agnostic, scalable, and customizable component that can integrate with different blockchain typologies and enable co-signing of client transactions.

As we have seen in this chapter, we could not fully modify a blockchain node to integrate group cosignatures without compromising the portability and adaptability of our solution to different blockchain platforms with a variety of configurations, so we split into two main components: (i) a signer node, responsible for cosignatures and transaction validations; (ii) and a validator node, whose goal is to store transactions along with the cosignature in a distributed immutable ledger to enable auditability and decentralization. To support different types of membership, we defined two protocols, one for permissioned blockchains, which assumes that an administrator has previously set up the signer nodes, and one for permissionless blockchains, which places the responsibility of installing the group key shares on the most interested party, the client.

We also introduced two novelty components: i) a crypto-provider capable of including primitives in any language and hot swap cryptographic primitives at runtime; ii) and an extended smart contract that allows to dynamically change the various parameters of the crypto-provider and accept or reject a transaction, at the co-signing phase.

In the next chapter, we will discuss the implementation of our prototype using the system model and software architecture presented in this chapter.

Chapter 4

System Implementation

In this chapter, we present our prototype implementation considering the system model and software architecture shown in the chapter 3. We start with the prototype overview to describe at a high level the components that were implemented and the technologies used for each of them. Then we go deeper and present a possible implementation that follows our system model and architecture.

4.1 Prototype Overview and Technologies

Prototype Overview: To give an overview of our prototype, we describe the main services and the structure of the components below:

- Signer Node[1] is the component responsible for producing threshold and group signatures and contains subcomponents such as, the crypto provider[2], REST API, P2P network, smart contract engines [3] [4], signature manager and some shared utils [5]. We implemented the signer node from scratch.

- Blockchain Validator Node represents any node of a blockchain platform. To prove that our solution is portable and agnostic against any implementation, we selected two different blockchain[6] [7] nodes that we adapted and integrated into our solution with minimal code changes by providing support for threshold and group signatures to increase the degree of resilience in these blockchain platforms.

[1] **Signer Node**: https://github.com/jffp113/SignerNode_Thesis
[2] **Crypto-Provider**: https://github.com/jffp113/CryptoProviderSDK
[3] **Algorand Smart Contract Engine**: https://developer.algorand.org/docs/features/asc1/
[4] **Hyperledger Smart Contract Engine**: https://github.com/jffp113/sawtooth-smartcontract
[5] **Shared Utils**: https://github.com/jffp113/go-util
[6] **Adapted Algorand**: https://github.com/jffp113/go-algorand
[7] **Adapted Sawtooth**: https://github.com/jffp113/sawtooth-core

- The benchmark tool[8] represents a client we developed to test, evaluate and validate our implemented prototype.

Technologies: As we mentioned earlier, we changed two blockchain platforms to prove the agnosticism and portability of our solution. For a permissioned blockchain platform, we choose Hyperledger Sawtooth[9], and for a permissionless blockchain, we choose Algorand[10], as they are two distinct blockchain platforms with different properties.

Hyperledger Sawtooth version 1.2.6[11] is written in both Python 3[12] and Rust 1.50.0[13] languages and can be supported by a variety of consensus protocols (e.g., PBFT, RAFT and PoET), but to provide byzantine fault tolerance we have chosen the PBFT consensus protocol.

To communicate with a Sawtooth node, it is necessary to do an HTTP REST call with the payload marshaled in the Google Protobuf[14] serialization technology. To support group signatures, we only had to modify the Python 3 code to add group signature verifications, and add new fields to the transaction and ledger block representation so that we can persistently store the group signature in the ledger. Sawtooth supports smart contracts written in many languages. To extend the smart contracts, we decided to modify the SDK[15] provided by Sawtooth, which is written in Golang 1.15, incorporating a way to validate the transactions and parameterize the crypto provider in the validator node.

Algorand version 2.5.* is written in Golang 1.15[16] and is supported by a PPoS consensus protocol. Unlike Sawtooth, which uses a well-known marshall protocol, Algorand defines its own data marshalling protocol called MSGP[17] (Message Pack), with client-to-node communication using REST with MSGP or JSON payload. As with the Sawtooth node, we supported group signature verification in Algorand, but in this case we added verifications written in Golang instead of Python. We also added new fields to transactions and blocks written in MSGP to make the group signature persistent and auditable under a distributed ledger. Algorand's smart contracts are written in TEAL, a stack machine language similar to Assembly. To extend these smart contracts, we created a TEAL interpreter in Golang using a subset of the operations available in TEAL[18]

We built the signer node from scratch using Golang 1.15. Different signer nodes

[8] **Benchmark tools**: https://github.com/jffp113/Thesis_Client
[9] **Hyperledger Sawtooth**: https://github.com/hyperledger/sawtooth-core
[10] **Algorand**: https://github.com/algorand/go-algorand
[11] **Sawtooth**: https://sawtooth.hyperledger.org/docs/core/releases/1.2.6/introduction.html
[12] **Python**: https://www.python.org/
[13] **Rust**: https://www.rust-lang.org
[14] **Protobuf**: https://github.com/protocolbuffers/protobuf
[15] **Sawtooth Smart Contract SDK**: https://github.com/hyperledger/sawtooth-core
[16] **Golang**: https://golang.org
[17] **MSGP**: https://github.com/algorand/msgp
[18] **TEAL**: https://developer.algorand.org/docs/reference/teal/specification/

communicate with each other using a library called LibP2P[19] version 0.13.0, which allows to easily build a publish-subscribe architecture on top of a Peer-To-Peer network. Protobuf is used to serialize any communication that takes place with the signer node. Our crypto-provider is also written in Golang, but uses a Router-Dealer communication pattern implemented by the ZeroMQ[20] messaging library, which allows primitives to be integrated in any language. We have integrated threshold BLS[21] and threshold RSA[22], both written in Golang, as our prototype cryptographic primitives. The same communication pattern Router-Dealer is used in the smart contract engine to support different implementations of smart contract engines that can be attached to our signer node. In our case, we have two different smart contract engines, one for Algorand and one for Sawtooth.

The deployment of the blockchain validator nodes, signer node, crypto-provider primitives, and smart contract engines was done using containerization technology, more specifically Docker 20.10.5[23] and Docker Compose 2 [24]. During development, it was used for fast deployment time and local virtualization of a network, during experimental evaluation and analysis, it was used to minimize deployment time on our distributed environment.

4.2 Prototype Architecture and Implementation

As mentioned earlier, our prototype implementation consists of three main components: i) signer node, which contains the implementation of subcomponents that we will discuss later in this chapter; ii) validator node, which can be a Algorand Node or a Sawtooth node; iii) and a client, which is responsible for generating the workload to simulate a real world client. The architecture of the implemented prototype can be seen in Figure 4.1.

In terms of implementation effort, the entire source code of the prototype, including the code base for the signer node components, and the two distinct nodes of the blockchain platform, are mapped to approximately 8315 lines of code, as shown in Table 4.1. The Validator node and its components, built from scratch, resulted in approximately 7285 lines of code. The changes to the Hyperledger Sawtooth and Algorand nodes are approximately 150 and 130, respectively.

Regarding the complexity of our implementation, we had to work with different technologies, for example, our prototype includes four different programming languages (Golang, Python, Rust, TEAL), four different communication libraries and protocols (LibP2P, REST HTTP, ZeroMQ, UNIX sockets), two different blockchain implementations (Algorand, Sawtooth), and three different marshaling protocols (JSON, Protobuf,

[19] **LibP2P**: https://github.com/libp2p/go-libp2p
[20] **ZeroMQ**: https://github.com/pebbe/zmq4
[21] **tBLS**: https://github.com/dedis/kyber
[22] **tRSA:** https://github.com/niclabs/tcrsa
[23] **Docker**: https://docs.docker.com/install/
[24] **Docker Compose**: https://docs.docker.com/compose/

Table 4.1: Prototype implementation extension metrics (LoC)

Components	Estimated LoC
Signer Node (Total)	7285
Hyperledger Extended Smart Contract Engine (Changes)	130
Sawtooth Smart Contract Specification	200
Algorand Extended Smart Contracts Engine	700*
Algorand Smart Contract Specification	130
Algorand Smart Contract Specification	130
Smart Contract Engine Proxy	380
Rest API	100
Protocol Manager	1435
Crypto-provider engine proxy	200
Crypto-Provider Engine	700
Crypto-Provider Handler TBLS and TRSA	1310
Communication Protocols (P2P,Engine-Proxy)	700*
Util	950
Signer Node Client	350*
Benchmarker	750*
Sawtooth Changes	150
Algorand Changes	130
Total	8315

MSGP). The complexity of the implementation is also increased by the fact that we respect our System Goals to create an agnostic, portable, and extensively configurable prototype. In addition, the lack of documentation and the number of cross-component changes for Sawtooth and other frameworks also increase the effort of the implementation.

In our prototype, we tried to achieve the maximum code quality and code coverage. When it was not possible to write unit tests, we performed integration tests and benchmarks to prove that our prototype works correctly.

The following subsections show how to different components and subcomponents were implemented using the architecture presented in Chapter 3, and how they communicate with each other to create a fully functional prototype that can be integrated into any blockchain or distributed system.

4.2.1 REST API & Interconnect

As shown in Figure 4.1, the REST API enables communication between a client and a signer node through various defined endpoints. The API is written in Golang and takes advantage of the HTTP mux package available in the Golang standard library. A client can initiate an interaction with one of the signer nodes by sending a request using HTTP over Transport Layer Security (TLS) to one of the available endpoints in the API. The request must be marshalled using Google's Protobuf protocol. When the request reaches the REST API, it is verified to see if it was created correctly, and only then is it submitted to our Interconnect component, which returns a Golang channel so that our Rest API can

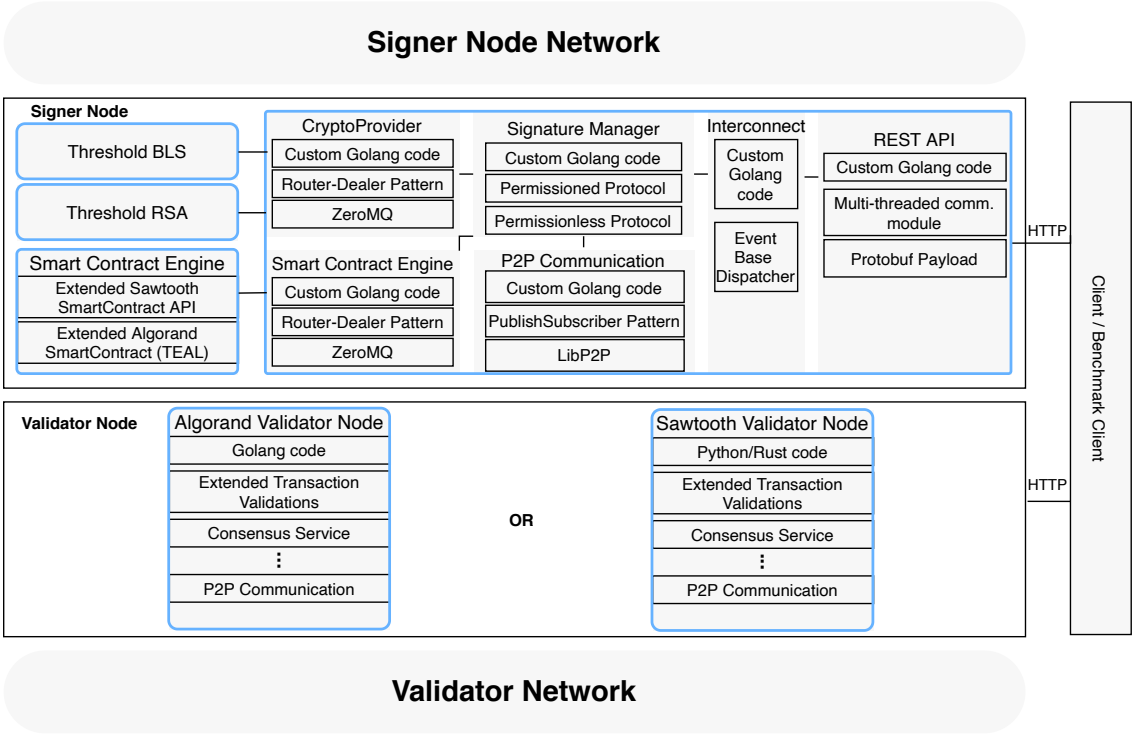


Figure 4.1: Prototype architecture: the prototype architecture is divided into two main components: a) Signer Node; b) Validator Node. Subsequently, each component is divided into sub-components. Components with blue line around them represent a Docker containerized component.

asynchronously wait for a response from a particular registered protocol.

The interconnect component is also written in Golang and serves as an intermediate dispatcher between the REST API and a registered protocol handler. New protocols can register handlers in the Interconnect component to respond to specific incoming messages from the REST API (similar to an event base or a publish-subscriber architecture). If there is no registered handler for a particular message, the interconnect discards the message. If there is more than one registered handler, the message is processed by each handler in the order in which they were registered.

