UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Building a Confidential Cryptocurrency

Gabriel Jan Cholewinski Freitas

Mestrado em Segurança Informática

Dissertação orientada por:

Professor Doutor Bernardo Luís Da Silva Ferreira

2022

Acknowledgements

First of all, I want to thank my Family for their support throughout my academic path, especially my Parents, my Heroes, Álvaro and Magdalena, who always believed in me, my brothers, Filipe, Larissa, Maria, Marco, my uncles, and my grandparents who were also always by my side. A huge thank you to my girlfriend Patricia for being the most special girl I have ever met, for always being on my side, pushing me, believing in me, and contributing to my future through love, encouragement, and patience. Additionally, I want to express my appreciation to all of my friends, from my primary school to the friends that the University gave to me. Thanks to all my colleagues on the Brazilian Jiu-Jitsu team, who despite all the adversities, always had my back, always pushed me hard in training, and never gave up on me.

I want to give my greatest thanks to Professor Bernardo Ferreira, who made it possible for me to do this thesis, for giving me the freedom to follow the project in an autonomous way, and for his support and motivation. A special thanks to Robin Vassantlal, for his help, for his patience, for his time, for his motivation, and support that allowed me to carry out the project in a more appropriate way. The realization of this work and the making of this journey would not be possible without these people by my side.

Finally, thank you Faculdade de Ciências da Universidade de Lisboa for these years and see you soon.

”Success is not final, failure is not fatal: it is the courage to continue that counts.”

Winston Churchill

Resumo

Atualmente, vivemos num mundo mais digital onde a Internet faz parte do nosso dia a dia, sendo que usamos sistemas, aplicações e tecnologias criadas sobre a mesma. Não obstante, o mundo é alvo de uma evolução tecnológica constante, onde novas tecnologias são criadas e mudam a perspetiva como damos uso a certas aplicações, como é o caso das blockchains.

As Blockchains são uma tecnologia emergente onde a confiança é retirada das entidades centrais e, em vez disso, descentralizada entre uma rede de entidades iguais. A aplicação clássica para construir sobre uma blockchain é uma moeda criptográfica, o que permite aos clientes fazer pagamentos e transferir dinheiro entre si. No entanto, as blockchains geralmente proporcionam um certo grau de privacidade. Trabalhos recentes provaram que os dados de localização pseudónimos armazenados numa blockchain podem tornar vulgar a identificação de um indivíduo. Esta é uma grande preocupação para as blockchains porque os dados persistidos numa blockchain estão abertos ao escrutínio de todos, mesmo daqueles que procuram explorar os dados para obter ganhos financeiros. Além disso, o registo imutável das blockchains agrava este problema. Uma vez persistido um dado individual dentro de uma solução que implemente uma tecnologia como a blockchain, todas as transações que transportam os dados desse indivíduo, por exemplo, dados financeiros, serão expostas como associadas a essa pessoa.

Até à data, têm surgido abordagens distintas para enfrentar estas preocupações. Como resposta a este problema, foram criadas soluções para proteger dados financeiros, como moedas privadas que cobrem o anonimato. Algumas delas oferecem total privacidade, a nível de identidade e de transação, e outras propõem uma delas. Contudo, apesar dos avanços recentes relevantes, as moedas privadas existentes não oferecem a quantidade de privacidade adequada e fazem um compromisso crítico entre segurança, confidencialidade e privacidade.

Hoje em dia, novas tecnologias que fornecem confidencialidade sobre dados armazenados e dados transacionais têm estado em desenvolvimento e têm sido tidas em conta. Estas tecnologias permitem que as moedas criptográficas armazenem a confidencialidade dos dados considerados como garantidos e permitem o anonimato tanto sobre as identidades envolvidas numa transação, como sobre a confidencialidade dos dados da transação. Tecnologias de exemplo reconhecidas são o COBRA, um esquema de ”secret sharing”, e o sistema Paillier, um esquema de cifra homomórfica.

Neste trabalho, estudamos em profundidade diferentes moedas privadas e novas tecnologias, com a intenção de conhecer e explorar a sua aplicação ao problema de oferecer confidencialidade sobre transações e persistência de dados. Com esta informação, conseguimos identificar os pontos fortes e os pontos fracos

das criptomoedas privadas existentes, e ao mesmo tempo perceber os diferentes algoritmos e protocolos implementadas pelas mesmas. Além disso, estudámos tecnologias adjacentes que implementam técnicas como a replicação de máquina de estados, secret sharing e cifras homomórficas.

Replicação Máquina de Estados tolerante a faltas bizantinas é uma técnica clássica para implementar sistemas tolerantes a faltas distribuídas mesmo quando uma fração das réplicas está sujeita a faltas arbitrárias. Neste tipo de sistema as réplicas executam um protocolo de consenso para concordarem sobre a sequência dos comandos que executam.

Secret sharing é um tipo de técnica que protege a confidencialidade dos dados quando os mesmos estão em repouso, através da divisão dos dados em shares (i.e., pedaços), de tal forma que os dados só podem ser recuperados através da combinação de um número mínimo de shares.

Cifra Homomórfica é uma forma de cifra que oferece a possibilidade a certos tipos de operações serem realizadas sob dados cifrados, a fim de se obter um resultado cifrado, que é de igual modo fazer a mesma operação sob os dados em claro, obtendo o mesmo resultado cifrado.

Nesta dissertação, propomos Nomad, uma moeda criptográfica privada confidencial. Nomad é aplicada através da construção de um sistema de pagamentos confidencial descentralizado, onde os dados financeiros, ou seja, os montantes de transferência (ou seja, moedas) podem ser armazenados com privacidade, reforçando a confidencialidade, o anonimato, a não rastreabilidade, e a integridade como as suas principais prioridades. Para alcançar estas propriedades, Nomad é implementada em cima de COBRA, uma Framework de Replicação Máquina de Estados que preserva a privacidade, que divide os dados em diferentes partes aleatórias para alcançar a confidencialidade nos dados armazenados, e implementa Paillier, um esquema de cifra homomórfica que permite operações em cima de dados cifrados, permitindo a confidencialidade nos dados transacionais.

Nomad segue um modelo do tipo permissional, onde clientes e réplicas que necessitam de entrar no sistema, precisam de autorização prévia. Ao contrário de sistemas práticos anteriores, com Nomad a ser implementada seguindo uma abordagem de replicação máquina de estados, o nível de privacidade já é maior comparando com a maioria de outras criptomoedas que têm como objetivo promover a confidencialidade. Isto porque este tipo de moedas são muitas vezes implementadas sob blockchains públicas, em que qualquer entidade tem acesso aos dados da blockchain, podendo ver os mesmos, ou seja, quais transações foram executadas, apesar de serem privados.

A inovação que apresentamos em Nomad, é o facto de estar implementada sob um modelo de secret sharing e um algoritmo de cifra homomórfica modificado. No caso do secret sharing, este tipo de técnica é de extrema importância para proteger a informação guardada num sistema replicado, porque nos sistemas tradicionais que seguem uma abordagem de replicação máquina de estados, existem muitas réplicas com os mesmos dados, o que aumenta a superfície de ataque de um agente malicioso caso este queira roubar os dados. O secret sharing permite que Nomad persista em cada réplica, os mesmos dados, mas divididos em partes diferentes. No caso da cifra homomórfica, permitimos que Nomad seja o mais confidencial possível, de modo a que todas as operações que envolvam o cálculo de novas moedas ou mesmo a verificação de que determinadas moedas podem ser gastas por um cliente, seja tudo cifrado. Ao implementarmos um algoritmo de cifra homomórfica sob Nomad, qualquer entidade pode ver ou mesmo ter acesso aos dados cifrados, no qual nada vai fazer com os mesmos se não tiver a chave correta para

decifrar os dados.

Nomad é a primeira criptomoeda privada, tendo como base um sistema de pagamentos decentralizado confidencial, implementada seguindo uma abordagem de replicação máquina de estados, que segue um modelo permissional, com anonimidade, confidencialidade, privacidade, e integridade. Comparando com sistemas anteriores, Nomad tem todas as propriedades que queremos para um tipo de sistema que tenha como objetivo ter confidencialidade e privacidade como principais requisitos.

Como resultado do trabalho desenvolvido, qualquer sistema bancário pode adotar esta solução e ser capaz de utilizar a moeda da Nomad. Este projeto também fornece uma visão geral para analisar se a Nomad é segura e escalável. Nesta dissertação avaliámos experimentalmente a solução desenvolvida com o intuito de medir o impacto de integração de um sistema totalmente confidencial sob um sistema de replicação máquina de estados. Os resultados obtidos mostram que o facto de termos implementado Nomad dando à vertente de confidencialidade uma maior atenção do que à vertente de performance, que o sistema é mais lento comparando com sistemas que seguem uma abordam de replicação máquina de estados. Pelo contrário, comparando Nomad com outras criptomoedas privadas, a mesma apresenta um desempenho positivo. Para além disso, a solução foi capaz de alcançar os objetivos esperados, que visa a implementação de um sistema de pagamentos decentralizado confidencial, tendo como base tecnologias inovadoras e únicas.

Palavras Chave: Blockchain, Moeda criptográfica, Privacidade, Confidencialidade, Replicação Máquina de Estados.

Abstract

Blockchains are an emerging technology where trust is removed from central entities and instead decentralized among a network of peers. The classical application to build on top of blockchains is a cryptocurrency, allowing clients to make payments and transfer money between each other. However, blockchains usually provide a certain degree of privacy. Recent works have proven that pseudonymous location data stored in a blockchain can make the identification of an individual trivial. This is a big concern for blockchains because blockchain data is open for scrutiny by everyone, even those who are looking to exploit data for financial gain. In addition, the immutable record of the blockchain aggravates this problem. Once an individual data is persisted within a blockchain solution, all of the transactions carrying that individual's data, for example, financial data, will be exposed as linked to that person.

To this day distinct approaches to tackle these concerns have emerged. Naive solutions to protect financial data, like private currencies covering anonymity have been created as an answer to this problem. Some of them offer full privacy, at the identity level and transaction level, and others offer one of them. However, despite relevant recent advances, existing private currencies don't offer the amount of proper privacy and make a critical trade-off between security, confidentiality, and privacy.