Our REST API exports a subset of the endpoints defined in chapter 3, which are described in detail below:

URL: https://<address>:<port>/sign *Method*: POST *Request format*: application/protobuf (Listing 4.1) *Response format*: application/protobuf (Listing 4.2)
Allows a client to request a group of witnesses to co-sign a particular transaction, producing a group signature signed by a set of signer nodes. If a smart contract with the requested <*SmartContractAddress*> is available, the request is co-signed with the specification resulting from the execution of the smart contract. After the co-signature is generated, the signature is returned to the client with all necessary meta-information.

URL: https://<address>:<port>/verify *Method*: POST *Request format*: application/protobuf (Listing 4.3) *Response format*: application/protobuf (Listing 4.4)
Allows a client to verify a group signature produced by a group of witnesses. This endpoint will return whether the signature is valid or not.

URL: https://<address>:<port>/install *Method*: POST *Request format*: application/protobuf (Listing 4.5)
Allows a client to install key shares in a signer node to be used during a co-signing session. The client must invoke this endpoint in all selected signed nodes that participate in the same witness group. A request to this endpoint is always successful.

URL: https://<address>:<port>/membership *Method*: GET *Response format*: application/protobuf (Listing 4.6)
Allows retrieval of a subset of the membership known to the signer node that the client contacts. A request to this endpoint always succeeds, but the response may have an empty set of members.

To marshall the data in our REST API and all Signer Node components, we choose to use Protobuf technology. Our choice to use Protobuf was not random. Protobuf is an efficient way to encode data in an extensible binary format that allows you to easily change the protobuf structure while maintaining integrity and backwards compatibility with the old message structures. In our REST API, we have defined 8 different message structures, two for each endpoint.

Listing 4.1 shows the structure of a sign request message. A Sign Request can be unequivocally identified by a *UUID* and contains a *SmartContractAddress* that is used by the signature manager to correctly select which smart contract to use to process a particular transaction, it also contains a field with the *content* to be signed, usually this

content contains a encoded transaction understood by the smart contract logic. If this sign message is used in a permissionless environment, the client may also specify a *keyId* to select a key previously deploy by a client.

Listing 4.2 shows the structure of a sign response message. A Sign Response message contains the scheme, represented as a string used to sign the transaction, and the group cosignature, represented in bytes.

Listing 4.1: Protobuf message used to encode a client sign request.

```
message ClientSignMessage{
  string UUID = 1;
  string SmartContractAddress = 2;
  bytes Content = 6;
  string KeyId = 7;
}
```

Listing 4.2: Protobuf message used to encode a sign response.

```
message ClientSignResponse{
  string scheme = 1;
  bytes signature = 2;
}
```

Listing 4.3 shows a structure of a verify message. A client who wishes to verify a signature must specify the *scheme* used to sign the message, the *public_key* of the witness group, the *disgest* of the transaction, and the group *signature* to be verified. A response to a verify message, shown in listing 4.4, contains only a status informing whether the signature is *invalid* or *ok*.

Listing 4.3: Protobuf message used to encode a client verify request.

```
message ClientVerifyRequest{
  string Scheme = 1;
  bytes public_key = 2;
  bytes digest = 3;
  bytes signature = 4;
}
```

Listing 4.4: Protobuf message used to encode a verify response.

```
message ClientVerifyResponse{
  enum Status {
    STATUS_UNSET = 0;
    OK = 1;
    INVALID = 2;
  }
  Status status = 1;
}
```

Listing 4.5 shows a message structure sent by a client to perform a key installation in one of the signer nodes. A client must provide in this message a group *public_key* encoded in bytes to allow a signer node to aggregate group signatures, a *private_key* share encoded in bytes to allow the signer node to sign a particular transaction, a *validUntil* indicating the time at which a key expires as Unix time (the number of seconds elapsed since January 1,1970 UTC), and a *isOneTimeKey* encoded in a bool to represent whether a key can only be used once.

Listing 4.5: Protobuf message used to encode a client key installation request.

```
message ClientInstallShareRequest{
  bytes public_key = 1;
  bytes private_key = 2;
```

```
  int64 validUntil = 3;
  bool isOneTimeKey = 4;
}
```

Listing 4.6 shows a message structure sent in response to a client get membership request. A response consists of a *status* indicating whether the request succeeded, and a set of *peers*, each consisting of a set of distinct addresses (*addr*) to represent the possibility of a signer node listening on more than one network interface, and a *id* to unequivocally identify a signer node.

Listing 4.6: Protobuf message used to encode a membership response.

```
message MembershipResponse{
  enum Status { STATUS_UNSET = 0; OK = 1; INVALID = 2; }
  message peer{
    string id = 1;
    repeated string addr = 2;
  }
  Status status = 1;
  repeated peer peers = 2;
}
```

4.2.2 CryptoProvider

We designed and implemented our crypto-provider as described in chapter 3 to build a pluggable and factory modular crypto-provider. The crypto-provider is divided into two components: i) a provider layer, where new primitives are registered; ii) and a cryptographic client. The communication between these two components is done using TCP or Unix sockets. Since we used the ZeroMQ communication library for the communication between the client and the engine and employed a Router-Dealer pattern, our client can support multiple provider layers in arbitrary languages, such as Java, Python, Ruby, C, etc. To create a proof-of-concept, we implemented a provider layer engine written in Golang and included various signature schemes such as RSA and BLS and their respective threshold versions. This is only possible because our crypto-provider is designed to support conventional signatures and group signatures.

In order to smoothly deploy the engine with the respective primitives, we containerized the engine using Docker containers, which allows easy integration of new primitives by simply plugging in a new engine with other primitives. If a bug is found in a primitive in one of the engines, it is possible to hot-swap with a new primitive without stopping the system. To do this, we first plug in the new engine, which informs the cryptographic client that each primitive is available at a new address, and only then do we stop the old engine.

To include new primitives in our crypto-provider, it is necessary to implement the methods present in the listing 4.7. There is one method for each cryptographic signature operation, one for signing, one for verifying, and another for aggregating the various

signature shares produced by the signer nodes. There is also a method for generating a public key and a list of private key shares. Since the primitives may have different formats of public and private keys, they must implement methods to unmarshal and marshal the keys to and from binary format. To identify the primitive and allow a cryptographic client to invoke it, it is necessary to implement the SchemeName.

Listing 4.7: Methods had to be implemented to add a new primitive to our crypto provider.

```
1  func (t *tbls) Sign(digest []byte, key crypto.PrivateKey) ([]byte, error) {...}
2
3  func (t *tbls) Verify(signature, msg []byte, key crypto.PublicKey) error {...}
4
5  func (t *tbls) Aggregate(shares [][]byte, digest []byte, key crypto.PublicKey, t, n int) ([]byte,
       ↪ error) {...}
6
7  func (t *tbls) Gen(n int, t int) (crypto.PublicKey, crypto.PrivateKeyList) {...}
8
9  func (tbls *tbls) SchemeName() string {...}
10
11 func (tbls tblsHandler) UnmarshalPublic(data []byte) crypto.PublicKey {...}
12 func (tbls tblsHandler) UnmarshalPrivate(data []byte) crypto.PrivateKey {...}
```

After implementing a primitive, it is necessary to create the engine (processor) and point it to the URL of the signer node, as shown in the listing 4.8. Here it is possible to choose a variety of protocols, from inproc to TCP. Now we are ready to register the primitives we want to make publicly available in the signer node. To do this, we call AddHandler in the engine on all the primitives we want to register.

Listing 4.8: Example of how to register a new primitive in a provider layer engine.

```
1  processor := crypto.NewSignerProcessor(opts.SignerNodeURL)
2  processor.AddHandler(tbls.NewTBLS256CryptoHandler())
3  processor.AddHandler(trsa.NewTRSACryptoHandler())
```

4.2.3 Smart Contract Engine

For this prototype, we implemented two different smart contract extensions, one for Hyperledger Sawtooth smart contracts and the other for Algorand smart contracts, following the specification shown in the previous chapter. Since the smart contracts have completely different natures, similar to the crypto-provider, it had to be split into three components so that our blockchain is agnostic to the smart contract language used. The three components are: a) an engine to execute/interpret extended smart contracts of a given blockchain; b) a client to invoke smart contracts in the engine; c) and the smart contract itself. Similar to the crypto-provider, the engine and client communicate via the ZeroMQ communication library with a Router-Dealer pattern that allows multiple engines to be connected to a single client. Again, we decided to containerize the smart contract engine in a Docker container to deploy the engines faster and be able to switch them around during execution.

To keep it simple, the main logic of a smart contract is capable of receiving three types of operations that manipulate the storage of a smart contract, such operations are, an inc operation that increments a value in a given key, a dec operation that decrements a value in a given key, and a set operation that sets a value in a given key. Both smart contracts that we have extended to validate transactions can also perform these three operations.

Hyperledger Sawtooth supports smart contracts across a range of languages. We chose to extend the smart contract SDK written in Golang because almost all the components we develop are written in Golang. Listing 4.9 shows an excerpt of an extended Hyperledger Sawtooth smart contract that follows the specification available in chapter 3. We have extended the base smart contract with a new function called *Validate* (in line 16,) that receives a smart contract validation request, validates the transaction, and generates the correct parameterization for our crypto provider. In the case of our excerpt, we have simplified the *Validate* function so that it always generates the same parameterization for each transaction. However, we can add roles in this function to choose the parameterization depending on the transaction we want to process.

Listing 4.9: Excerpt of a Hyperledger Sawtooth Smart Contract written in Golang.

```
1  type IntkeyPayload struct {
2    Verb,Name string
3    Value int
4  }
5  type IntkeyHandler struct {
6    namespace,scheme string
7    t,n int
8  }
9  func NewIntkeyHandler(namespace string,t,n int, scheme string) *IntkeyHandler {
10   return &IntkeyHandler{namespace,t,n,scheme}
11 }
12 //...
13 func (self *IntkeyHandler) FamilyName() string { return FAMILY_NAME }
14 func (self *IntkeyHandler) FamilyVersions() []string { return []string{"1.0"} }
15 func (self *IntkeyHandler) Namespaces() []string { return []string{self.namespace} }
16 func (self *IntkeyHandler) Validate(request *smartcontract_pb2.SmartContractValidationRequest) (
       ↪ processor.ValidateResponse, error) {
17   //can have very complicated logic over here to determine the parameters to sign a transaction
18   return processor.ValidateResponse{
19     //Type: THS/normal/multisignature
20     SignatureScheme: self.scheme,
21     N: self.n,
22     T: self.t,
23   }, nil
24 }
25 func (self *IntkeyHandler) Apply(request *processor_pb2.TpProcessRequest, context *processor.Context
       ↪ ) error {
26   //... Transaction decoding and validations
27   var newValue int
28   storedValue, exists := collisionMap[name]
29   if verb == "inc"  verb == "dec" {
30     if !exists {
31       return &processor.InvalidTransactionError{Msg: "Need␣existing␣value␣for␣inc/dec"}
```

```
32      }
33      switch verb {
34      case "inc":
35        newValue = storedValue + value
36      case "dec":
37        newValue = storedValue - value
38      }//... more validations
39    }
40    if verb == "set" {
41      if exists {
42        return &processor.InvalidTransactionError{Msg: "Cannot set existing value"}
43      }
44      newValue = value
45    }
46    collisionMap[name] = newValue
47    data, err = EncodeCBOR(collisionMap)
48    if err != nil {
49      return &processor.InternalError{
50        Msg: fmt.Sprint("Failed to encode new map:", err),
51      }
52    }
53    addresses, err := context.SetState(map[string][]byte{
54      address: data,
55    })
56    if err != nil {
57      return err
58    }
59    if len(addresses) == 0 {
60      return &processor.InternalError{Msg: "No addresses in set response"}
61    }
62    return nil
63  }
```

Algorand supports smart contracts via TEAL, a non- Turing-Complete stack machine language. Listing 4.10 and 4.11 shows an excerpt from an extended Algorand smart contract. Since a stack machine language does not have a function abstraction, we had to take a different approach to extend TEAL smart contracts. We decided that the best approach was to add a new transaction type called ValidateTransaction. Teal smart contracts have the ability to use a branch operator (b,bz,bnz) to jump ahead to a specific label. In our case, if a transaction is of type ValidateTransaction, we jump to the label validate_transaction (,line 14). Below the label, many verifications can be performed on the transaction, but for simplicity we have not defined transaction validation. Normally, a correct execution in an Algorand smart contract ends with a value of one on the stack. However, in our implementation, the execution of a transaction over a Teal smart contract must leave three values on the stack to tell the signer node how to sign a transaction: i) the signature scheme to use; ii) the number of signer nodes to sign; iii) and the lower bound on the correct signature shares required to create the group signature.

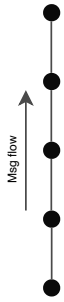
Listing 4.10: Algorand SC excerpt first part.

```
txn OnCompletion
int ValidateTransaction
==
bnz validate_transaction

txn OnCompletion
int NoOp
==
bnz handle_noop
//... Other transactions
// Unexpected OnCompletion value.
err //Should be unreachable.

validate_transaction:
byte "TBLS256Optimistic"
int 5
int 3
return
handle_noop:
//Verify if the transaction does not
    ↪ contain any
//argument
txn NumAppArgs
int 0
==
bnz return_succ
//Verify if the number of arguments
    ↪ equals to 2
//Otherwise give error
txn NumAppArgs
int 3
!=
bnz error
//Check what is the smartcontract
    ↪ operation
//First check if is a inc op
txna ApplicationArgs 0
byte "inc"
==
bnz inc_op

//Next check if is a dec op
txna ApplicationArgs 0
byte "dec"
==
```

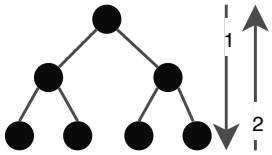
Listing 4.11: Algorand SC excerpt second part.

```
bnz dec_op

//Finally check if is a set op
txna ApplicationArgs 0
byte "set"
==
bnz set_op

//Inc Operation starts here
inc_op:
txna ApplicationArgs 1
dup
app_global_get
txna ApplicationArgs 2
btoi
+
app_global_put
b return_succ //Return Successfuly

//Dec Operation starts here
dec_op:
txna ApplicationArgs 1
dup
app_global_get
txna ApplicationArgs 2
btoi
-
app_global_put

b return_succ //Return Successfuly

//Set Operation starts here
set_op:
txna ApplicationArgs 1
txna ApplicationArgs 2
btoi
app_global_put
b return_succ //Return Successfuly

return_succ:
int 1 //Return Successfuly
return
//... Other transaction types
error:
err
```

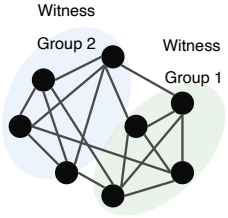
4.2.4 P2P Network



(a) LinkedList Overlay. Messages flow from down to up. When minimum threshold is reached propagation can stop.



(b) Tree Overlay. It is necessary to have two messages propagation: 1) to propagate down the message to sign; 2) and to collect the signature shares when ascending.



(c) PubSub Overlay. Peers in the same group have a high level of connectivity. Peers between different groups have low level of connectivity.

Figure 4.2: Possible Overlays for our Signer Node P2P Network

In building our signer node prototype, if we did not include a specialized P2P network for the signer nodes, the responsibility for aggregating the signature shares would be transferred to a client, which must contact each signer node individually to ask it to produce a signature share. This approach makes group signatures cumbersome for our clients, which have to contact each signer node for its signature share in order to aggregate them, plus the client would now require more resources and energy, which eliminates the possibility of mobile devices being candidate clients.

Our approach to this problem is based on a specialized P2P network in which signer nodes communicate with each other to collaboratively create a final group signature. Here, we could have used a simple linked-list communication pattern (Fig. 4.2a), where signer nodes would produce their signature shares and pass all previous signatures plus their own to the next signer node in the list, producing $2n$ messages, or t messages if a signer node with enough shares can reply directly to the client. This approach would not be reliable, since any node crash before a signer node could collect a minimum number of threshold signatures, could bring the system to a halt. Another approach might be to create a tree-like overlay (Fig. 4.2b), which would produce $2log_2(n)$ messages. This approach is more reliable than a list, but crashes near the tree root can halt the signature process.

Since normally threshold signature groups are small, we decided to allow each signer node to be connected with all members of the same witness group, as shown in Figure 4.2c. To help us with this problem, we chose a framework called LibP2P, which allows us to create a publish/subscribe system over a P2P network. This way, only members of a given witness group need to receive each share and can create the final signature. An obvious problem with this approach is the large number of messages produced, however the groups are small and each signer node can reply to the client with the final signature.

Nonetheless, we implemented a static configuration that, in low-byzantine environments, allows the signature to be sent only to the requester signer node, reducing the number of messages produced from n^2 to $2n$. This approach can significantly improve the transaction throughput of our prototype.

4.2.5 Signature Manager

The signature manager is an essential component that hosts the various protocols (permissioned and permissionless) defined in the previous chapter. We designed it in such a way that in the future it will be easy to integrate new protocols into our prototype with zero lines of code changes. When the signature manager is created, a protocol is injected in a kind of factory pattern, which allows the protocol to register each handler in the interconnect component, then proxies to the smart contract engine and crypto provider are created.

For example, to create a new protocol, we just need to define a method that accepts a interconnect component with a registration function, as shown in Listing 4.12.

Listing 4.12: Interface to be implemented by a new protocol and Interconnect interface.

```
type Interconnect interface {
  RegisterHandler(t HandlerType, handler Handler)
  EmitEvent(t HandlerType, content ICMessage) HandlerResponse
}
type Protocol interface {
  Register(ic ic.Interconnect) error
}
```

Then a protocol simply registers the handlers using the RegisterHandler function, as shown in Listing 4.13.

Listing 4.13: Permissionless protocol functions registration.

```
func (p *permissionlessProtocol) Register(interconnect ic.Interconnect) error {
  interconnect.RegisterHandler(ic.SignClientRequest, p.sign)
  interconnect.RegisterHandler(ic.InstallClientRequest, p.installShares)
  interconnect.RegisterHandler(ic.NetworkMessage, p.processMessage)
  p.interconnect = interconnect
  return nil
}
```

4.2.6 Client and Benchmarker

This work uses a whole set of heterogeneous distributed systems to achieve the goal. To this end, we decided to implement a simple benchmark tool that allows us to guarantee the same testing conditions between the different systems. Our benchmarker is written in go and in an extensible way to support our test handlers and other handlers that can be integrated later. The benchmarker can be parameterized with the number of concurrent clients sending requests, the handler to be used to send requests to a given system, and the duration of the tests.

This benchmarker supports different handlers such as:

- **Sawtooth**: This handler allows to benchmark the Hyperledger Sawtooth without integrating with the signer node to produce the threshold signature and store it in the immutable ledger.

- **Algorand**: Like Sawtooth, this handler allows benchmarking the Algorand without the integration with the signer node.

- **Signer Node**: This handler allows benchmarking of the signer node, which means we can get metrics without integrating with a blockchain to get realistic results about the cost of producing a group signature in different environments without the mechanics of different blockchain platforms getting in the way.

- **SignerNode + Blockchain**: Our benchmarker also includes an integration between our signer node and a extended blockchain platform (Algorand and Sawtooth).

The client can behave in two modes, permissioned or permissionless. If we recall the permissioned mode, the key share setup has already been done in the signer node and our client only needs to send a request with the smart contract to be executed and the transaction to be signed. After the request is received, if everything goes right, the client will receive back the group signature associated with that transaction and can send it to the corresponding blockchain node. Permissionless mode is a bit more complex, as it puts the burden on the client to install the key shares, which can be a bit heavy if we want a lot of security in our transactions. To this end, we have implemented two permissionless modes, one where we simply provision new key shares from time to time, and another where each key can only be used once, which means that if we send a lot of transactions, we need to generate a key before sending the transaction, or already have a pool of keys generated in advance. As can be seen, the first method is less secure because we can reuse the key for many transactions, and the second method is very resource intensive. The decision to use one or the other method depends on the sensitivity of the operation we are performing on the blockchain.

Another thing that our client implements is group selection. We have implemented two methods. One method simply randomly selects a subset of nodes from the membership, and the other method selects nodes according to their proximity. We implemented the second approach using the ICMP protocol, where our client probes (pings) each signer node and computes the average RTT between the client and the signer node, then selects the n nearest neighbors. Of course, this implementation has a flaw when packets are dropped by the server that is hosting the signer node. To circumvent this problem, we could have implemented an endpoint in the signer node, simply for pinging.

4.2.7 Validator Node

To prove that our signer node is independent of the blockchain used, we integrated it with two different blockchain platforms: i) Hyperledger Sawtooth for a permissioned blockchain; ii) and, Algorand for a permissionless blockchain. When prototyping and designing our signer node, our goal was to create something that required few code changes to be integrated with the signer node.

In both Algorand and Hyperledger Sawtooth, we had to change the way data is stored in a block to incorporate the notion of group witnessed transactions, and we also introduced new verifications that are performed when these transactions arrive.

Hyperledger is a bit special and has a higher level of abstraction called a batch. A batch is nothing but a set of transactions that need to be committed atomically together. Since a transaction must always be encapsulated within a batch, we decided to include the group signature in a batch, which means that the signature is performed over a set of transactions, as shown in Listing 4.14.

Listing 4.14: A batch with an incorporated group signature. A set of batches commited together form a block.

```
message GroupEnvelop {
  // Group public key for the signer node witnesses that sign the Batch
  bytes public_key = 1;
  //Group signature over the transactions
  bytes signature = 2;
  //Scheme used to sign the batch with the signature shares.
  string scheme = 3;
}
message BatchHeader {
  // Public key for the client that signed the BatchHeader
  string signer_public_key = 1;
  // List of transaction.header_signatures that match the order of transactions required for the batch
  repeated string transaction_ids = 2;
  // Group Envelop
  bytes group_envelop = 3;
}
message Batch {
  // The serialized version of the BatchHeader
  bytes header = 1;
  // The signature derived from signing the header
  string header_signature = 2;
  // A list of the transactions that match the list of transaction_ids listed in the batch header
  repeated Transaction transactions = 3;
}
```

Algorand is a little bit different, but the concept is similar, it has a signed transaction with all kinds of signatures, like, multi-signature, conventional signatures and now we have introduced in this structure a group signature. In addition to the group signature, since we're in a permissionless environment, we've also included a way to visualize which witnesses are participating in the signing of that transaction to find nodes that are colluding and ensure maximum fairness in that process, as shown in Listing 4.15.

Listing 4.15: A extended Algorand signed transction.

```
// SignedTxn wraps a transaction and a signature.
```

```
type SignedTxn struct {
  ...
  Sig crypto.Signature `codec:"sig"`
  Msig crypto.MultisigSig `codec:"msig"`
  Lsig LogicSig `codec:"lsig"`
  Txn Transaction `codec:"txn"`
  AuthAddr basics.Address `codec:"sgnr"`
  GroupSignature GroupEnvelop `codec:"gsig"`
}

//GroupEnvelop represents a signature from signer node witnesses
type GroupEnvelop struct {
  ...
  //public key for the signer node witnesses that signed the Transaction
  PublicKey []byte `codec:"pubkey"`
  //Group signature over the transactions
  Signature []byte `codec:"sig"`
  //Scheme used to sign the batch with the signature shares
  Scheme string `codec:"scheme"`
  //Witnesses IDs
  IDs []string 'codec:"ids"'
}
```

4.3 Summary

In this chapter, we have presented some details about the prototype and the effort involved in creating such a prototype based on the system model and our system goals. First, we provided a high-level overview of our prototype and described the technology used for each component. Then, we described each component in detail (e.g., the REST API, the interconnect, the cryptoprovider, and P2P network) and explained some implementation decisions we had to make. We also explained some decisions that led us to change Algorand and Sawtooth in some ways.

Our prototype has about 8315 lines of code and was designed to be highly extensible to allow for future proof upgrades, including integration with new protocols, cryptographic schemes, and blockchain platforms. The code is fully open source and available in multiple repositories [25].

In the next section, we present the experimental evaluation and analysis of our prototype.

[25] **Prototype source-code:** https://github.com/jffp113/Thesis_Finder

Chapter 5

Experimental Evaluation and Analysis

In this chapter, we discuss the evaluation of our prototype over the development described in Chapter 4. In Section 5.1 we describe our benchmark environment, then in Section 5.2 and below we present our experimental evaluation results, which we then analyze and explain.

Succinctly, we hope to answer the following questions during our evaluation process:

- What is the transaction throughput and finality latency in a baseline permissioned[1] and permissionless blockchain[2] platform, and how is it affected by the integration of our co-signing service plane?
- How is latency and throughput affected when the number of witnesses and the size of the keys are varied?
- How does the performance of threshold signatures compare in two different environments: Non-integration and integration with our modular crypto provider?
- What is the performance impact of receiving invalid signature shares compared to receiving all valid signature shares, and what is the impact of committee members refusing to sign?
- How does the performance compare between a single datacenter deployment of our prototype and a geo-distributed deployment between datacenters on different continents?
- How long does it take to form committees under different temporal conditions, and how does it impact in a possible aggressive settings for a one-time committee.

[1] Hyperledger Sawtooth
[2] Algorand

5.1 Test-bench Environment

Our test-bench environment to evaluate our implementation, follows the specification below:

- To support our testing, we selected 10 Virtual Private Server (VPS) instances for our prototype and 1 node to serve as support to our benchmark clients, each of which has a technical specification that can be found in the table 5.1.
- Our prototype is fully dockerized so that it can be easily deployed on any of the above machines.
- External components belonging to a signer node (crypto-provider and smart contract engine) are also dockerized, but run in a private network together with the respective signer node.
- Due to the heterogeneity of our prototype, we will use a simple tool[3] we created to benchmark the signer node and the validator node (Algorand, Hyperledger Sawtooth), but we will benchmark isolated signer node components using Golang tools.