Nowadays, new technologies providing confidentiality on top of stored data and transaction data have been under development and taken into account. These technologies allow cryptocurrencies to store data taken confidentiality as guaranteed and allow anonymity on both identities involved in a transaction, as well as confidentiality regarding transaction data. Recognized example technologies are COBRA, a secret sharing scheme, and Paillier cryptosystem, a homomorphic encryption scheme.

In this work, we study different private currencies and new technologies in depth, with the intent to know and explore their application to the problem of offering confidentiality on transaction and persisted data. We propose Nomad, a confidential private cryptocurrency. Nomad is enforced by building a confidential decentralized payment system, where financial data, that is, transfer amounts (i.e., coins) can be stored with privacy, enhancing confidentiality, anonymity, untraceability, and integrity as its main priorities. To achieve these properties, Nomad is implemented on top of COBRA, a privacy-preserving State Machine Replication Framework, which splits data into different random shares to achieve confidentiality in stored data, and implements Paillier, a homomorphic encryption scheme that allows operations on top of encrypted data, allowing confidentiality in transactional data. As a result of the work done, any banking system can adopt this solution and be able to use Nomad's currency.

Keywords: Blockchain, Cryptocurrency, Privacy, Confidentiality, BFT SMR.

Contents

1 Introduction **1**

1.1 Context and Motivation . . . 1
1.2 Goals and Contributions . . . 3
1.3 Thesis Outline . . . 4

2 Background and Related Work **5**

2.1 Private cryptocurrencies . . . 5
2.1.1 Monero . . . 6
2.1.2 ZCash . . . 8
2.1.3 zk-SNARK . . . 8
2.1.4 Dash . . . 11
2.1.5 Beam . . . 13
2.2 BFT State Machine Replication . . . 15
2.2.1 COBRA . . . 16
2.2.2 BFT-SMaRT . . . 20
2.3 Privacy preserving Technologies . . . 21
2.3.1 Homomorphic Encryption . . . 21

3 Nomad Solution Analysis **23**

3.1 Architecture and System Model . . . 23
3.1.1 Adversary Model . . . 26
3.1.2 Nomad's Properties . . . 26
3.1.3 Accounting Model . . . 27
3.1.4 Replicas . . . 29
3.1.5 Clients . . . 30
3.1.6 Transactions . . . 32
3.1.7 Ledger . . . 35
3.2 Protocols and Algorithms . . . 36
3.2.1 State Machine Replication Protocol . . . 36
3.2.2 Secret Sharing Scheme Protocol . . . 37
3.2.3 Homomorphic Encryption Protocol . . . 37

3.3 Use Cases . . . 38
3.3.1 Confidential European Central Bank . . . 38
3.3.2 Confidential E-commerce Platform . . . 39
3.3.3 Confidential Gaming Platform . . . 39
3.3.4 Confidential Food Delivery Platform . . . 39
3.3.5 Confidential Crypto Exchange Platform . . . 40

4 Nomad Implementation **41**
4.1 Overview and Structure . . . 41
4.2 Wallet Implementation . . . 42
4.2.1 Master Keys . . . 42
4.2.2 Child Keys . . . 43
4.2.3 Transaction Dispatch . . . 43
4.3 Client Implementation . . . 44
4.3.1 Node . . . 44
4.3.2 Client . . . 44
4.3.3 Validator . . . 45
4.3.4 Client Controller . . . 46
4.3.5 Storage . . . 46
4.4 Server Implementation . . . 47
4.4.1 Server . . . 47
4.4.2 Storage . . . 48
4.4.3 Snapshot . . . 48
4.5 Transaction Implementation . . . 49
4.5.1 Transaction . . . 49
4.5.2 Transaction Output . . . 49
4.5.3 Transaction Input . . . 50
4.5.4 Transaction Signature . . . 50
4.5.5 Transaction Validation . . . 51

5 Experimental Evaluation **53**
5.1 Setup and Methodology . . . 53
5.2 Operations Overall: Updates, Proves, Reads . . . 54
5.2.1 Transaction Operation . . . 55
5.2.1.1 Comparing different underlying systems . . . 55
5.2.1.2 Comparing with other cryptocurrencies . . . 56

6 Conclusion **59**
6.1 Future Work . . . 59

References **61**

List of Figures

2.1 Monero Ring Signatures 7
2.2 Zcash zk-SNARKs 10
2.3 Secret sharing scheme 18
2.4 BFT-SMaRt Architecture 20

3.1 Nomad's Architecture 25
3.2 UTXO Model 28
3.3 Wallet Structure 31
3.4 Transaction Flowchart 34

5.1 Nomad's Throughput for 4 replicas 57
5.2 Nomad's Throughput for 7 replicas 57
5.3 Nomad's Throughput for 10 replicas 57

List of Tables

5.1 Throughput and latency for Nomad operations. 54
5.2 Throughput for different systems considering 4,7 and 10 replicas. 55
5.3 Throughput and latency for different cryptocurrencies. 56

Chapter 1

Introduction

1.1 Context and Motivation

Emerging technologies shape the way we see the world around us. From the creation of the world wide web to the massive adoption of smartphones, today the world is facing a new revolution, the blockchain era [1]. Blockchain technology has as its main purpose providing a reasonable lightning-fast option that intends to build ground-breaking payment methods for all business types [2].

Blockchain technology has been rising in recent years and its unique properties such as immutability, transparency, anonymity, and security create a new basis of trust for business transactions that could contribute to a considerable simplification and acceleration of the economy [3]. Blockchain processes as a distributed ledger, where it holds a record of every data that everyone in the network can access, thereby eliminating the need for a central authority. Day after day, companies are adopting this technology to drive greater transparency across the digital information ecosystem, boosting awareness of this technology and giving it a proper use like never seen before. However, blockchain technology has its flaws, and it's vulnerable to the leakage of transactional privacy because of the visibility of what happens within the network by everyone [4].

Data is the world's most valuable asset, and every person in the world has associated a piece of data, which defines the uniqueness of each individual. Every time a person uses their data, the person is facing the risk that their data can be disclosed or lost through unauthorized access. Not only that but, the shape of our data tells us everything we need to know about our data, from its obvious features to confidential secrets. What could happen if this data is changed or stolen from prying eyes? Countries, enterprises, and businesses depend on the quality of their data and need to have it available and secure at the highest possible cost. That's why blockchain cannot protect all data in the world.

Although blockchain technology keeps growing, its users are aware of and cite privacy and security as their main concerns when adopting such a service [4]. In this context, as the main application and use case, blockchains were built primarily to offer a decentralized payment system, through the creation of digital

currencies, known as cryptocurrencies (i.e., blockchain was firstly introduced in 2008, with the creation of Bitcoin [5] by Satoshi Nakamoto), allowing clients to make payments and transfer money between each other. That is, cryptocurrencies nowadays are being assessed as the next generation of financial money, helping make payment systems more efficient. Nevertheless, most of the existing cryptocurrencies, raising the adjacent blockchain issues, do not provide any means for securing the confidentiality of a user's data.

Everyone has the right to privacy and untraceability in their financial transactions that is also applied for cryptocurrency transactions. Some cryptocurrencies focus on convenient payment or fast transactions, while some others are more focused on providing anonymity for users. With the emergence of cryptocurrencies and the surrounding problem of privacy, different types of currencies started to be developed, such as private currencies. Private currencies are designed to have high levels of cryptographic qualities for providing anonymity and confidentiality for their users. That is, private currencies' core property is to obfuscate transaction information to offer privacy to users, such as: (1) not saving transaction history as they are routed across multiple networks; (2) encrypting transaction information to protect confidentiality; (3) hiding the identities of the entities transacting; (4) it is ensured that all coins are pairwise indistinguishable [6].

Consequently, private currencies present alarming problems, particularly regarding their confidentiality and scalability [7]. More specifically, we recognized that some private cryptocurrencies offer partial privacy, having a payment system that supports public and private transactions [8]. Additionally, we identified confidential payment systems that rely upon a single master node [9], which makes it have a single point of failure. Moreover, there are systems that do not meet the amount of confidentiality they initially offered.

To properly handle these problems, we developed Nomad, a confidential decentralized payment system application on top of COBRA, a secret sharing scheme framework, that offers privacy and confidentiality as its main properties. Nomad implements a Byzantine Fault Tolerance Consensus Protocol (i.e., BFT), following a state machine replication approach. Works like Astro [10] and SMaRtCHAIN [11] follow a state machine replication approach, where Nomad relies on certain properties coming from these projects.

This dissertation aims to characterize how Nomad can be used to execute transactions between peers providing anonymity hiding their identities and providing confidentiality, obscuring transaction data.

However, given the limitation of private currencies, specific questions arise: How secure is Nomad? How private is Nomad? How much confidentiality does Nomad offer? How could Nomad be scalable in the future in case needed? These questions make the scope of this project.

1.2 Goals and Contributions

This dissertation is based on the work developed by Bessani et al. [12] proposing a library implementing a BFT state machine replication consensus protocol, that is, BFT-SMaRt. Following this work, Vassantlal et al. [13] found that despite integrity and availability is offered by BFT-SMaRt in a state machine replication environment, confidentiality protection is a limitation, not presented in this kind of system. With that, COBRA was proposed, a protocol for dynamic proactive secret sharing that allows implementing confidentiality in practical BFT SMR systems [14]. Nevertheless, neither BFT-SMaRt nor COBRA offer confidentiality on transactional data or anonymity in any application. Astro [10], a payment system application developed on top of BFT-SMaRt doesn't offer any levels of confidentiality and anonymity in a payment system following a BFT SMR approach. Other works, such as SmartChain [11], a blockchain platform, is another application based on BFT-SMaRt, following a BFT SMR approach [15]. Neither confidentiality nor anonymity is offered within this platform.

In this master thesis, we propose Nomad, a confidential cryptocurrency that tackles the privacy problem of private currencies and the confidentiality in existing payment system applications following a state machine replication approach.

Nomad's objectives are to offer a confidential payment system, using a state machine replication protocol where all transactions are private, with the following properties:

- **Confidentiality**, in which the amount of a transaction as well as the data that are part of it are obscured.
- **Anonymity**, in which the entities involved in a transaction have their identities hidden.
- **Integrity**, in which the amount of a transaction being transferred over the network is not changed at any time.
- **Verifiability**, in which the authenticity of a transaction's data is maintained and verified.

We wanted scalability to be part of the properties to be fulfilled, but since the scalability of the system depends on the underlying technologies, such as COBRA and BFT-SMaRt, the way we can control scalability within Nomad is through the algorithms and data structures used.

In this context, the first goal of this thesis was to design and implement Nomad on top of COBRA to learn how could we implement a payment system application following a secret sharing scheme. This goal was intended for us to understand how COBRA works and how we could implement a solution on top of it.

The second goal was to understand how could we adapt Nomad, not only to have confidentiality on stored data (i.e., offered by COBRA) but to implement confidentiality on top of transactional data, where data being transferred in the network is secret, and offer anonymity to the end users involved in a transaction. This goal was planned to have Nomad fully confidential.

The third goal was to evaluate Nomad's solution, comparing it with other private currencies, regarding performance and security.

In this scenario, the contributions of this thesis are the following:

- A preliminary study of private currencies and applications implemented on top of a BFT SMR system, their advantages and disadvantages in order to create a collection of know-how about these works.
- An approach based on a BFT SMR system, fault-tolerant, that allows transactions to be sent and executed between a pool of clients, that enhances privacy for transaction data, confidentiality on top of stored data, and anonymity for users.
- Adapt and offer a fully confidential BFT SMR payment system, that delivers confidentiality in all its operations.

1.3 Thesis Outline

This thesis is organized as follows:

- Chapter 1 gives an introduction, describes the current problem and explains the goals and contributions of this work.
- Chapter 2 presents background and related work. The chapter begins by describying different private currencies, their architecture and underlying protocols. We then introduce the state machine replication problem and its fundamentals. We then present a few systems following a BFT SMR approach. Finally, we finalize this chapter by explaining and giving a description of privacy-preserving technologies.
- Chapter 3 introduces Nomad's solution, explaining its properties, its underlying architecture, describing Nomad's components, the protocols supporting Nomad, and finally, we provide examples of real use cases applying Nomad's solution.
- Chapter 4 describes Nomad more technically. We follow the same approach as chapter 3 but the depth of explanation is greater, varying from the configurations used to achieve Nomad's properties to the protocols that are part of Nomad's foundation.
- Chapter 5 presents the results gathered from testing Nomad. We describe how we conducted the experiments, what was achieved, and give real comparisons with different private currencies and systems following a BFT SMR approach.
- Chapter 6 provides the conclusions of the developed research and discusses possible future work.

Chapter 2

Background and Related Work

In the subsequent chapter related work is given that served as support to develop our work. The context within these projects and the problems that they tackle took us to a level of motivation that is relevant. Section 2.1 analyses and discusses different private currencies, their protocols, including properties regarding privacy plus confidentiality, and services that anonymize transactions plus addresses, supporting these currencies. Section 2.2 explores the BFT State Machine Replication Paradigm, where COBRA [13], a secret sharing scheme, and BFT-SMaRt [12], a state machine replication library, are addressed and reviewed. At last, Section 2.3, comprehends the purpose of using emerging technologies, their benefits, and utilities, as these are being adopted nowadays.

2.1 Private cryptocurrencies

A cryptocurrency is a digital currency, that is, a digital asset that allows secure payments online. Cryptocurrencies are built and based on top of blockchain technology, where they are distributed in a network across a large number of workstations and aren't issued by a central authority, that is, cryptocurrencies operate without the need of a central third party (i.e., governments cannot interfere or manipulate the creation of cryptocurrencies). As cryptocurrencies are applications that are established on top of blockchains, a blockchain's purpose is to keep track of cryptocurrency transactions and the integrity of their data.

Cryptography and encryption algorithms are the foundation of cryptocurrencies, and despite anonymity and security being offered as a main priority of these virtual currencies, full anonymity and confidentiality cannot be supported by any cryptocurrency. This is because the major part of blockchains that exist worldwide, their data can be accessed on the public ledger by anyone who has access to the network. However, the cryptocurrencies anonymity problem made the adoption of cryptocurrencies with privacy-adapted features.

Private cryptocurrencies are built-in with privacy features, declared to offer security features such as anonymity, confidentiality, and integrity. There are many types of private cryptocurrencies, and in

the following chapter, we present different private cryptocurrencies, protocols implemented, including properties regarding privacy plus confidentiality, and services that anonymize transactions, supporting these currencies.

2.1.1 Monero

Monero [16], the most famous private currency in the market, is known for its use of complete privacy and anonymity.

In Monero's infrastructure, all data held is private as well properties like anonymity and privacy are ensured. Anonymization of transactions and addresses is accomplished by ring signatures, stealth addresses, and ring confidential transactions in Monero's ecosystem.

In Monero's network, all nodes are given a set of keys: a public/private view key, a public/private spend key, and a public address, which is a public key (i.e., a public key in the crypto space is an identity - address in the context of cryptocurrencies). The spend key is used regarding sending payments, the view key is used to display information about transactions received, and the public address is used by other nodes for a specific node to receive the amount sent in those transactions.

Ring Signatures. Ring Signature [17] is a type of signature implemented by Monero to ensure the privacy of the identity of the node initiating the transaction. It is a type of digital signature where multiple nodes sign a transaction to ensure that it is valid. A Monero ring signature is composed of the actual signer and non-signers (i.e., the actual signer defines a set of members of its choosing), that together form a ring. The signer creates a signature using its private key and all the other ring members' public keys. The actual signer is a one-time spend key that corresponds to an output being sent from the sender's wallet. The non-signers are past transactions outputs pull from the blockchain, which act as decoys. These outputs together make up the inputs of a transaction. To a third party, all of the inputs of the transaction appear equally likely to be the output that is going to be spent in that transaction. Therefore, a given node cannot tell which is the actual signer of the transaction since all inputs are indistinguishable from each other. This makes it impossible to know who is the node that initiated the transaction since the transaction is signed by keys from different nodes. This brings up a question where if a given node doesn't know which output should be used, how can double-spending be avoided? Through each output a key image is generated, that is, a cryptographic key derived from an output being spent, that is concatenated to a ring signature transaction, making it unique for each output. According to Monero's cryptographic properties, it isn't possible to know which output generated which key image. A list of all key images is held on the blockchain so that miners can verify that outputs aren't being spent twice. Figure 2.1 shows how the ring signatures process works.

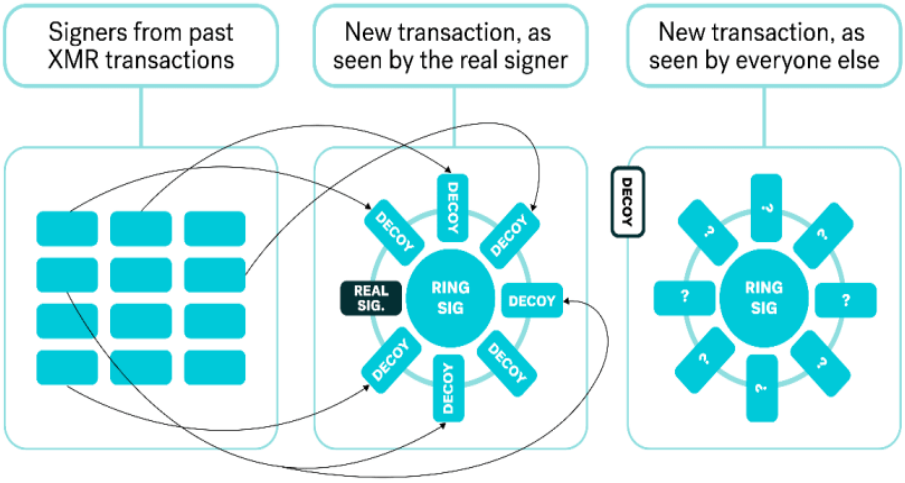

Figure 2.1: Monero Ring Signatures

Stealth Addresses. Stealth Addresses, known as one-time public keys are also an important feature for ensuring the privacy of the recipient's identity. It is a randomly generated one-time public key using the recipient's public view key and public spend key, concatenated with random data. All nodes in the network can see this one-time public key (i.e., unique generated address), but only the recipient (i.e., node to which the transaction is sent), holder of its private view key can locate this stealth address. After the recipient finds this address, it calculates a one-time private key that is equal to the one-time public key generated by the sender. These stealth addresses are unique on the blockchain, where they prevent outputs from being associated with wallet addresses.

Ring Confidential Transactions. Ring Confidential Transactions [18] is a technique of using a commitment scheme to hide the amount of a transaction in Monero's blockchain. Ring CT is an improved version of ring signatures. Pedersen Commitments are the reason how Monero hides transaction amounts. The objective of a commitment scheme is to allow a party to commit to a specific value, without revealing or change it. This is achieved by homomorphic encryption. When Monero is spent within a transaction, the sum of the inputs that a specific node is spending minus the sum of the outputs a specific node is sending equals zero.

This is possible because input and output amounts are encrypted using homomorphic Pedersen commitments, meaning that although a specific node cannot know the real values, it can verify that they sum to zero. Proving that they sum to zero is achieved by proving that a specific node knows the private key of the 'zero' Pedersen commitment (i.e., commitment generated by a node that is part of a ring signature) for at least one of the combinations of possible inputs and outputs. Pedersen's commitments also make it

possible for nodes on the blockchain to verify that transactions are valid and coins aren't created secretively. That being said, so if a specific amount within a specific transaction cannot be determined, how does Monero prevents senders to commit to negative values? This is achieved by Range Proofs, where amounts (i.e., committed through a specific Pedersen commitment) that are used within a transaction are greater than zero and less than a specific value. This allows the system to secure the supply of coins in circulation.

Advantages & Disadvantages. As advantages, in Monero's system: privacy is more attractive to more users; complete anonymity is given; low transaction costs, which will continue to decrease over time; Monero also allows users to transparently share the transactions they make according to their wishes, where if a specific user wants to prove their ownership of a certain amount of Monero for tax purposes, they can share their private view key with their countries tax authority.