	VPS Single Instance	Benchmark Client
CPU	2 vCore	6 Cores (12 CPU threads)
RAM	4 GB	16 GB
Storage	80 GB NVMe SSD	250 GB SSD
OS	Ubuntu 20.10	MacOs 11.2.3
Model	OVH VPS Essential	MacBook Pro 2019

Table 5.1: Benchmark environment technical specification.

Our VPS's are evenly distributed across three data centers:

- 3 instances in Germany (Frankfurt);
- 3 instances in France (Gravelines);
- 2 instances in Canada (Beauharnois);
- 2 instances in Australia (Sydney);

Our clients are located in Portugal, more specifically in Lisbon. Figure 5.1, shows the RTT latency between the data centers where the signer nodes and clients are deployed, measured with the ICMP protocol, more precisely with the tool ping.

The client that is situated in Portugal has latency of 45 ms to Frankfurt, 40 ms to Gravelines, 329 ms to Sydney and 126 ms to Beauharnois

[3] https://github.com/jffp113/Thesis_Client

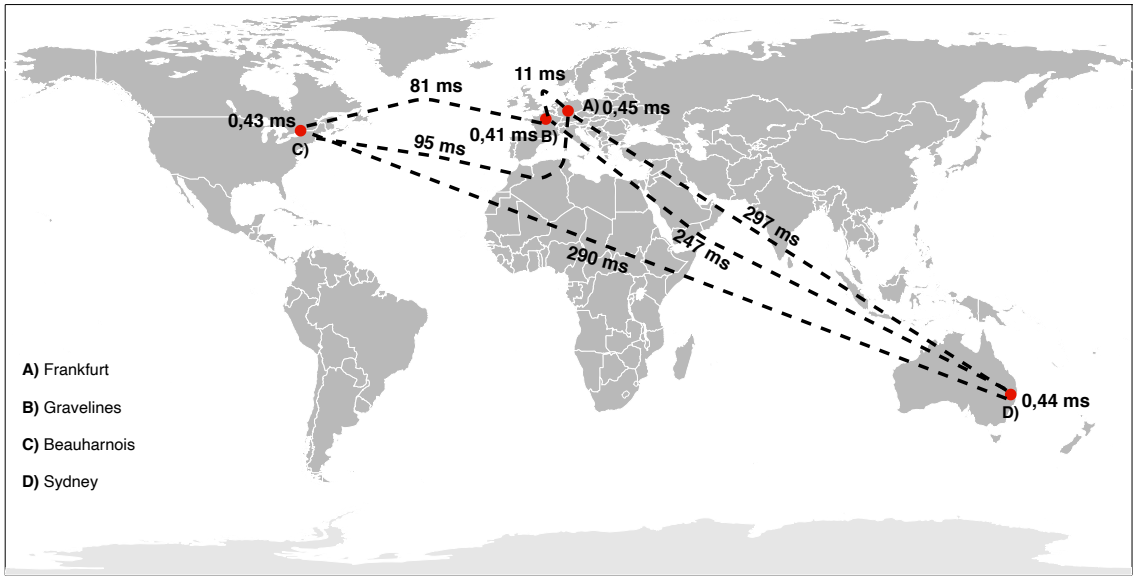


Figure 5.1: Latency between benchmark environment data centers.

5.2 Benchmarks and Analysis

In the following sections, we will perform different experimental evaluations. We will start with a blockchain validator node (Sawtooth and Algorand) as a baseline test, which means that we have not integrated our signer node and the blockchain node has no code changes. Next, we will proceed with a microbenchmark of our crypto-provider to determine the cost of building a language-independent crypto-provider for our signer node. We then conduct an experimental evaluation under the same conditions as in the baseline, but with our signer node (and all subcomponents) integrated. To better analyze the performance of our signer node, we perform an experimental evaluation in which we isolate the signer node. In a more advanced test, we will also evaluate our signer node under different fault assumptions (, such as byzantine and crash faults). Finally, we evaluate the cost of group formation in a permissionless environment.

In all the described experimental evaluations where our signer node and crypto-provider are integrated, we assume a Byzantine fault tolerance of $n = 3f + 1$, where f is the maximum number of faults that can occur simultaneously at a given time. In terms of cryptography, we will perform tests with different schemes, such as threshold BLS (tBLS) with an elliptic curve public key of 256 bits and threshold RSA (tRSA) with different modulus values of N (1024,2048,3072). For the aggregation of signature shares, we will use two different algorithms: i) Optimistic; ii) and Pessimistic. If nothing is mentioned, you can assume that the Optimistic Algorithm is used to combine signature shares into the final signature.

It is also worth mentioning that when using Sawtooth to better approximate Algorand, we will not use any optimization such as batching transactions. Algorand does not support transaction batching, and could give Hyperledger Sawtooth a huge increase in

transaction throughput. However, our signer node supports batch signing and this can be used to increase Sawtooth's performance outside of our experimental evaluation.

5.3 Validator Nodes Baseline Performance Metrics.

Before we start the experimental evaluation of our prototype, we first want to obtain some baseline results to understand the behavior of the two chosen blockchain platforms (Algorand and Sawtooth). To measure the baseline behavior, we decided to launch our blockchain platforms in two different settings: I) A local configuration where four Blockchain nodes are distributed across different machines in the same data center; II) and a distributed setting where four Blockchain nodes are geo-distributed across the aforementioned data centers (Gravelines, Frankfurt, Sydney, and Singapore). During testing, we increased the number of concurrent clients that would send transactions to the different blockchain nodes.

Before we run the baseline tests, we can try to predict some outcomes. Sawtooth uses the PBFT consensus protocol, which in the default configuration tries to publish a block every 1,000 ms and supports 1,024 batches in each block. This means that if we have enough clients and infinite computational power, we could theoretically achieve a maximum of 1,024 transactions per second and a latency of 1,000 ms per operation. For Algorand, we could not obtain the configurations for the time between block proposals and the number of transactions per block, as well as the block size, to calculate the maximum achievable throughput. However, we know that PPoS is designed to be used in a permissionless environment with thousands of nodes, so with enough clients and enough Algorand nodes, we can achieve better transaction throughput than Sawtooth.

Our results in this section have a relative standard deviation of 2–6% for the Hyperledger Sawtooth and 0–3% for Algorand. From the baseline results shown in Figure 5.2, we can see a linear increase in transaction throughput in Algorand as the number of clients increases, and a nearly constant latency of about 8 seconds for Algorand. The results of Hyperledger Sawtooth are somewhat different from those of Algorand. Initially, there is a huge increase in transaction throughput, but when we reach 64 concurrent clients, the transaction throughput stops increasing and becomes constant, and the latency increases from 1 second to almost 4 seconds. We can also see that geo-replication has more impact in Sawtooth than in Algorand nodes.

Although Sawtooth has better transaction throughput in this scenario, we can easily see that the PBFT consensus protocol cannot handle a heavy workload due to the high number of messages required to commit a block, more precisely $O(n^2)$ messages. Had we increased the number of clients or the number of nodes, Algorand would have shown less performance degradation compared to Sawtooth, and at one point even better performance metrics, since the Algorand consensus protocol selects a node for block creation depending on the stack it holds, meaning that the number of messages to propagate is much lower than in PBFT.

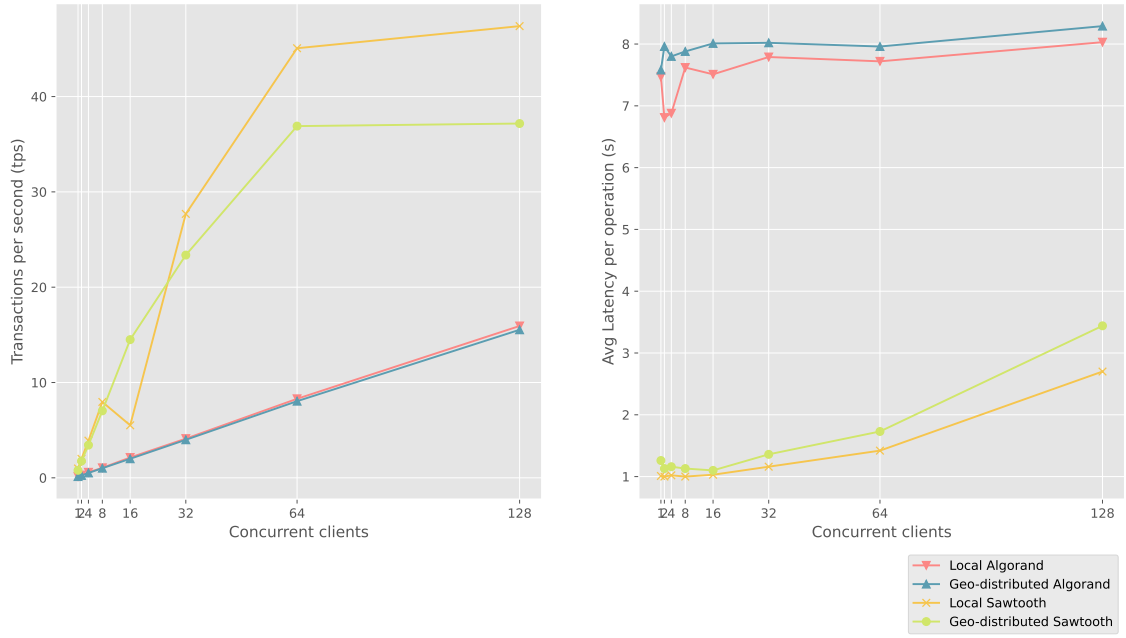


Figure 5.2: Baseline Algorand and Sawtooth latency and throughput when varying the number of concurrent clients.

To analyze whether our statement is true, we decided to run an additional baseline test where we gradually increase the number of blockchain nodes from 4 to 19, evenly geo-distributed across the data centers, to see if Sawtooth suffers more performance penalty due to the time complexity of the PBFT protocol. We set the number of clients to 128, so we expect Algorand to maintain the transaction throughput and latency we saw earlier.

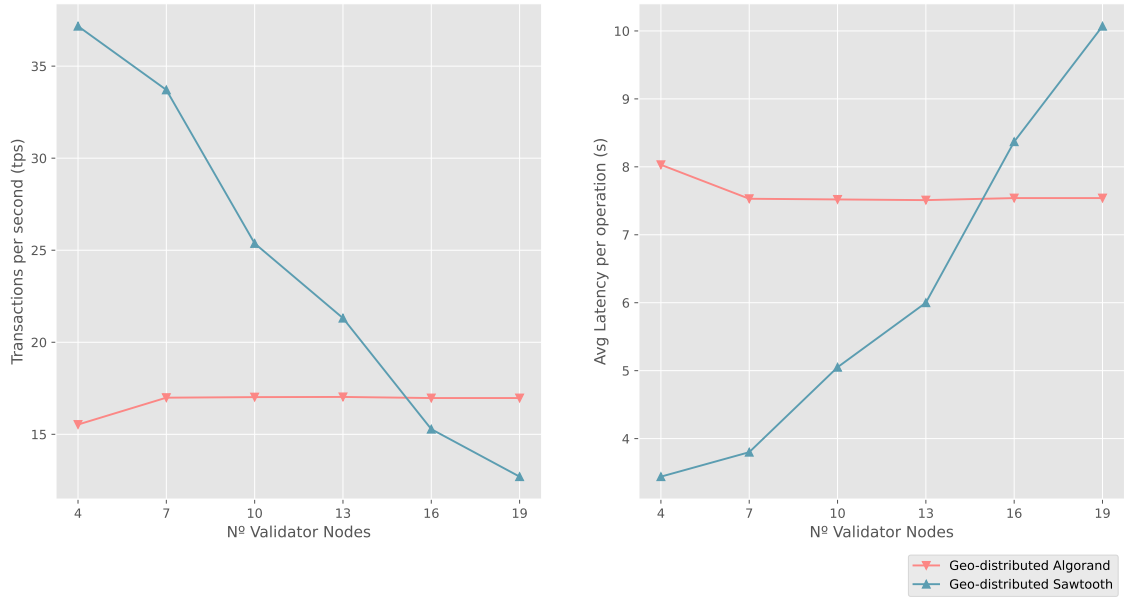


Figure 5.3: Baseline Algorand and Sawtooth latency and throughput when varying the number of signer nodes.

Our additional results are shown in Figure 5.3. As expected, Algorand maintained its performance while increasing the number of nodes. In fact, it even improved transaction throughput and reduced latency from 4 to 7 clients. The explanation for this might be related to the fact that the clients are distributed over more Algorand nodes. On the other hand, Sawtooth abruptly decreased the transaction throughput and increased the average latency per operation, as we would expect. The results also show that Algorand performs better than Sawtooth when the number of nodes increases above 16.

5.4 Signer Node integration with Validator Nodes

In this experimental evaluation, our goal was to evaluate the impact of integrating our signer node with a validator node on the throughput and latency. Similar to the baseline, the test conditions are run with 4 validator nodes, with a signer node attached to each validator node. The nodes are run in two settings, evenly geo-distributed across the different data centers and locally in only one data center. We will also vary the number of concurrent clients to see the performance impact, and use different threshold signature schemes in our signer node, specifically threshold BLS256 and threshold RSA with different modulus sizes (1024, 2048, and 3072). In both signer node protocols (permissioned and permissionless), the key shares are already deployed, to eliminate the cost of deploying the keys, which will be discussed later.

As we add a new security layer on top of a blockchain, we expect a performance loss in both blockchain platforms. This happens because a group signature must now be created, which takes time, and each validator node must validate this new signature and all others that are already supported by the corresponding validator node.

Figure 5.4 and 5.5 show the results for our experimental evaluation with a relative standard deviation of 0–3% and 2–6%, respectively. The first information we can take from the Algorand result is that if we incorporating our signer node with TBLS256 and TRSA1024, the cost is relatively inexpensive, even if we increase the number of concurrent clients sending transactions to the blockchain, however, the threshold RSA with modules 2048 and 3072 have more impact on the latency and transaction throughput. When we incorporating Sawtooth with the Signer node, we immediately see a drop in transaction throughput to more than half and a doubling of latency per operation when we reach 32 concurrent clients.

One explanation for why this is not the case with Algorand could be related to the latency for block finality and the time it takes to create a group signature. As we saw in the baseline tests, Algorand takes about 7–8 seconds to close a block, and this is true even when there are no transactions to include in the block. Sawtooths, on the other hand, takes about 1 second when the load on the system is light, and increases up to 3 seconds when 128 clients send requests to the node at the same time. Algorand suffers less impact, if we do not consider group formation, because if the signer node can create a signature and submit it before the block is closed, the transaction can be included in that block.

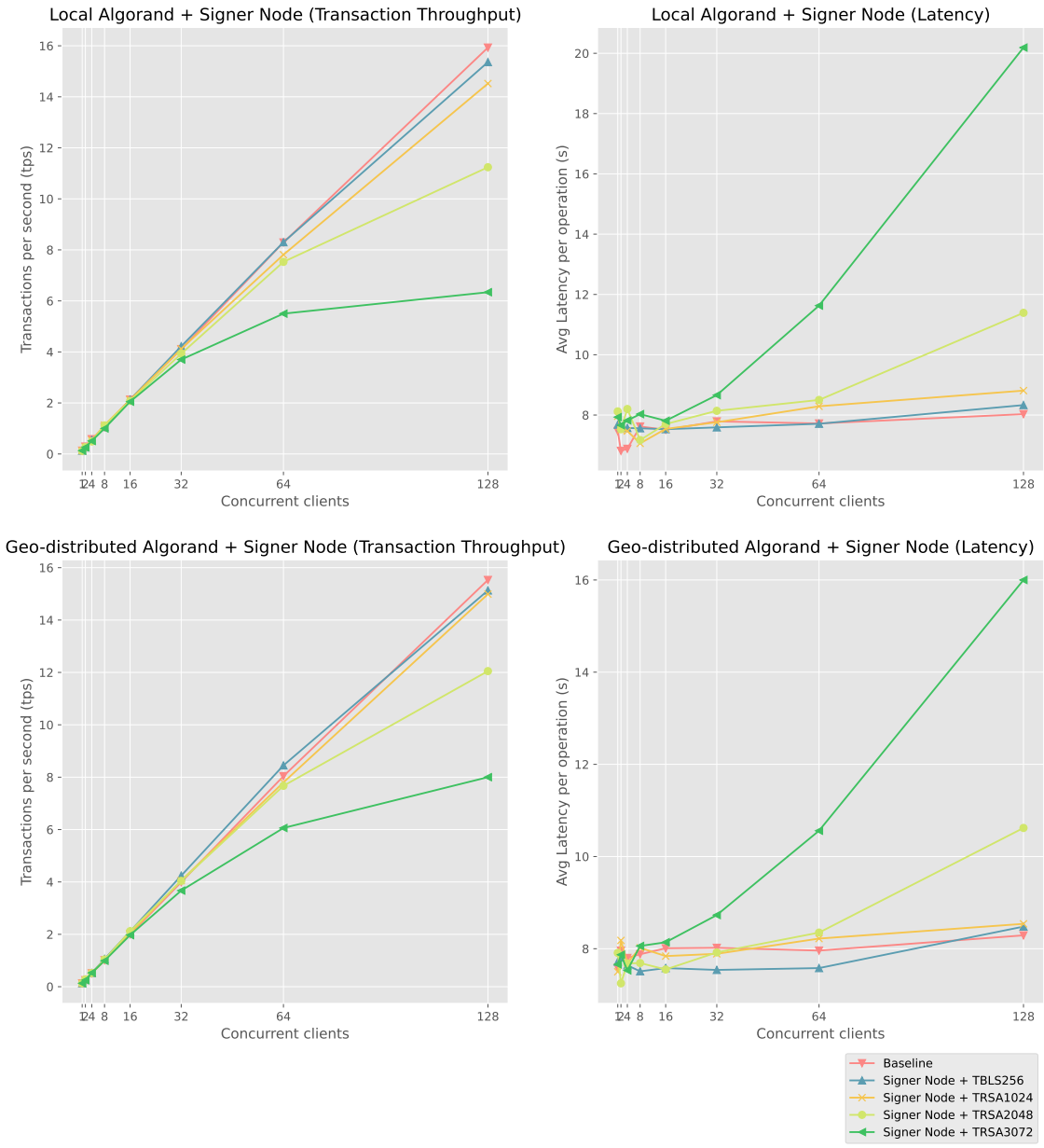


Figure 5.4: Latency and throughput when integrating the signer node with Algorand.

This is not the case with Sawtooth, since the transactions to be included in the block must be known to all validators beforehand in order for the signature to be validated.

In both cases, we observe an increase in latency and a decrease in transaction throughput when we increase the size of the threshold RSA key from 1024 to 2048 and to 3072.

It can also be seen in both Figure 5.4 and Figure 5.5 that the latency tends to be smaller when the nodes are distributed. This can be explained by the phenomenon of head-of-line blocking: The closer the nodes are to each other, the more messages need to be processed per unit time, causing some transactions to take longer and increasing latency for queued requests. We will see this better in further experimental evaluations.

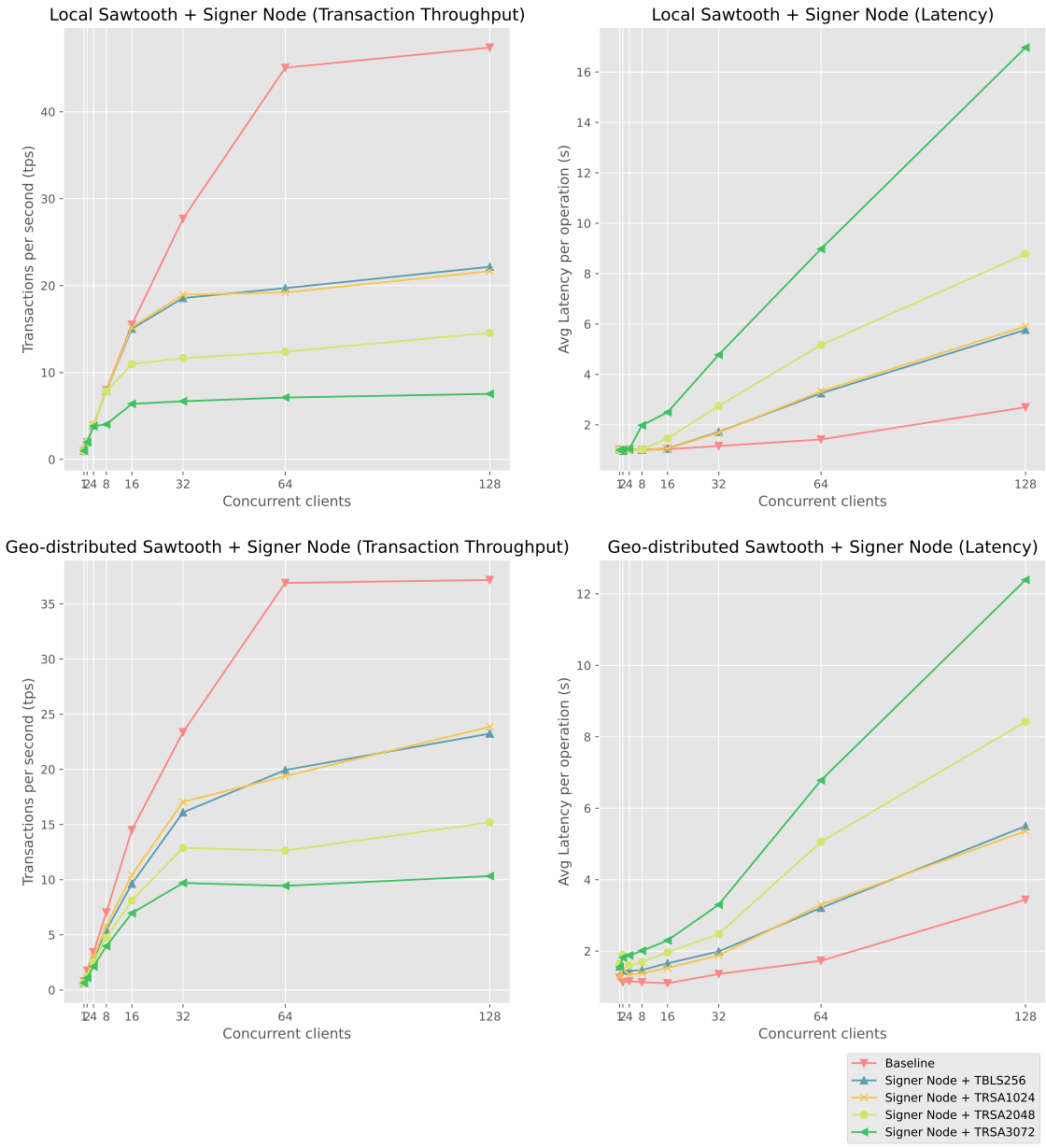


Figure 5.5: Latency and throughput when integrating the signer node with Sawtooth.

5.5 Crypto-provider isolated performance.

In this experimental evaluation, our goal was to evaluate the impact of integrating a threshold primitive scheme into our crypto-provider and compare it to the same primitive outside our crypto-provider. To test each primitive in the different environments, we used the go benchmark tools that repeatedly run the benchmark until it finds that the time per operation converges to a certain value.

Table 5.2 and Table 5.3 show the time required per operation when the primitive is raw and when the primitive is integrated into our cryptoprovider, respectively.

Scheme	Sign (ms)	Aggregate (ms)	Verify (ms)
TBLS256	0,19	8,5	2,7
TRSA1024	2,40	0,25	0,02
TRSA2048	12,00	0,60	0,06
TRSA3072	36,00	0,90	0,11

Table 5.2: Time required per THS cryptographic operation.

Scheme	Sign (ms)	Aggregate (ms)	Verify (ms)
TBLS256 Crypto-Provider Remote	1,13	9,41	9,13
TRSA1024 Crypto-Provider Remote	7,97	1,60	0,55
TRSA2048 Crypto-Provider Remote	40,45	2,40	0,71
TRSA3072 Crypto-Provider Remote	103,32	3,61	0,94

Table 5.3: Time required per THS cryptographic operation when integrated in a remote cryptoprovider.

Comparing both results, we can see that when we run the primitive inside our crypto-provider, it takes slightly longer to sign transactions, aggregate shares and verify signatures, this is due to the serialization of the private key shares, the serialization of the public keys and the latency between the components when communicating (over TCP). Serialization is an obvious bottleneck when we look at the time it takes to sign a transaction with different TRSA private key sizes.

By creating a crypto-provider decoupled component that allows easy extension and hot-swapping of primitives, we expected some performance degradation, but not as pronounced as the results show. In chapter 6 we present some options to improve performance in future work.

5.6 Prototype Performance With Different Signature Schemes

In this experimental evaluation, our goal is to see how our signer node behaves in isolation under different conditions. For this purpose, we will evaluate the transaction throughput and latency. First, we set the number of signer nodes to 4, with the Byzantine Fault Tolerance of $3f + 1$, and we place these 4 nodes in a geo-distributed environment and in a local environment, increasing the number of concurrent clients sending transactions, as can be seen in Figure 5.6. We then increased the number of distributed signer nodes with a fixed number of 64 concurrent clients to see how more nodes with the same Byzantine Fault Tolerance can affect performance, as shown in Figure 5.7. In both scenarios we will use different threshold schemes, TBLS256 and TRSA 1024, 2048 and 3072 with 2 different aggregation algorithms, the pessimistic and the optimistic. The results in this section have a relative standard deviation of 1–5%.