Monero seems to be the ideal private currency but has some disadvantages concerning the size of blocks and it's transactions. To achieve a lower price regarding gas fees, blocks on the Monero blockchain can reach very large sizes which can make it impractical for users not being able to transfer all the data on the network if the currency is adopted globally since it takes a lot of space (i.e, there is no limit on the production of coins). On the other side, the size of transactions is very large due to the number of encryption algorithms that are used to ensure privacy. With this project, other than ensuring privacy we concern also amidst the scalability problem. Also, a group of researchers from the University of Illinois at Urbana-Champaign and Princeton University wrote a paper implementing two distinct techniques that can break Monero's untraceability. This paper was released before the implementation of Ring CT (i.e., January 2017), whereas it stands for these days, one of the problems was fixed by Monero's developers while the other still remains unsolved. More can be found here [19].

2.1.2 ZCash

ZCash [20] is an optional private currency that is based on the zk-SNARK protocol. Anonymity and privacy are optional properties, in which the node initiating the transaction has the ability to choose whether it will be public or private. Before explaining how transactions and addresses work on ZCash blockchain, we firstly introduce the zk-SNARK protocol.

2.1.3 zk-SNARK

zk-SNARK [21] technology is a cryptographic proof based on the zero-knowledge proof protocol, where through facilitating mathematical proofs, one party proves that has possession of certain information without revealing what that information is (i.e., for example, a secret message). zk-SNARK is an acronym for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge.

In the Bitcoin protocol, addresses, transactions, inputs, and outputs values are all public on the blockchain where they are all linked together and can be visible to everyone on the network [5]. On the other sideline, currencies that implement the zk-SNARK protocol prove that all the rules needed for a valid transaction have been satisfied without disclosing any information about the transaction, addresses involved and the amount being transferred. With zk-SNARK, a user constructs a proof to show that, with high probability:

- The sum of the input values is equal to the sum of the output values for each transaction.
- The sender proves that it has the private keys of the input, which gives a user the authority to spend. The spending of outputs is achieved by the same scheme that Bitcoin uses, through unspent transaction outputs (UTXOs), where transactions are tracked to see if they are spendable.
- The private spending keys of the input are cryptographically linked to a signature that concerns the entire transaction, in a manner in which a transaction cannot be modified by a party who did not know these private keys.
- The zero-knowledge proof proves that the user is in fact owner of the keys that sign the transaction, authorizing it to spend the funds.

There are, of course, concerns related to the zk-SNARK protocol. For instance, if someone was able to access the private key that was used to create the parameters of the proof protocol, they could create false proofs that nonetheless looked valid to verifiers. This would allow a person to create new tokens of a specific currency through a counterfeiting process. To prevent this from happening, currencies that implement the zk-SNARK protocol are designed to circumvent this failure.

T-addresses & Z-addresses. ZCash uses two types of addresses: t-address and z-address. Addresses of type t-address are public addresses so that they can be seen by any node on the blockchain. Addresses whose type is z-address are private addresses, which guarantee that a transaction is private, like its value and also the identity of the sender/recipient. Based on these addresses, transactions on the ZCash blockchain can be sent in two ways: transparent and protected. Transparent transactions work in the same way as Bitcoin, whose Zcash codebase was originally based on: they are sent between public addresses and are recorded on the blockchain. Protected transactions, on the other hand, make use of the zk-SNARK protocol, to allow completely anonymous transactions to be sent over an immutable public blockchain. The fact that the transaction took place is recorded in the public ledger, but the sending and receiving addresses and the amount sent aren't revealed to the public. All protected transactions within the ZCash system occur within a shielding pool. Figure 2.2 shows what the zero-knowledge layer adds in terms of confidentiality. With this layer, all transactions in Zcash are fully encrypted on the blockchain to ensure that the confidentiality of transaction metadata is fully preserved.

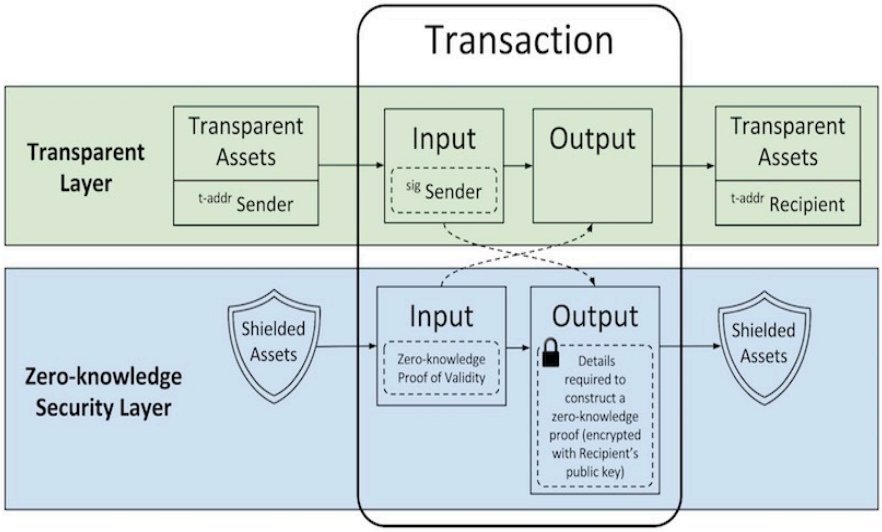

Figure 2.2: Zcash zk-SNARKs

How zk-SNARK protocol is applied to create a shielded transaction?

In ZCash, equivalent to a UTXO in Bitcoin is called a commitment, and spending a commitment is known as revealing a nullifier. In Zcash blockchain, all nodes keep a list of every commitment and every nullifier that exists, where only the hash of these operations are stored, to avoid exposing any information regarding commitments and nullifiers and their linkage.

Whenever an input is created within a shielded transaction, a commitment is published by the sender that consists of a hash of the destiny address, the input that was sent, the amount being sent, a unique secret number (i.e., rho), and a random nonce.

When a shielded transaction is spent, the node (i.e., sender) uses its private spending key to publish a nullifier that corresponds to the hash of the unique secret number (i.e., rho) from a commitment that exists in the blockchain that has not been spent, and also a zero-knowledge proof is provided saying that the specific node is authorized to spend it. This zero-knowledge proof verifies that: for each input created, a commitment exists; all commitments and nullifiers created so far were correctly executed; since the hash of each nullifier is unique it is unlikely for two different nullifiers to collide.

Regarding spending keys that are used to unlock the rights for its owners to spend coins associated with their accounts, Zcash uses other different keys to create and check zero proofs. These different keys are generated through a multi-party ceremony hosted by different supernodes where all shielded transactions of Zcash rely on zk-snark public parameters to construct and verify the zero-knowledge proofs of the Zcash blockchain. In this ceremony, a trusted set of nodes generate a part of a private/public key pair, where each node has a share (i.e., shard) of the public/private key set. Once the public/private

pair shards are complete, the nodes combine their public key shards to generate the public parameters of Zcash, and then each one destroys their shard of the private key (i.e., this is done to make it infeasible for an attacker to try to create these public parameters and spend counterfeit coins). These public parameters are then sent to all nodes on the blockchain. Whenever two nodes want to create and verify a specific zero-knowledge proof, these public parameters are used to generate a key pair, producing a proving and a verification key. For all shielded transactions, a sender uses a specific proving key to create a proof saying that the inputs within a transaction are valid. Now, only the miners verify if the proof is correct through the verification key, to check if the prover's computation is valid and all consensus rules were followed. This verification process is done by the miners because they are the ones that verify transactions and create new blocks with these transactions.

A major disadvantage of applying the zk-SNARK protocol is that to apply it to shielded transactions is much more complex and costly because of the major computational work involved, where creating shielded transactions can take several seconds (i.e., sometimes even minutes), while verifying that a transaction is valid only takes very few seconds. This is the reason why in the Zcash system most users prefer to execute transactions through transparent addresses.

Advantages & Disadvantages. Studies have shown that cryptocurrencies that implement optional privacy, as is the case with ZCash, often end up with users not choosing to use private transactions. A group of researchers from the University College London concluded that not only do users prefer to execute public transactions through the public ledger, but the main problem in the existence of this shielding pool on the Zcash system is that anonymous transactions require more computational power, which has led to longer transaction times and larger memory requirements [8]. This problem is being addressed by ZCash where they propose a Sampling technique, that its purpose is to decrease the intensity of computational power when dealing with shielded transactions. These researchers also conclude that transactions and addresses on the Zcash blockchain can be traceable through the use of heuristics.

2.1.4 Dash

Dash [22] is a private currency that guarantees optional privacy and speed of transactions. Its purpose is to be used for payments and transactions that are fast (i.e., instantaneous), secure, and provides low-fees. As in Zcash, in Dash blockchain is it possible to make transactions in a way that the identity of both the sender and the recipient can be revealed or not. This is achieved through two services: InstaSend and PrivateSend. The nodes qualified for the proper execution of these services are called master nodes.

Master nodes are servers that have their resources allocated to processing and verifying transactions in the Dash network in exchange for rewards. Master nodes belong to the system and have more than 1000 coins in their account. These nodes have certain primary functions: they have to maintain an up-to-date copy of the blockchain; they have to support InstaSend (i.e., instant transactions); they have to

support PrivateSend (i.e., private transactions); they vote on decisions about the system; they decide how funds are distributed through the network.

InstaSend is a type of service where certain transactions can bypass the mining process performed by miners and passed directly to the master nodes, where they can be processed. Transactions that use this kind of service are confirmed by the master node. It takes about 4 seconds for an InstantSend transaction to get confirmed. Here, master nodes are responsible for doing the initial consensus on the validation of each specific transaction, and they promise to the rest of the nodes in the network that the transaction will be fully validated in the coming blocks and not allowing the spending of the inputs of this transaction during the confirmation period.

PrivateSend is a type of service that uses a process called Coin Mixing (i.e., CoinJoin). A particular user choosing this feature mixes its coins with other users' coins. The network of master nodes is responsible for this process. The identity of both sender and recipient is hidden, while also the amount being transferred is unknown. This mixing process doesn't involve coins being transferred between different accounts, but it serves to obscure where funds come from.

CoinJoin. CoinJoin [23] is a protocol that anonymizes transactions by employing a method called "coin mix". To increase privacy, the background of this protocol is to let multiple users combine all inputs and outputs from different transactions on a single transaction. The idea is to combine different addresses with random addresses and have no linkability between them. This provides privacy similar to k-anonymity [24] (assuming k participants) since no observer can distinguish which coins end up at each recipient. To achieve this, a master node is needed to coordinate nodes that desire to mix coins. In CoinJoin, the specific steps are needed to process a transaction:

1. First, a specific node (i.e., client) determines if it wants to join an existing pool of users (i.e., users that want to mix their coins) or create a new one.