The first thing we can get from both scenarios represented by Figure 5.6 and Figure 5.7 is that under a no-fault environment, the optimistic algorithm has better performance

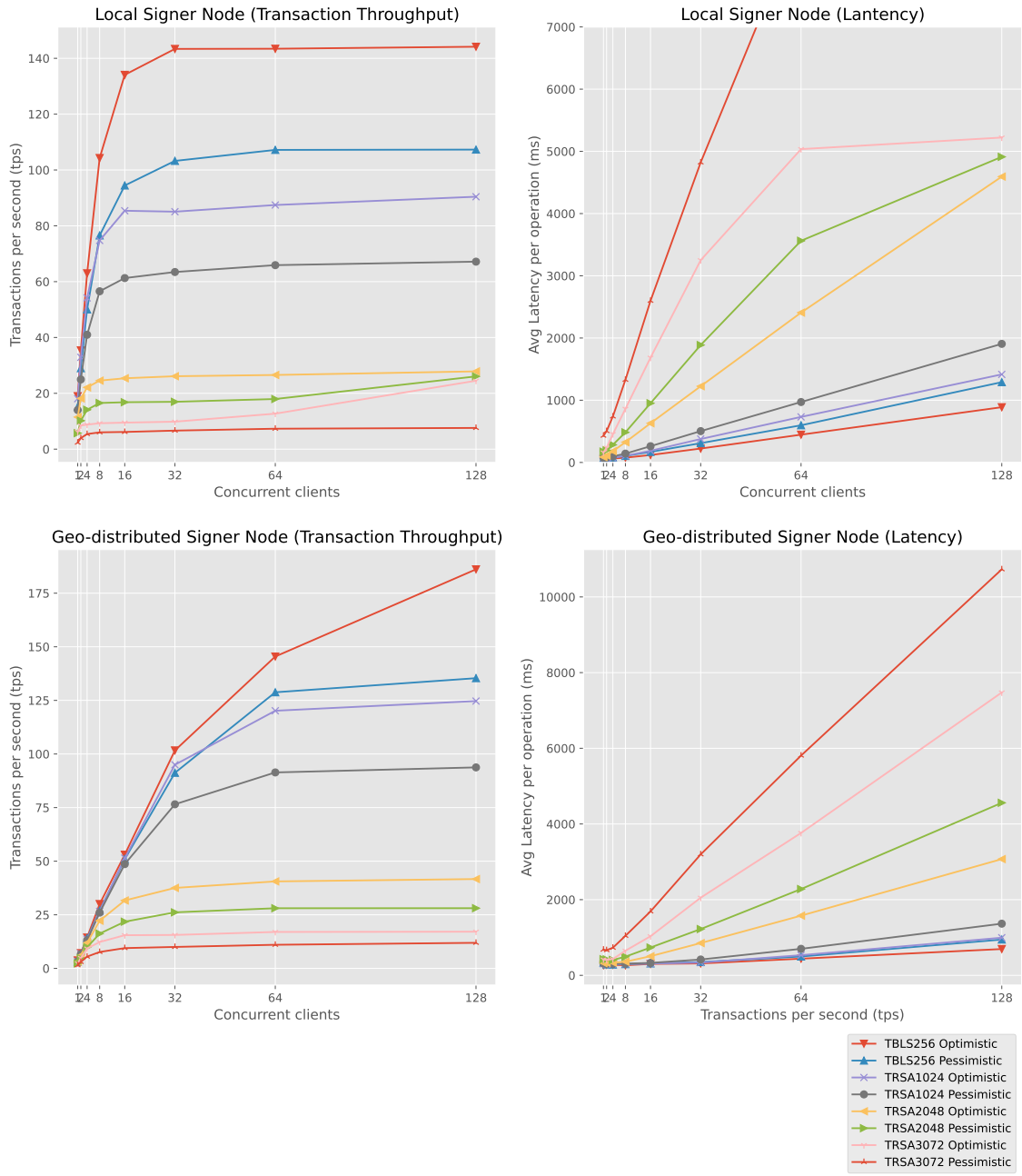


Figure 5.6: Latency and throughput when isolating the 4 signer nodes.

than the pessimistic algorithm. If we recall, a pessimistic algorithm has to check whether all signature shares are correct, and only these are combined to form the final signature. In contrast, the optimistic algorithm simply randomly selects t shares from the n maximum available signature shares, aggregates them first, and only then checks whether they have been aggregated correctly. With this explanation, it is easy to understand why the pessimistic algorithm behaves worse than the optimistic algorithm in a no-fault environment; it has the additional cost of verifying each signature (which will be correct in that environment).

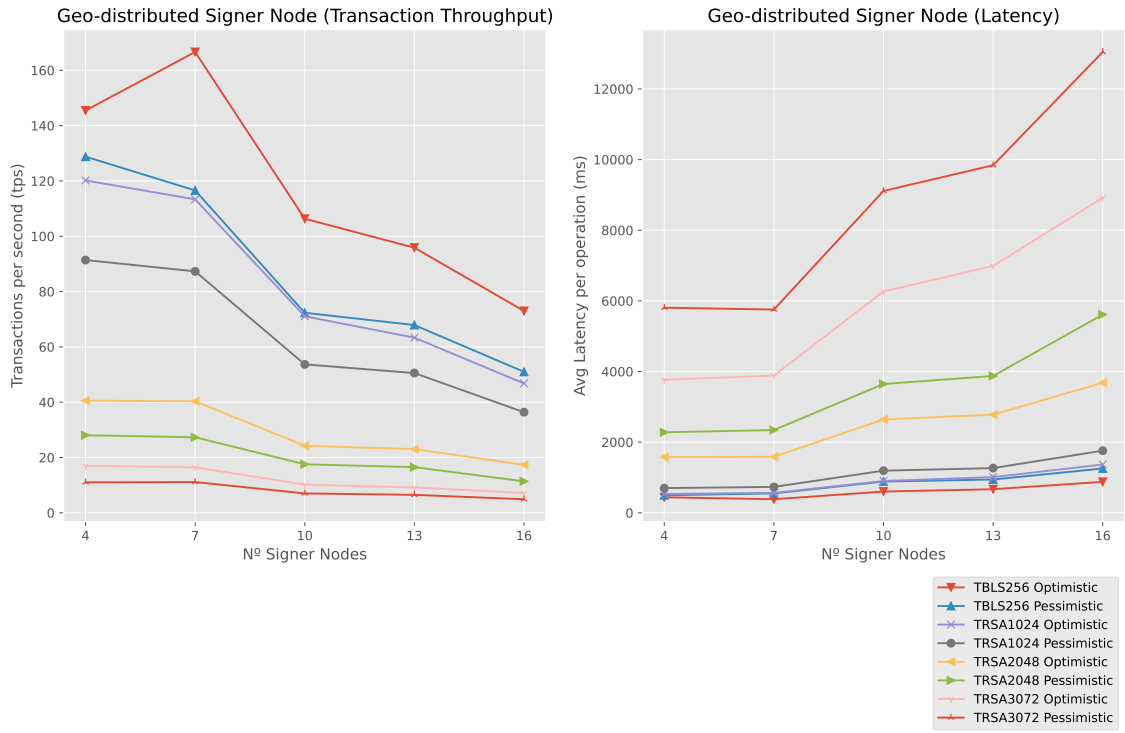


Figure 5.7: Latency and throughput when varying the number of signer nodes

From the first scenario (Figure 5.6), we can also obtain an interesting comparison between a geo-distributed environment and a local environment. In a local environment, the throughput increases rapidly but peaks at 32 concurrent clients in almost all schemes with a maximum of 140 transactions per second for Optimistic TBLS256. In a distributed environment, on the other hand, the transaction throughput increases slowly and peaks at 64 concurrent clients in almost all schemes except Optimistic TBL256 which peaks at 128 concurrent clients with a maximum of 175 transactions per second. This result can again be explained by the pressure of other transactions, in a local environment the nodes have low latency between each other, resulting in more messages being processed per unit time at that moment. In a distributed environment, the system has more time to process all these messages as the latency between nodes is higher, which explains why the transaction throughput is also higher in this case. However, in a geo-distributed environment, more clients are initially needed to achieve the transaction throughput of a local environment.

In the second scenario (,Figure 5.7,) we can see that as we increase the number of signer nodes, the transaction throughput decreases and the average latency per operation increases. This happens because in order to obtain the $3f + 1$ Byzantine Fault Tolerance we need to increase the number of collected signatures until we can start aggregation, e.g., with 16 nodes we need to collect at least 11 correct signature shares, while with 4 nodes we need to collect only 3 correct signature shares. In the same scenario, we can also see that the Optimistic TBLS256 transaction throughput increases when the number

of signer nodes increases from 4 to 7. We initially thought at first this was an outlier, but after repeating the experimental evaluation, we concluded that having more signer nodes to distribute the 64 concurrent clients and wait for an additional 2 signature shares compensates in this case.

5.7 Prototype Performance Inducing Faults

In this experimental evaluation, we want to evaluate how our signer node behaves under an environment where stop and byzantine faults can occur. Both these tests are run in a setup with 16 geo-distributed nodes with Byzantine Fault Tolerance of 3f + 1, which means that at most 5 nodes can fail. For this test, we set the number of concurrent clients to 32 and we will evaluate the performance for TBLS256 and TRSA2048 under two different aggregation algorithms (optimistic and pessimistic). The results in this section have a relative standard deviation of 1–5%.

Figure 5.8 shows the results under a stop fault environment. First, we see an increase in transactions per second and a decrease in latency until we reach 3 stop faults, and then a stabilization. One explanation for this could be related to the number of messages being transmitted. When nodes crash, fewer messages are generated per sign operation, which means that signer nodes have to handle with fewer messages, explaining why the performance increases until we reach 3 node crashes. After 3 crashes, the performance stops improving, and for some cryptographic schemes it even decreases until we reach 5 failures. It is true that the number of messages needed continuously decreases, but then we had more nodes creating shares, and we only need 11 valid shares. Now, when more nodes fail, we have to wait for the slowest nodes to start aggregating shares for being able to create the final signature.

In the previous results shown in Figure 5.8, the optimistic protocols had better performance than the pessimistic ones under the different number of crash faults. However, when we introduce our prototype under Byzantine faults, it is easy to see that the choice between an optimistic and a pessimistic protocol can be relevant for the different environments, depending on the number of Byzantine faults that occur at a given time. This is evident from the analysis of the plot in the Figure 5.9. For sporadic Byzantine faults (0 to 2), we can see that a scheme with optimistic aggregation behaves better than the pessimistic one. However, for a higher number of Byzantine faults (3 to 5), we can see that the pessimistic algorithm behaves better than the optimistic one.

This result can be explained by the fact that the optimistic protocol assumes that every combination of t shares is a correct combination, aggregates them, and only verifies whether the final signature is indeed well constructed. To compute the number of possible combinations, we use the formula $C_t^n = \frac{n!}{t!(n-t)!}$. Instantiating this for our test scenario, we have $t = 5$ and $n = n$, resulting in $C_5^{16} = 4368$ possible combinations, where as the number of invalid shares increases, the number of valid combinations in the set of size

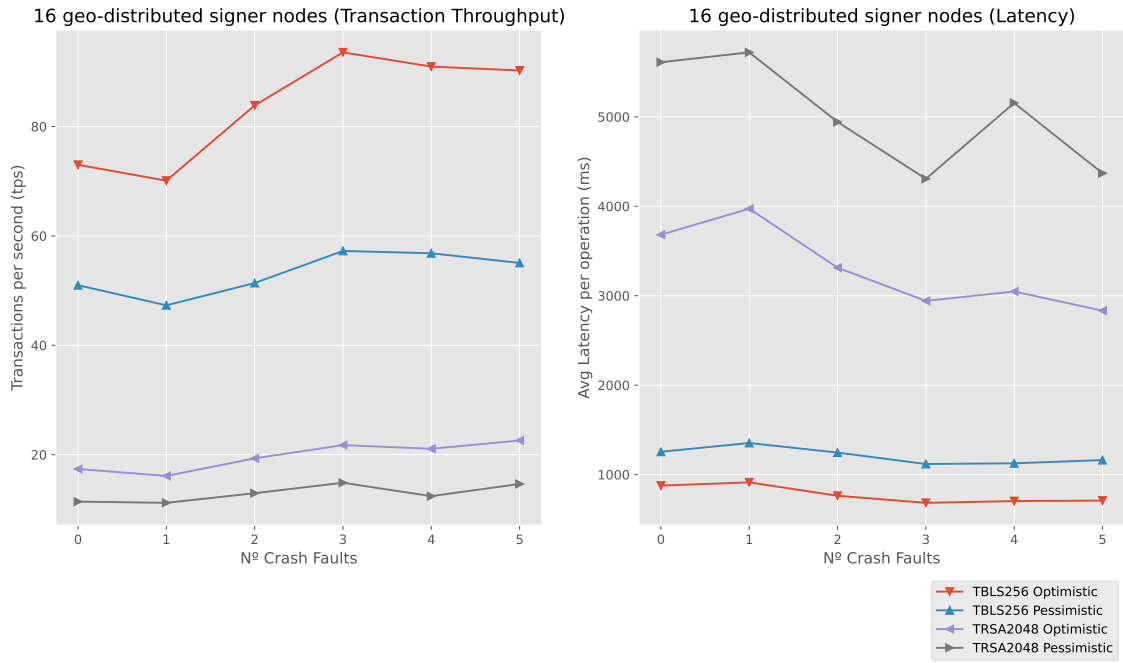


Figure 5.8: Latency and throughput when varying the number of stop faults.