2. The node sends a message to the master node in charge of its CoinJoin transaction to join an existing pool or create a new one.

3. The master node sends a message back to the client providing a pool status, where specific flags (i.e., these flags have specific codes represented through bits) are within the message that indicates if the client can join the pool.

4. Upon receiving the master node message, the client provides to the master node a list of all inputs and a list of all outputs where the values from its account should be sent.

5. The master node checks the client's message and sends it an update of the pool status saying that its inputs and outputs were added to the pool

6. After getting all inputs and outputs from each client that is going to participate in this CoinJoin transaction, a message is sent to the total group with the final transaction for verification.

7. Following the verification of the final transaction, each client sign their inputs and send them back to the master node.

8. Finally, the master node verifies the signed inputs, creates a message to broadcast the transaction through the network, and report to the clients that the transaction is complete.

The key detail of CoinJoin is that all inputs within the CoinJoin transaction need to have the same amount. For example, if a node wants to send 1 DASH through a CoinJoin transaction, then all inputs in that transaction also need to be equal to 1 DASH. This increases anonymity and unlinkability between inputs and where they were spent. Another feature of CoinJoin is that a specific value that needs to be transferred can be split into different denominations (i.e., they represent also input values). From the sender side, all of these denominations have a given address that after the denomination (i.e, input value) is spent, the address no longer exists. From an attacker's perspective, is going to be very difficult to know where the inputs are going to be spent to which output. The bigger the transaction with loads of inputs, the harder the difficulty to link the inputs to the outputs, but also more computational power is needed to process the transaction.

Advantages & Disadvantages. As advantages, Dash although applying optional privacy can ensure its users have financial and transaction privacy through the use of the PrivateSend service. Also, with the InstantSend feature transactions are much faster compared with other cryptocurrencies that require lots of time to confirm transactions. Since CoinJoin can merge transactions into a single big transaction, transaction fees can be lower.

The one and only disadvantage that is the most critical one is regarding privacy issues. Since all CoinJoin transactions need to be followed by a specific master node, these transactions can still be traced down if an attacker controls the master nodes to which the transaction belongs. This represents a single point of failure with a massive impact. To the best of our knowledge, the CoinJoin protocol is a good protocol to be implemented in a way that is implemented with other protocols that enhance privacy and anonymity properties. For some developers in the crypto space community, it's claimed that the mixing process of coins from different nodes is nothing more than a coin thumble [7].

2.1.5 Beam

Beam [25] is a complete privacy currency, and one of the firsts to implement the Mimblewimble technology. Beam's project wants to act as a store of value, like Bitcoin, but in a confidential way where all data is private. Within the Beam blockchain: all transactions are private, meaning that users are the only party able to determine what information is available and to which parties, having full control over their data following their wishes and applicable laws; user's addresses (i.e., sender and receiver addresses) aren't stored on the blockchain; users create new addresses every time they make a transaction; scalability is

achieved through compact blockchain size, which through the technologies offered by Mimblewimble, Beam's blockchain is much smaller than any other blockchain implementation. From what we know, Beam, as Monero, offers true full privacy, whereas the unique difference between these two is the scalability that the former offers. While other blockchains have been struggling with scalability problems, Beam believes they've resolved the issue by having a strong and compact blockchain. As it stands for today, the Beam project has not had a breakthrough since its launching, where other coins that offer privacy as default, like Monero, still remains the leader in the private crypto space.

Mimblewimble. Mimblewimble [26] is a privacy protocol based on Bulletproof (i.e., recently developed zero-knowledge proof protocol), that empowers a fully private transaction platform through a unique security framework that is significantly different from Bitcoin. In Mimblewimble, there are no addresses and transactions are entirely confidential. Its distributed ledger is also comparatively thicker than other blockchains. Mimblewimble's mission is to improve users privacy, and it can achieve that with the combination of diverse technologies and cryptographic protocols:

- **Elliptic curve cryptography (ECC)** enables Private-Public key encryption, a way to prove you know something without revealing the content of the encrypted information.
- **Confidential transactions** allow for public confirmation of the transaction without exposing any significant details such as amounts or addresses. This is achieved by utilizing Pedersen Commitment scheme (i.e., previously explained in Monero's section). This step is done with the help of ECC, where nodes on the Beam network can blind or hide information regarding the amount that is being transferred. With this association of technologies, only the sender and the receiver know the amount which are being transacted with, by not allowing access to anyone who isn't involved in the specific transaction.
- **CoinJoin** is built through a mechanism that enables transactions from multiple senders to be grouped into a single transaction. As in Zcash, by creating a mix of transactions (i.e., where all coins from different accounts are mixed), Beam can offer better privacy since nodes that aren't involved in this process cannot identify which specific payment went to which node. Also, the workload of the miners that need to validate this type of transaction is more efficient than coins that need to have all of the transaction data validate because Beam's confidential transactions run on homomorphic encryption.
- **Dandelion**, an improved network layer anonymity solution that contains increased privacy techniques. It uses hops in between nodes before publicizing the transaction to the neighboring nodes. Its usage is to hide nodes' IP addresses. Since the launch of this protocol, numerous faults were discovered that point to its deanonymization with some idealistic adversaries. A new and improved version was developed, Dandelion++.

- **Cut-through**, a feature that enables data compression, where currencies that implement Mimblewimble have smaller blocks on their blockchains. For example, node A wants to send a specific amount to node B. Then, node B wants to send the same amount (i.e., in another transaction) to node C. With Cut-through technology, instead of intermediary transactions form a block, they are merged into a large transaction. The exclusive thing that the network needs to bother is storing the current state without the need to do this for the whole transaction tale. This feature delivers better scalability on Beam's blockchain. Nodes joining the network only need to work with compacted information, such as information on system state and blockchain headers. Under the usage of Cut-through, the amount of information that a node needs to retrieve is reduced due to the fact that they don't need the entire transaction history.

In contrast to what is found in other blockchains, through Mimblewimble, Beam intends to build a system without the use of addresses, which are used as proof of ownership of the coins. As Bitcoin, Beam wants to use the UTXO model just to check if a specific output was spent and to sign also the usage of blinding factors (i.e., from what we know they encrypt the amount of cryptocurrency the users want to send). The commitment is the only information kept in the blockchain. Through the technologies offered by the Mimblewimble protocol, personal information is removed from the blockchain structure, whereas each address on the Beam network has an expiration time of 24 hours. To increase privacy, whenever a node is interacting within a transaction, new addresses should be created.

Advantages & Disadvantages. Mimblewimble has some main advantages when compared with other blockchain platforms that use distinct protocols and algorithms: anonymity, fungibility, and scalability.

In contrast, Mimblewimble has its limitations: longer transaction throughput because systems that support confidential transactions are going to suffer from lower transaction speed; and since this technology is reliant on digital signatures it's also vulnerable to quantum computer attacks, where powerful calculations are made to deanonymized transactions. Also, Mimblewimble is a protocol that joins technologies that are offered by other privacy coins. But, the conjunction of these technologies makes this project have great potential, but the truth is that the much it promises has not been achieved to this day.

2.2 BFT State Machine Replication

Byzantine Fault-Tolerant State Machine Replication is a classic paradigm and is one of the most difficult challenges faced by a distributed computer system, like blockchains [15] [14]. Cryptocurrencies are applications that are built on top of specific blockchains and these blockchains are systems that are comprised of many participants that together form a large network. Whenever these participants need to agree or reach a consensus regularly about the current state of the blockchain, some nodes are going to

be faulty (i.e., malicious nodes that intend to disrupt the system) [27]. Byzantine Fault Tolerance offers a property that allows it to overcome this problem (i.e., safeguard of the system) and form a consensus even when are some faulty nodes who disagree with the rest [27]. Within each blockchain, a consensus protocol must exist, so that nodes can validate and agree on a single chain. Through the following chapter, we aim to present two main works that address the BFT SMR problem, that serve as the core of our work:

1. **COBRA**, a framework that provides proactive verifiable secret sharing in a dynamic groups of processes and allows implementing confidentiality in practical BFT SMR systems.
2. **BFT-SMaRt**, a BFT SMR library carrying all characteristics needed in functional BFT SMR systems such as tolerance to asynchrony, crash recovery, and group reconfiguration.

2.2.1 COBRA

COBRA (i.e., COnfidential Byzantine ReplicAction) is a framework composed of a set of protocols, which through dynamic proactive secret sharing tries to preserve the confidentiality of stored data on a distributed system through a state machine replication approach. COBRA is meant to:

- Build specific protocols, such as for distributed polynomial generation, share recovery, and dynamic secret resharing that can operate in non-synchronous environments.
- Possess a dynamic set of participants, so to support services to enable the operation of a distributed system like replica servers crashing, recovering, leaving, and joining the system.
- Preserve confidentiality, integrity, and availability on the system.
- Support comprehensive large and scalable applications that have privacy and fault tolerance as main requirements.

COBRA is implemented on top of a BFT library, called BFT-SMaRt (i.e., briefly explain in the next section), which supports specific features much needed in a BFT system as crash recovery and group reconfiguration. COBRA integrates its dynamic proactive secret sharing scheme with BFT-SMaRt in order to provide BFT services in a BFT system as well as providing confidentiality in stored data. Also, as stated previously, COBRA is built upon a secret sharing scheme, where nodes on the network ensure that the data stored is confidential from prying eyes. A secret sharing scheme protects data, called secrets, by splitting it into different parts, called shares, where the secret can only be reconstructed when different shares are merged. The advantage of this scheme is that if an adversary compromises a node, only its share is obtained, which reveals nothing about the secret itself. [13]

In this type of scheme, usually there are three different roles, which are: the dealer, the shareholders and the combiner. The dealer's task is to generate n shares of a secret. Then, these shares are distributed among n shareholders. Upon receiving the shares, shareholders store them in a safe storage location. Finally, the combiner collect $t + 1$ shares from distinct shareholders and reconstructs the secret. Figure 2.3 shows the different roles in a secret sharing scheme.

Nowadays, as different types of secret sharing schemes have some disadvantages regarding mobile adversaries, COBRA proposes a proactive verifiable secret sharing scheme that can be used to protect against mobile adversaries by regularly renewing the shares stored by the replicas without revealing the secret. Consequently, if replicas need to reboot and shares require to be changed within a time frame before an adversary collects enough of them, the data is going to remain secret. Also, if an adversary can collect many shares that together can break the secret, this isn't going to be useful, because, after the renewing of the shares, the old shares cannot be combined with the renewed shares.

The central idea of a secret sharing scheme is based on arithmetic through the use of polynomials, more specifically on the Lagrange interpolation theorem, where a random polynomial P of degree k is built such that $P(0) = S$ (i.e, equal to the secret) and n shares $< S1, ..., Sn >$ are generated as points of $P(x)$, where $Si = (i, P(i))$. The secret S can only be recovered by picking $k + 1$ shares (i.e., represented as polynomial points), interpolating the polynomial, where calculations are made to recover P and determine $P(0)$. Meanwhile, COBRA has as its core a protocol for distributed polynomial generation. This protocol uses a Byzantine consensus protocol to create secrets and random polynomials with shares allocated to $f + 1$ correct servers. For example, a specific secret is represented through a polynomial, where replicas have a share of the secret, and each one of these replicas has assigned a random polynomial. For safety, no replica knows the central polynomial that forms the secret, so to recover the secret, all replicas that have a share within the secret use their polynomial, and they are all summed up to form the central polynomial. After this process, $P(0)$ can be calculated and the secret is retrieved. Because COBRA is implemented on top of BFT-SMaRt, share recovery, secret resharing, and group reconfiguration are services achieved by COBRA with the use of auxiliary polynomials through the application of a distributed polynomial generation protocol. Among the flaws that exist today in distributed systems regarding malicious adversaries, often during secret resharing, some correct replicas could complete the generated polynomial with invalid shares. This is done with the use of commitments. During secret resharing, all replicas must be assigned with new shares, derived from the old shares to keep the secrecy of the secret, or otherwise, the secret cannot be recovered and can be lost. To address this issue, COBRA introduces a recovery protocol that during the polynomial generation, replicas that are responsible for the creation of invalid shares are removed. With this approach, correct replicas can execute the recovery protocol and new shares are generated and distributed correctly (i.e., through secret resharing or group reconfiguration).

To understand more in-depth COBRA's DPSS scheme, the protocols that are used are the ones followed:

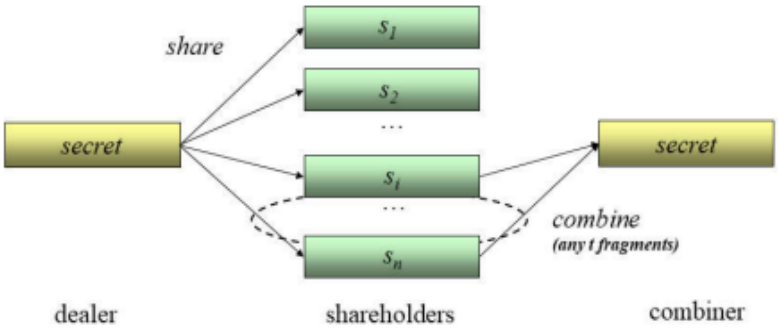

Figure 2.3: Secret sharing scheme

- **Distributed Polynomial Generation:** this protocol acknowledges a group of replicas to create a random polynomial P of degree k with a hidden point (x, y), where y can take two different values depending on what service is needed. For example, with y being zero, the protocol is going to be called for share recovery or if y is a random secret, the protocol is going to be called for secret resharing. At the end of the protocol execution, each correct replica obtains a point (i, P(i)) of the polynomial P and its commitment cP. This protocol splits into three steps: the first one is that each replica locally generates a random polynomial and distribute its shares to the group of replicas; the second one is that a set of f + 1 of these polynomials is selected and a Byzantine consensus (i.e., through BFT-SMaRt) is used to ensure that all correct processes agree on the same assemblage of polynomials; and the third one is that the f + 1 selected polynomials are summed, resulting in the shares of a polynomial P (i.e., which represents the secret). The main property ensured by this protocol is that at least f + 1 correct processes will obtain a valid point of the polynomial P, resulting in a share for each replica.

- **Share Recovery:** if a replica fails and it recovers, the same can obtain its shares through a share recovery procedure. To recover a specific share, for example, Sk, the polynomial where the share (i.e., represented by a point) is generated, can be interpolated in a way that Sk = P(k). The replica that failed, send messages to the other replicas in the group asking to generate a random recovery polynomial R (i.e., through the distributed polynomial generation protocol), where R(k) = 0 (i.e., this means that the protocol is going to be executed with the share recovery mode). As explained previously, f + 1 replicas will obtain valid shares of the polynomial R, and to maintain the privacy of their shares, replicas blind their shares of the polynomial P, generating shares of P + R, which are sent to the replica that failed. After receiving f + 1 correct shares of P + R, the replica that crashed can recover its share by calculating (P+R)(k) = P(k) + R(k) = P(k). This method can only be achieved if all correct replicas have correct shares of the polynomial P. But, as COBRA

is implemented on top of a BFT SMR library, and BFT systems are used, a malicious adversary within the system can collide with faulty replicas and distribute invalid shares that are used with the recovery of a replica. To address this issue on COBRA, after the distribution of shares to correct replicas at the end of the distributed polynomial generation protocol, they recognize that their share is invalid and they have specific proofs of the faulty replica that lapsed during the generation of the polynomial that resulted in invalid shares. With this solution, every replica within the group of replicas can ignore messages from the malicious replica and restart the protocol with a new one. The share recovery process can fail up to f times (i.e., because of faulty replicas), removing f faulty replicas, and in the end, the recovery will execute accurately.

- **Secret Resharing:** the idea of this protocol is to be executed in a way where a group of replicas transfer the shared secret with their shares to a new group of replicas intending to maintain the secrecy of the secret. This protocol makes that a different version of the distributed polynomial generation protocol is executed in a process that two polynomials are created, Q and Q0, one for the current group of replicas and the other one for the new group of replicas. Each polynomial encodes the same secret, in which Q(0) = Q(0) 0. Replicas that are to be replaced blind their shares with the polynomial Q and send them to the new group of replicas. Within the new group of replicas, each correct one joins f + 1 blinded shares and the root of the blinded polynomial is calculated. Through this blinded polynomial, the correct replicas of the new group of replicas obtain renewed shares when subtracting their valid points of Q0 (i.e., their actual shares) with the blinded polynomial, obtaining the original secret. From what was discussed earlier regarding valid shares, a replica within the new group of replicas can only obtain its new shares if it has a valid point of the polynomial Q0 (i.e., this is cannot be ensured for all replicas). Consequently, when invalid points are received, replicas with invalid shares must use the share recovery protocol to get a valid share of Q0, and malicious replicas that were responsible for creating these invalid shares are removed from the group of replicas.

As discussed, the COBRA protocol stack for confidential BFT SMR enables fundamental features typically needed in practical BFT SMR systems, such as replica recovery and group reconfiguration. This work describes how COBRA DPSS scheme can be used for supporting BFT SMR operations and achieving confidentiality on stored data.

To the best of our knowledge, COBRA produces a very good work in ensuring the confidentiality of stored data in BFT SMR systems, through the use of a protocol for dynamic proactive secret sharing. Stating this, the application we built is implemented on top of COBRA.

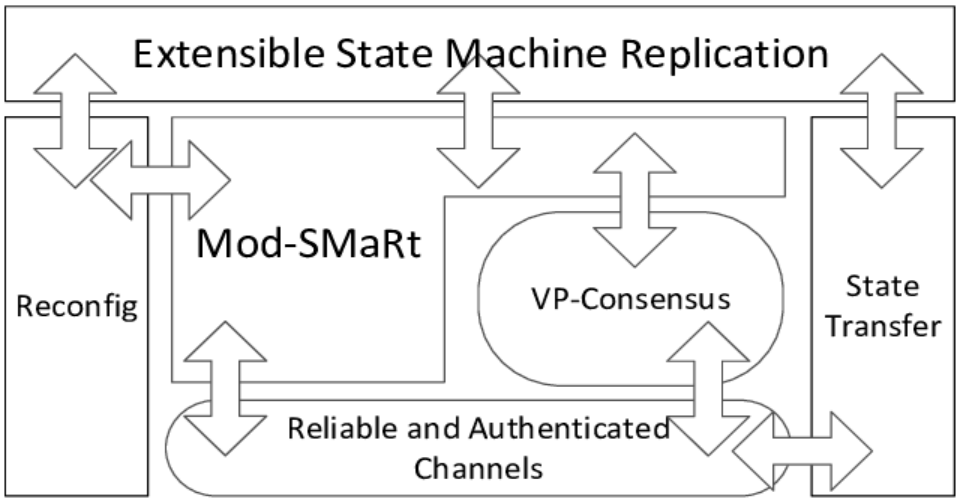

Figure 2.4: BFT-SMaRt Architecture

2.2.2 BFT-SMaRT

BFT-SMaRt is a state machine replication library developed in Java. This library tries to improve fault tolerance, scalability, and performance in distributed systems by implementing a protocol very similar to works like PBFT (i.e, most famous architecture that provides a practical Byzantine state machine replication that tolerates Byzantine faults by assuming there are independent node failures and manipulated messages sent through specific nodes) [28], plus other protocols which offer state transfer and reconfiguration of replicas [12].

The BFT-SMaRt library runs the state machine, which stores the state of the system. The state machine receives a set of inputs, and then implement these inputs in a sequential order using a transition function to generate outputs and an updated state. To scale the system, the state machine replication method is used, where a fault-tolerant service is implemented by replicating numerous servers, called replicas, which are coordinated and interact with clients [29] [15] [12]. BFT-SMaRt offers the properties we want to have in a distributed BFT SMR system. If we use the advanced features of the BFT-SMaRt library, instead of writing naively over a regular blockchain, we can perform faster and give a stronger level of durability. To the best of our knowledge, the main disadvantage that the BFT-SMaRt library possesses is confidentiality, which offers none of it. But, as discussed before, COBRA is implemented on top of BFT-SMaRt, and through the use of its secret sharing scheme, we can achieve confidentiality on stored data and have a BFT SMR system through the features offered by the BFT-SMaRt library. Figure 2.4 details the architecture of BFT-SMaRt, showing the different existing modules.