4368 decreases. On the other hand, the pessimistic protocol needs to verify at most 16 signatures.

These results show that a combination of optimistic and pessimistic algorithms is possible, for example we could run both aggregation algorithms at the same time and only use the first response from our cryptoprovider, this approach would have other performance penalties, but if resources are not an issue this could be a solution. Another possible solution could be to create a dynamic algorithm that detects how many Byzantine faults occur and selects the appropriate aggregation algorithm according to the system metrics, e.g., we do not expect many Byzantine signature shares in a permissioned environment, while we might get more in a permissionless environment.

When we designed and implemented our cryptoprovider to be decoupled from the signer node (language independent), we knew that we would incur additional costs when it came to Byzantine environments. Both protocols, the pessimistic and the optimistic, are designed in a synchronous way, which means that every time the cryptoprovider receives new shares (for $shares > t$), it runs the aggregation protocol from the beginning, which means that the algorithms in the cryptoprovider have no state and do not know which combinations or which shares they have already validated. This was an implementation option to ensure a high degree of decoupling between the cryptoprovider and the signer node.

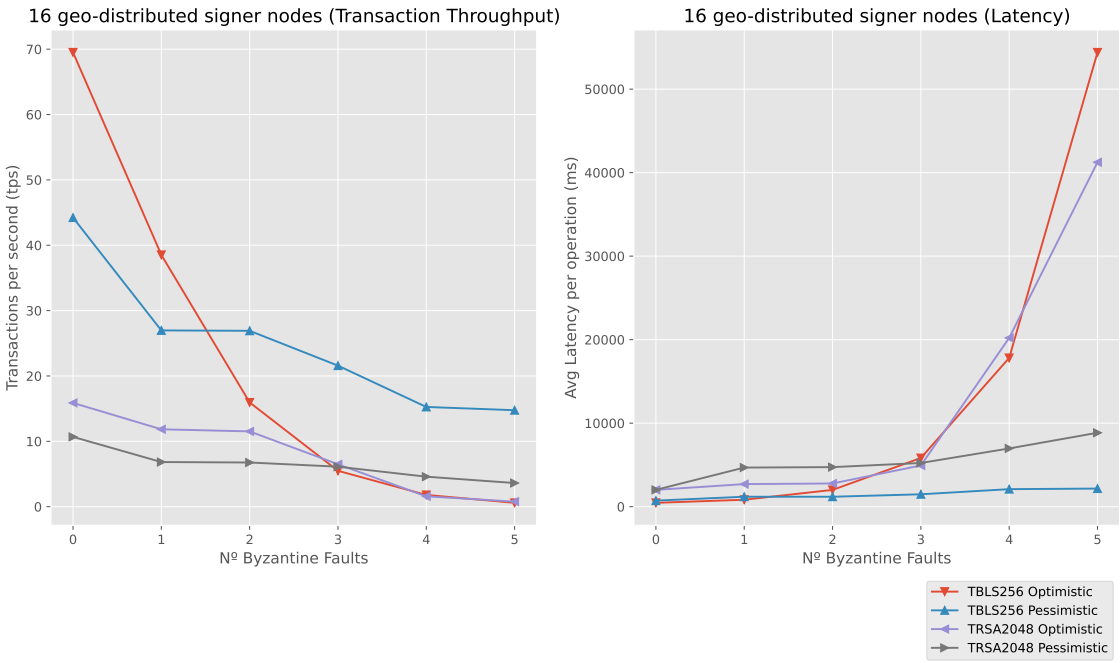


Figure 5.9: Latency and throughput when varying the number of byzantine faults.

5.8 Permissionless Group Formation

In this experimental evaluation, our goal is to analyze the cost of deploying shares in a permissionless environment. For this experimental evaluation, we deployed 16 geo-distributed nodes in the 4 different datacenters, where in the first test we randomly select signer nodes and deploy a share in each one. This test is performed with a single client, choosing group sizes of 4, 7, 10, 13, and 16 signer nodes to deploy the shares in two modes: concurrent or sequential. Next, we evaluate the cost of arranging these 16 nodes by distance to determine the cost of selecting the closest nodes instead of randomly. To do this, we sort different set sizes of signer nodes (e.g., 4,7,10,13,16) by distance using the ICMP protocol present in our benchmark client.

From the figure 5.10, it can be seen that sequentially installing shares in each signer node is very costly compared to installing the key shares concurrently. This is true for any scheme used (TBLS and TRSA) with any key size. We expected that installing shares for a tRSA 3072 would be more expensive than installing shares for TBLS 256 due to the key size, however the latency between the client and each distributed signer node is the key factor in the key installation latency time. We must note that the latency in the figure 5.10 excludes key generation time.

From the figure 5.11, it is easy to see that parallel sorting of the group can be performed in constant time for the number of signer nodes we evaluated. Sequential sorting leads to a constant increase in the time required. As the previous results show, the key factor in sorting nodes by distance is the latency between the client and all the nodes to be sorted. Note that this process must be performed once for each new subset of signer

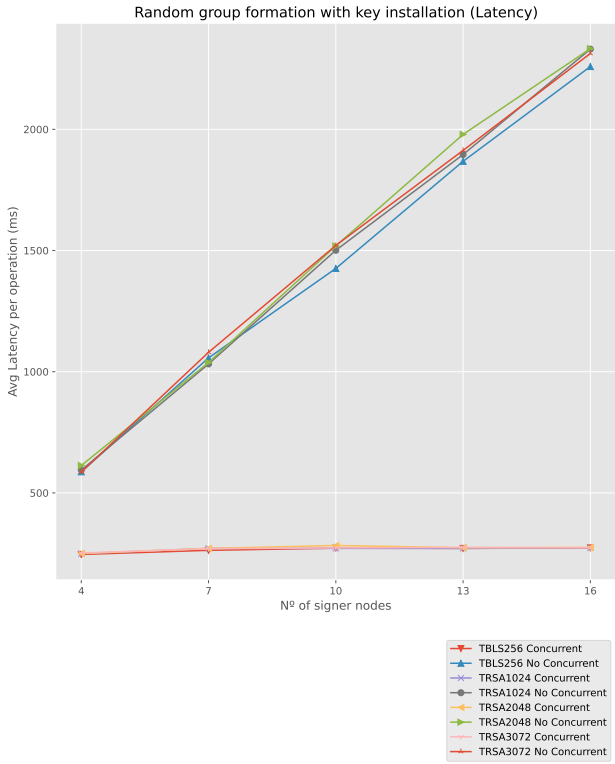


Figure 5.10: Shares installation latency, when varying the number of signer nodes.

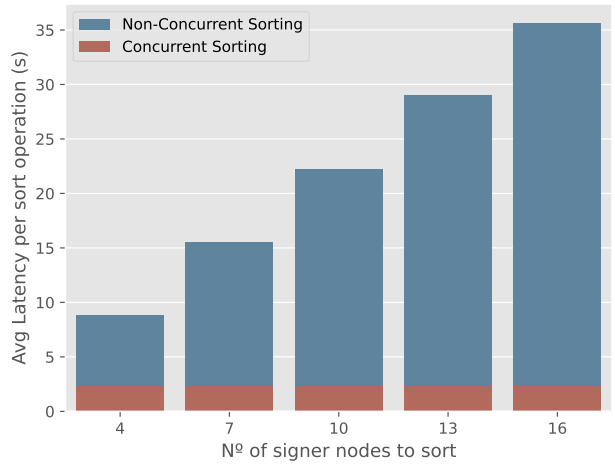


Figure 5.11: Latency per sort operation when sorting different set sizes of signer nodes.

nodes, since signer nodes do not change their location. However, if the client does not care at all where the nodes are located, we do not need to sort the nodes by distance, which imposes an additional cost on the client later when creating the group signature.

From all the results of this experimental evaluation, we can conclude that for some workloads it makes sense to install the shares when it is necessary to submit the transaction. However, for more intensive workloads, it is recommended to generate the shares offline and deploy them in advance.

5.9 Summary

In this chapter we presented different experimental evaluations over our prototype. We conducted tests in both blockchain platforms (Algorand and Sawtooth), to the integration between our prototype and the blockchain platform, we have also performed isolated experimental evaluations to determine the cost of each sub-component. All of this tests where intended to answer the questions formed in the beginning of the chapter. We also analysed the experimental evaluations summarized bellow:

- A baseline performance (, average latency and transaction throughput,) when running Hyperledger Sawtooth and Algorand distributed across data centers and in the same data center, comparing both consensus protocols (PBFT and PPoS).
- Cost of threshold signature schemes in raw mode, with our crypto provider and a comparison of the two cases.
- The average latency and transaction throughput when integrating our prototype with Algorand and Sawtooth in a geo-distributed environment with different signature schemes: threshold bls and threshold RSA with different key sizes.
- The average latency and transaction throughput when isolating our prototype from the blockchain, georeplicating across data centers and in the same data center.
- The impact and ability of our prototype to withstand crash and Byzantine faults, comparing the performance of different THS scheme, varying key sizes.
- The cost of forming groups of different sizes and installing the shares in member.
- The cost of sorting signer nodes by distance from the client.

Given our results, we can conclude that our prototype of the signer node is capable of scaling and can be deployed agnostically on any blockchain platform or other type of distributed system that wishes to decentralize digital signatures. We believe that threshold signatures can achieve high throughput when formed by small groups of participants. Looking at our results, we saw that in the case of Algorand, the limiting factor was not our prototype, but the blockchain platform itself. In the case of Sawtooth and our prototype, both support batch transactions. Although we did not show the results, we believe this is a viable option to improve transaction throughput in this scenario.

Our prototype even supports key installation for a permissionless protocol. This key installation incurs some overhead, but we can do it in parallel, which reduces latency and increases the security of the threshold signature since we keep re-keying, minimizing the key exposer and reducing the time an attacker has to compromise t signer nodes.

As a final remark, we recommend the use of witness groups with sizes from 4 to 16 nodes, with threshold BLS256. Using threshold RSA with 1024 bits can be insecure, and keys with sizes larger than 1024 can cause huge overheads in group signature formation.

Chapter 6

Conclusion and Final Remarks

This chapter draws some conclusions about the research topic addressed in this dissertation and finally makes some suggestions to point out some open issues and limitations of the system model and the prototype itself that should be addressed in future work.

6.1 Conclusion

The research for this dissertation has allowed us to understand more about how to remove the single point of failure through the decentralization of digital signatures present in current blockchain implementations. We explored different types of group signatures, their properties, and costs to better understand how to harness their potential to decentralize blockchains. We also studied how distributed ledger technology and smart contracts work and how they are implemented in different blockchain platforms. We made a comparison between them to better illustrate their limitations and the impact of each decision made to building such a platform. We have also explored different approaches by different authors to decentralize the single point of failure in current blockchain platforms.

Our proposed system model aims to eliminate the single point of failure of traditional signatures and create a new Byzantine-Fault tolerance layer that can be applied to any blockchain. Our intention with this model was to create a new level of decentralization for blockchain platforms by implementing in the design some system goals, such as, platform agnostic, auditable, resilient, fair, dynamic and reconfigurable.

Based on our system model, we built a fully functional prototype that is platform agnostic and can be deployed against different blockchain platforms, with a modular and portable cryptographic provider that supports cryptographic operations that can be expressed and executed by smart contracts. As a proof-of-concept, we decided to apply our prototype to two different blockchain nodes that we analysed in the related work: i)

Algorand, because it is permissionless and uses the PPoS; ii) and Hyperledger Sawtooth, because it is permissioned and uses the PBFT. Our implementation takes into account the heterogeneity of platforms and smart contract languages, so we also implemented the possibility to integrate a different smart contract engine into our prototype.

We have performed experimental evaluations to assess the performance of our prototype in different scenarios, such as transaction throughput and latency when varying the number of clients and group signature scheme and key sizes, georeplication of our solution in different data centers, inducing crash and Byzantine faults, and the cost of forming groups of different sizes. From the results, we can conclude that our prototype is scalable and can be agnostically deployed on any blockchain platform or other types of distributed systems that wishes to decentralize digital signatures.

In summary, we have achieved the objectives and contributions we set for this dissertation by conceiving a system model and a fully functional prototype capable of decentralizing the single point of failure (digital signatures) by providing a new layer of resilience and integrating with any blockchain platform.