As far as we know, BFT-SMaRt is the most up-to-date work to implement state machine replication, remaining actively supported.

2.3 Privacy preserving Technologies

Emerging private technologies have turned into faster actual adoption, as businesses are analyzing their massive potential and embracing them into their corporations nowadays, as the speed of innovation is increasing more and more and the need for security in their data is a must.

As in this thesis we are interested in technologies involving distributed systems, technologies involving blockchains and cryptographic algorithms are of more relevance.

Through this following chapter, we are going to address one technology, which is homomorphic encryption technology, a type of encryption that allows computations to be performed directly on encrypted data without the need to decrypt it with a secret key.

2.3.1 Homomorphic Encryption

Currently, one of the biggest problems in blockchains is data transparency, where data that is being transferred between nodes can be seen by everyone who has access to the public ledger. This happens in public blockchains, where any entity can use the blockchain and see what is happening, and because of that, there is zero privacy. By contrast, a permissioned blockchain is a type of blockchain that can only be accessed by entities that have permission to join the network. Although isn't totally private, it offers more privacy than public blockchains.

For example, to achieve privacy on transferring data, one node could encrypt a transaction with a secret key and then send it to the receiver of that transaction, and the receiving node could decrypt the transaction. This isn't feasible because blockchains are systems that are composed of thousands of different nodes, and in this situation, whenever a node wants to send a transaction to another node, it needs to send also the key to decrypt the transaction (i.e., symmetric encryption is being discussed here but asymmetric encryption can also be used with a central authority). So a new technology, known as homomorphic encryption, was developed to resolve this scalability problem and also to facilitate privacy issues.

Homomorphic Encryption is a special type of encryption where computations can be executed on top of encrypted data, keeping it private without the need to decrypt the data with a specific decryption key. Results from these computations are encrypted and can only be decrypted by the owner of the decryption key. The implementation of homomorphic encryption on top of blockchains is useful because it offers the privacy needed of blockchain transactions without a node to reveal its amount. Through homomorphic encryption, transactions in a specific blockchain can be validated using commitment schemes, where through random values trade between nodes, a commitment value of the transaction is calculated, and a node can prove the veracity of that transaction without revealing its data.

The idea of using this type of technology in our work is given by the Paillier cryptosystem. Additionally, this type of encryption scheme is used within our confidential cryptocurrency to ensure data privacy. [13]

As stated in Pretty Good Confidential project [30], is it possible to accomplish a confidential decentralized payment system through a homomorphic public-key encryption scheme. The great disadvantage of homomorphic encryption is that the computations involved around it are complex and heavy.

Regarding Paillier, it has the following advantages:

- **Security:** The Paillier cryptosystem is based on the mathematical concept of homomorphic encryption, which provides a high level of security for encrypted data.
- **Flexibility:** The Paillier cryptosystem allows for the performance of computations on encrypted data, without the need for decryption.
- **Ease of use:** The Paillier cryptosystem is relatively easy to use compared to other public-key encryption systems.

On the other hand, it has the following disadvantages:

- **Performance:** The Paillier cryptosystem can be computationally intensive, leading to slow performance in some applications.
- **Limited functionality:** The Paillier cryptosystem is limited to specific types of computations, and cannot be used for all types of encrypted data.

Chapter 3

Nomad Solution Analysis

In this chapter, we introduce the solution, explaining every property embedded inside Nomad, detail the architecture behind Nomad, describe each one of its components, point out the attack surface and the adversary model, explain the protocols that support Nomad's system, and lastly, give examples of real use cases where Nomad can be applied.

3.1 Architecture and System Model

Nomad's architecture follows a typical client-server model, where the client sends a request for and with data to the server through a secure cryptographic communication protocol. The server, upon receiving the request, processes the request and sends a response back to the client. Since we developed a confidential decentralized payment system where all the transactions of Nomad are ordered and recorded properly, we used a state machine replication protocol where a set of stateful replicas execute the same batch of requests in the same order, deterministically. On top of the requests, a secret sharing scheme protocol is applied to split a request into different shares and distribute them between the replicas. Ultimately, a homomorphic encryption protocol is implemented on top of the transactions to ensure that operations are performed within Nomad without revealing confidential data.

Nomad follows a permissioned blockchain model, where access is required to be part of it. In Nomad, a control layer runs on top of it, implemented by the replicas, that governs the actions performed by the allowed participants. This is because Nomad is implemented on top of a BFT State Machine Replication Protocol that provides a consensus scheme. A BFT State Machine Replication Protocol adapts very well for permissioned blockchains, where every replica of the network is known, all of them execute the consensus (i.e., to decide the outcome of incoming operations) and need permission to participate in the network. We deemed that one of Nomad's features is maintaining order in stored data, so a consensus mechanism is well defined.

Nomad's architecture main components are as follows:

- **Client:** a client is the end-user responsible for sending transactions between themselves and requests to the replicas, to be persisted.
- **Replica:** a server known as a replica, is responsible for the execution of the state machine replication protocol, and more specifically the consensus algorithm. Different replicas have their services replicated and they are accountable for storing the transactions and the shares of different data where the secret sharing algorithm is applied. Within Nomad, replicas are the engine to process requests and form a secure network.
- **Transaction:** a transaction inside Nomad follows the same scheme of digital transactions. A transaction is a transfer of coins (i.e., digital currency) from one peer to another peer. When clients transfer cryptocurrency funds, transactions are then recorded in a local ledger maintained by all replicas inside Nomad.
- **Wallet:** a digital wallet is a mechanism that allows clients to store cryptocurrencies, and make transfers to other clients with the help of cryptographic keys.
- **Ledger:** inside Nomad's cryptocurrency solution, a private local ledger is maintained and updated through time. This private ledger is a record-keeping system, acting as a storage service. The ledger keeps a record of all transactions executed between clients.
- **Snapshot:** to keep Nomad's state up and running from faulty reasons, snapshot backups are used to restore Nomad's state to a running state and serve as a recovery point for Nomad when a snapshot was taken. It isn't a backup copy because it doesn't store data, but rather it is only defining where and how data is stored and organized.

Nomad's architecture, presented in Figure 3.1, shows the overall communication between clients and replicas, and replicas between themselves.

Following the architecture presented in Figure 3.1, client's hold:

1. A master public key $\mathcal{K}pu$ and a master private key $\mathcal{K}pr$.
2. A list of its own outputs $\mathcal{O}utxo$, with different outputs $U0, U1, U2, ..., Un$.
3. A private log file $\mathcal{P}l$ that consists of different public keys, $P0, P1, P2, ..., Pn$. The elements are one-time public keys used in specific client transactions.

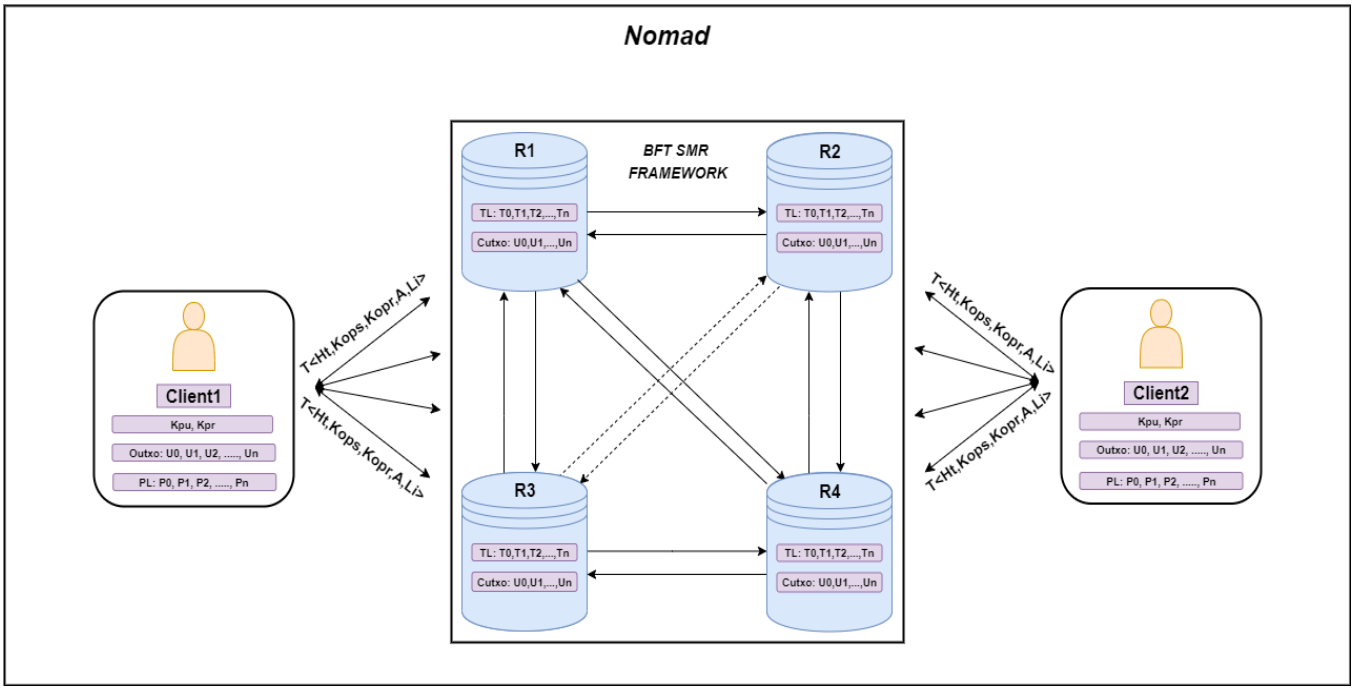

Figure 3.1: Nomad's Architecture

As demonstrated in Figure 3.1, client's transfer and receive transactions with:

1. A hash of the transaction $\mathcal{H}t$.
2. A one-time public key $\mathcal{K}ops$ that belongs to the sender.
3. A one-time public key $\mathcal{K}opr$ that belongs to the recipient.
4. The amount $\mathcal{A}$ being transferred, which is encrypted by a homomorphic encryption scheme.
5. The list of inputs $\mathcal{L}i$ that contains the outputs to be spent, which are encrypted by the same homomorphic encryption scheme mentioned above.