6.2 Future Work

We have addressed our proposed goals and contributions that we originally laid to this dissertation, furthermore we were able to develop a fully functional prototype in relation to our system model and goals, however there is still room for some improvements in the future:

- Regarding our crypto provider, there is too much latency introduced by our pluggable and factory-based architecture. This aspect could be improved by revising the way the different cryptographic materials (private key share, public key and signatures) are marshalled and unmarshalled and, if the crypto-provider is on the same machine as the signer node, using Linux sockets instead of TCP to remove the overhead caused by TCP handshake and IP routing. Our prototype supports the latter, but no experimental evaluations were performed to prove whether Linux sockets would reduce latency.

- Regarding our signer node, we could develop a smarter algorithm that detects if a peer is constantly behaving Byzantine, and provide ways to ban him from the trusted peers to improve performance by not validating and accepting signature shares coming from that node. In the same way, the smarter algorithm could detect the degree of Byzantine members and automatically choose which aggregation algorithm to use at a given time to reduce the time required to generate a group signature. We could also dynamically choose whether the signature shares produced by the witnesses should be broadcasted to all witnesses or directly forwarded to the signer node responsible for aggregating the signature shares into the group signature.

Bibliography

[1] *Algorand Dev*. URL: https://developer.algorand.org/docs/features/asc1/ (visited on 11/18/2020).

[2] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino. "SoK: Unraveling bitcoin smart contracts." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10804 LNCS (2018), pp. 217–242. ISSN: 16113349. DOI: 10.1007/978-3-319-89722-6_9.

[3] M. Bellare and G. Neven. "Identity-Based Multi-signatures from RSA." In: *Topics in Cryptology – CT-RSA 2007*. Ed. by M. Abe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 145–162. ISBN: 978-3-540-69328-4.

[4] *Bitaps*. URL: https://bitaps.com (visited on 09/20/2020).

[5] D. Boneh, B. Lynn, and H. Shacham. "Short signatures from the weil pairing." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2001. ISBN: 3540429875. DOI: 10.1007/3-540-45682-1_30.

[6] P. Bright. *Another fraudulent certificate raises the same old questions about certificate authorities*. 2011. URL: https://arstechnica.com/information-technology/2011/08/earlier-this-year-an-iranian/ (visited on 09/06/2020).

[7] P. Bright. *Independent Iranian Hacker Claims Responsibility for Comodo Hack*. 2017. URL: https://www.wired.com/2011/03/comodo-hack/ (visited on 09/06/2020).

[8] F. Casino, T. K. Dasaklis, and C. Patsakis. "A systematic literature review of blockchain-based applications: Current status, classification and open issues." In: *Telematics and Informatics* 36 (2019), pp. 55–81. ISSN: 0736-5853. DOI: https://doi.org/10.1016/j.tele.2018.11.006. URL: http://www.sciencedirect.com/science/article/pii/S0736585318306324.

[9] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance." In: *Proceedings of the Symposium on Operating System Design and Implementation* (1999). ISSN: 07342071. DOI: 10.1145/571637.571640. arXiv: arXiv:1203.6049v1.

[10] D. Chaum and E. Van Heyst. "Group signatures." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 547 LNCS.iii (1991), pp. 257–265. ISSN: 16113349. DOI: 10.1007/3-540-46416-6_22.

[11] K. Christidis and M. Devetsikiotis. *Blockchains and Smart Contracts for the Internet of Things*. 2016. DOI: 10.1109/ACCESS.2016.2566339.

[12] J. Cieplak and S. Leefatt. "Smart Contracts: A Smart Way To Automate Performance." In: *Georgetown Law and Technology Review* 1.2 (2017), pp. 417–427. URL: http://www.treasurer.ca.gov/cdiac/reports/rateswap04-12.pdf.

[13] G Coulouris, J Dollimore, and T Kindberg. "Distributed Systems: Concepts and Design." In: 3rd. Addison-Wesley, 2001, pp. 451–462. ISBN: 9780201619188.

[14] I. Damgård, T. P. Jakobsen, J. B. Nielsen, J. I. Pagter, and M. B. Østergaard. "Fast threshold ecdsa with honest majority." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2020. ISBN: 9783030579890. DOI: 10.1007/978-3-030-57990-6_19.

[15] Dedis. *DEDIS Advanced Crypto Library for Go*. URL: https://github.com/dedis/kyber (visited on 10/07/2020).

[16] Y. Desmedt and Y. Frankel. "Threshold cryptosystems." In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by G. Brassard. New York, NY: Springer New York, 1990, pp. 307–315. ISBN: 978-0-387-34805-6.

[17] Q. DuPont. "Experiments in algorithmic governance: A history and ethnography of "The DAO,"a failed decentralized autonomous organization." In: *Bitcoin and Beyond: Cryptocurrencies, Blockchains, and Global Governance* January (2017), pp. 157–177. DOI: 10.4324/9781315211909.

[18] C. Dwork and M. Naor. "Pricing via Processing or Combatting Junk Mail." In: *Advances in Cryptology — CRYPTO' 92*. Ed. by E. F. Brickell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–147. ISBN: 978-3-540-48071-6.

[19] Enzoh. *Boneh–Lynn–Shacham Signature Scheme*. URL: https://github.com/enzoh/go-bls (visited on 10/06/2020).

[20] Ethereum. *Solidity v0.7.4 Documentation*. URL: https://solidity.readthedocs.io/en/v0.7.4/ (visited on 11/06/2020).

[21] C. Evans, C. Palmer, and R. Sleevi. *Public Key Pinning Extension for HTTP*. Tech. rep. 2015. DOI: 10.17487/RFC7469. URL: https://www.rfc-editor.org/info/rfc7469.

[23] M. S. Ferdous, M. J. M. Chowdhury, M. A. Hoque, and A. Colman. *Blockchain Consensus Algorithms: A Survey*. 2020. arXiv: 2001.07091 [cs.DC].

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process." In: *J. ACM* 32.2 (1985), pp. 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121. URL: https://doi.org/10.1145/3149.214121.

[25] G. Francisco. "Bringing Order into Things." Master of Science. FCT-UNL, 2018.

[26] R. Gennaro and S. Goldfeder. "Fast multiparty threshold ECDSA with fast trustless setup." In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2018. ISBN: 9781450356930. DOI: 10.1145/3243734.3243859.

[27] R. Gennaro, S. Goldfeder, and A. Narayanan. "Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security." In: *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Proceedings*. Ed. by M. Manulis, S. Schneider, and A.-R. Sadeghi. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Germany: Springer Verlag, 2016, pp. 156–174. ISBN: 9783319395548. DOI: 10.1007/978-3-319-39555-5_9.

[28] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. "Robust threshold DSS signatures." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 1996. ISBN: 354061186X. DOI: 10.1007/3-540-68339-9_31.

[29] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems." In: *Journal of Cryptology* 20.1 (2007), pp. 51–83. ISSN: 1432-1378. DOI: 10.1007/s00145-006-0347-3. URL: https://doi.org/10.1007/s00145-006-0347-3.

[30] S. Goldfeder, J. Bonneau, J. A. Kroll, H. Kalodner, E. W. Felten, R. Gennaro, and A. Narayanan. "Securing Bitcoin wallets via a new DSA / ECDSA threshold signature scheme." In: *International Conference on Applied Cryptography and Network Security. Springer, Cham.* (2016), pp. 156–174.

[31] L. Goodman. "Tezos Position Paper." In: (2014), pp. 1–33.

[32] H-online.com. *Trustwave issued a man-in-the-middle certificate*. 2012. URL: http://www.h-online.com/security/news/item/Trustwave-issued-a-man-in-the-middle-certificate-1429982.html (visited on 09/06/2020).

[33] S. Haber and W. S. Stornetta. "How to Time-Stamp a Digital Document." In: *J. Cryptol.* 3.2 (1991), pp. 99–111. ISSN: 0933-2790. DOI: 10.1007/BF00196791. URL: https://doi.org/10.1007/BF00196791.

[34] F. Information and P. Standards. "Digital Signature Standard." In: *Safeguarding Critical E-Documents* July (2015), pp. 221–221. DOI: 10.1002/9781119204909.app1.

[35] M. Jansen, F. Hdhili, R. Gouiaa, and Z. Qasem. "Do smart contract languages need to be turing complete?" In: *Advances in Intelligent Systems and Computing* 1010.March (2020), pp. 19–26. ISSN: 21945365. DOI: 10.1007/978-3-030-23813-1_3.

[36] F. Knirsch, A. Unterweger, and D. Engel. "Implementing a blockchain from scratch: why, how, and what we learned." In: *Eurasip Journal on Information Security* 2019.1 (2019). ISSN: 2510523X. DOI: 10.1186/s13635-019-0085-3.

[37] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, B. Ford, E. Kokoris-Kogias, and B. F. Epfl. "This paper is included in the Proceedings of the 25th USENIX Security Symposium Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Sig." In: (2016). arXiv: 1602.06997. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias.

[38] E. Kokoris-Kogias, E. C. Alp, L. Gasser, P. Jovanovic, E. Syta, and B. Ford. "Calypso: Private data management for decentralized ledgers." In: *Proceedings of the VLDB Endowment* 14.4 (2020), pp. 586–599. ISSN: 21508097. DOI: 10.14778/3436905.3436917.

[39] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger." In: *IACR Cryptology ePrint Archive* (2017), p. 406. URL: https://eprint.iacr.org/2017/406.

[40] L. Lamport. "Paxos Made Simple." In: *ACM SIGACT News* (2001). ISSN: 01635700.

[41] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem." In: *ACM Transactions on Programming Languages and Systems* (1982), pp. 382–401. URL: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/.

[42] W. Lloyd and E. Berlekamp. *Error Correction for Algebraic Block Codes. US Patent 4,633,470*. 1986.

[43] S. Micali. *Algorand's Smart Contract Architecture*. URL: https://www.algorand.com/resources/blog/algorand-smart-contract-architecture (visited on 11/18/2020).

[44] E. Mik. "Smart Contracts: A Requiem." In: *SSRN Electronic Journal* December (2019), pp. 1–22. ISSN: 1556-5068. DOI: 10.2139/ssrn.3499998.

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

[46] Niclabs. *Golang Threshold Cryptography Library - RSA implementation*. URL: https://github.com/niclabs/tcrsa (visited on 10/05/2020).

[47] Niclabs. *TCECDSA - Threshold Cryptography Eliptic Curve Digital Signature Algorithm*. (Visited on 11/10/2020).

[48] K Ohta and T Okamoto. "Multi-Signature Schemes Secure against Active Insider Attacks (Special Section on Cryptography and Information Security)." In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 82 (1999), pp. 21–31.

[49] T. P. Pedersen. "A threshold cryptosystem without a trusted party." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 547 LNCS (1991), pp. 522–526. ISSN: 16113349. DOI: 10.1007/3-540-46416-6_47.

[50] QuantumMechanic. *Proof of stake instead of proof of work*. 2011. URL: https://bitcointalk.org/index.php?topic=27787.0 (visited on 09/19/2020).

[51] R. L. Rivest, A Shamir, and L Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[52] R. L. Rivest, A. Shamir, and Y. Tauman. "How to leak a secret." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2001. ISBN: 3540429875. DOI: 10.1007/3-540-45682-1_32.

[53] A. Shamir. "How to Share a Secret." In: *Commun. ACM* 22.11 (1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[54] V. Shoup. "Practical threshold signatures." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2000. ISBN: 9783540675174. DOI: 10.1007/3-540-45539-6_15.

[55] J. Sousa, A. Bessani, and M. Vukolic. "A byzantine Fault-Tolerant ordering service for the hyperledger fabric blockchain platform." In: *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018*. 2018. ISBN: 9781538655955. DOI: 10.1109/DSN.2018.00018. arXiv: 1709.06921.

[56] C. Stathakopoulou and C. Cachin. "Research Report: Threshold Signatures for Blockchain Systems." In: (2017).

[57] D. R. Stinson and R. Strobl. "Provably secure distributed schnorr signatures and a (T, n) threshold scheme for implicit certificates." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2119 (2001), pp. 417–434. ISSN: 16113349. DOI: 10.1007/3-540-47719-5_33.

[58] 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: *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016* (2016), pp. 526–545. DOI: 10.1109/SP.2016.38. arXiv: 1503.08768.

[59] N. Szabo. "Smart Contracts: Building Blocks for Digital Free Markets." In: *Extropy Journal of Transhuman Thought* (1996). ISSN: 1527-7755.

[60] E. Vanderburg. *The Threat of Rogue Certificate Authorities: A Certified Lack of Confidence*. 2018. URL: https://www.tcdi.com/the-threat-of-rogue-certificate-authorities/ (visited on 09/06/2020).

[61] B. Wiki. *Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/ (visited on 11/07/2020).

[62] G. Wood. "Ethereum: a secure decentralised generalised transaction ledger." In: *Ethereum Project Yellow Paper* (2014), pp. 1–32. ISSN: 1098-6596. arXiv: arXiv:1011.1669v3.

[63] K. Zīle and R. Strazdiņa. "Blockchain Use Cases and Their Feasibility." In: *Applied Computer Systems* 23.1 (2018), pp. 12–20. ISSN: 2255-8691. DOI: 10.2478/acss-2018-0002.