Illustrated in Figure 3.1, inside BFT SMR Framework replicas are going to communicate with each other and preserve:

1. A unique log $\mathcal{T}l$ with the transactions executed on Nomad.
2. A list of all clients outputs $\mathcal{C}utxo$, with different outputs $U0, U1, U2, ..., Un$.

3.1.1 Adversary Model

All systems are predisposed to fail. Without Byzantine Fault Tolerance, any member of a network could provide invalid information to a network and undermine the reliability of the network. For example, in Bitcoin, a node can join the network without permission and start creating transactions and broadcasting blocks. When a system is Byzantine Fault Tolerant, nodes can verify every transaction and block within the network. However, systems following a BFT SMR approach, are prone to Byzantine failures. Replicas and clients who fail, are said to be faulty or corrupted. On the other side, clients or replicas that are not faulty are said to be correct or honest.

A Byzantine Fault Tolerant System evolves in views, numbered sequentially. In each view v, one server is the primary, and the others are the backups. Therefore, clients multicast requests to all servers, and then, servers execute the consensus.

By implementing Nomad on top of COBRA, we consider the same adversarial model as COBRA. Within Nomad, we consider active and adaptive adversaries, trying to compromise the confidentiality, integrity, and availability of the system. Furthermore, we consider these adversaries capable of controlling the network and are able to corrupt a certain number of clients and a portion of the replicas in each view. In each view v, the system remains secure as long as the adversary doesn't control more than 3f+1 nodes. This guarantee comes from COBRA, and the way Nomad works. From COBRA, we know that replicas can still be corrupted even after they leave the system. In Nomad, a faulty client trying to send two times the same transaction, get detailed information about other clients, and impersonate other clients, cannot perform these actions and fails by doing so. Within Nomad, we assume that the adversary cannot break the cryptographic premises employed in our protocols.

In addition, Nomad may be the target of other attacks that are not in the scope of COBRA or BFT-SMaRt. These attacks can be of external origin, such as a Distributed Denial of Service, where we have a set of attackers that exhaust the network's resources, thus causing the network to be inoperable. Another type of attack, very common in existing cryptocurrencies, is the Sybil attack. In a Sybil attack, an adversary creates multiple fake identities to gain control over a large portion of the network, potentially allowing them to carry out malicious activities or disrupt the normal functioning of the network. Other known attacks that Nomad may be vulnerable to are the Eclipse attack and the Man-in-the-middle attack. Regarding attacks from internal sources, internal threats may include attacks from within the network, such as collusion between nodes, double-spending, and other forms of fraud. Collusion between nodes can occur when multiple nodes work together to control the network and undermine its integrity. Double-spending involves a user attempting to spend the same cryptocurrency more than once, by manipulating the system to approve multiple transactions with the same coins.

3.1.2 Nomad's Properties

We assumed from the start that we wanted to build Nomad offering the highest level of privacy and confidentiality (i.e., to the best of our knowledge). Nomad's architecture solution was designed based on

three central properties:

1. **Anonymity:** with anonymity, clients inside Nomad can have their identities obscured and unknown. The important idea here is that Nomad ensures a client's privacy by hiding its identity when involved in a transaction. Whenever a transaction is executed, both the sender and receiver addresses are hidden. This is achieved by a client using a different address every time it's affected in a transaction.

2. **Confidentiality:** with confidentiality, transactions inside Nomad have their amounts private. This property was the most difficult one to achieve, but it was also the most important because it gives the level of privacy and confidentiality that we want. Each time a transaction is executed, the amount being transferred in the network is hidden, as well as when the transaction is persisted by the replicas in the ledger, the amount is hidden. This is possible by enforcing a modified homomorphic encryption protocol, that allows calculations over encrypted data. The idea is that each time a transaction is executed, its value is encrypted with a specific cryptographic key, which allows for validating the transaction value without ever decrypting it.

3. **Verifiability:** this property is important within Nomad because it verifies the significant steps: (1) how can a client spend coins in a transaction without revealing its identity; (2) how the value of a coin is validated; (3) how can a recipient can deliver coins to its wallet.

These properties together constitute the possibility of building a confidential decentralized payment system with a certain degree of accuracy, privacy and confidentiality.

3.1.3 Accounting Model

Cryptocurrency is a type of digital asset that is intangible and that uses cryptography to securely send transactions within a decentralized system. In every executed transaction new cryptocurrencies are created. Within Nomad, the accounting model that we define covers the following topics:

1. Authorize cryptocurrencies to be spent and sent between customers.

2. Securely store cryptocurrencies in a safe place.

3. Create new cryptocurrencies from spent cryptocurrencies.

Within Nomad system model, we define an accounting method in which coins are managed using digital wallets. This accounting method, known as UTXO (i.e., Unspent Transaction Output) was first introduced by Bitcoin [5], which fundamentally spent cryptocurrencies called outputs represent the output of a transaction received by a user (i.e., in this case, the coins received by the user). Therefore, that user

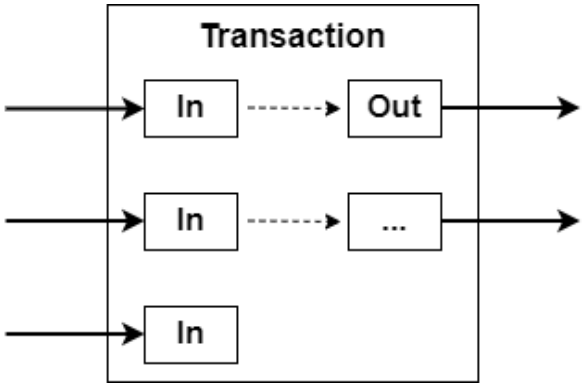

Figure 3.2: UTXO Model

can spend these coins in the future. Figure 3.2 demonstrates how inputs and outputs combine within a transaction.

By Nomad following this accounting method, transactions are created by consuming existing UTXOs and producing new ones. UTXOs cannot be divided and are consumed altogether within a transaction. If the total value of all UTXOs within a transaction exceeds the desired value, then the change is returned to the entity that initiated the transaction.

In Nomad, each client has a digital wallet derived from a master public key and a master private key. As explained, we designate a cryptocurrency wallet as a service that allows a client to store and retrieve their cryptocurrencies. The private key purpose is for a client to prove the ownership of the funds stored in its wallet (i.e., needed to authorize a transaction). Furthermore, the private key is also used in the digital signature of a transaction, to prove that the client that issued the transaction is the rightful owner of the coins being spent.

The purpose of the public key is to generate new public keys to be used in transactions. For example, we can imagine these keys as a house, where the public key represents a house address and the private key represents the house keys, respectively. In this situation, people will need the house address to send mail, yet only the owner can enter the house with the respective house keys and get access to the received mail. So, every time a transaction is executed within Nomad, the recipient must always provide its public key to the sender.

As explained, in this accounting model, each coin describes a specific amount of currency owned by a client. As in Bitcoin, in our model, each transaction is composed of inputs and outputs. Inputs consume an existing UTXO, while outputs create a new UTXO. When a transaction takes place, within the transaction, a given number of inputs, that is a list of UTXOs, is consumed to produce a certain number of outputs.

The assumption we came across about this implementation was that each client has a list of their own UTXOs in their wallet. To keep a record of all UTXOs within Nomad, we consider that replicas have

access to a list that contains all the UTXOs of all clients. For a client to have its balance updated, the client communicates with the replicas and provides proof to them that it owns the outputs it wants access to. The steps are the following:

1. First, the client sends the ids of the UTXOs it wants access to.

2. Therefore, replicas upon receiving the requested UTXOS by a client, need to verify if these UTXOs belong to the client. In this step, is important to highlight that anonymity is guaranteed because as explained, a one-time public key is used per UTXO. To perform this operation, replicas send a challenge to the client so that the latter can prove that it can access the UTXO. In none of the messages exchanged between the client and the replicas does the client need to provide its address or identify himself. What allows this proof to be established is through an RSA key parameter present in the UTXO. This RSA key is composed of a public parameter and a private parameter. When UTXOs are persisted by replicas, the UTXOs are stored with the public parameter of the RSA key inside. As a result, replicas use this public parameter to encrypt a challenge and send it to the client.

3. Consequently, the client upon receiving the encrypted challenge uses the private parameter of the RSA key and decrypts the challenge. With the challenge already decrypted, the client sends it to the replicas.

4. Finally, when replicas receive the response to the challenge, they compare the results, and depending on the outcome, the client proves that the UTXO belongs to it.

This process is conducted whenever a client needs to retrieve a specific UTXO. In the end, a client's balance is the total value of its outputs, all added together. After getting all its UTXOs, the client performs a sum operation to return the total amount of its available balance. The operation to check a given client's balance is always called when the client acts as the sender or recipient in a transaction. Any new UTXOs it receives when the balance check is called, these UTXOs are added to its list of local outputs in its wallet.

3.1.4 Replicas

We consider Nomad's system with $n \geq 3f + 1$ replicas that tolerate at most f simultaneously Byzantine faults. Nomad is partially asynchronous and the communication between replicas is done using secure point-to-point channels (i.e., using SSL/TLS).

Since Nomad is built on top of a state machine replication protocol, therefore Nomad architecture on the server-side was designed following this idea. By having different replicas via a replication process, Nomad can increase its service performance and capacity since having more replicas implies having more

resources. Additionally, by using distinct replicas, Nomad can achieve fault tolerance, making the system operate despite the failure of a fraction of these replicas.

Within Nomad, replicas are servers that have copies of the same state machine service replicated. When requests to execute operations arrive from the clients, a consensus protocol keeps replicas synchronized, in which:

1. In a first step, replicas start at the same state.

2. In a second step, replicas agree on an order to execute the incoming request.

3. In a third step, if replicas are correct, then they execute the operations one at a time in an agreed order, so that every replica reaches the same final state.

Replicas within Nomad are in charge of maintaining and updating Nomad's ledger of transactions, so we reached a proposal for the storing of the transactions. Since Nomad's architecture follows a BFT state machine approach, each replica produces a unique log for state synchronization. Revisiting Nomad's properties, we wanted to be sure that we keep confidentiality when storing transactions, so every value inside a transaction is first encrypted following a homomorphic encryption scheme and then a secret sharing algorithm is performed on top of the encrypted value. Each time a transaction is executed and validated, upon receiving the transaction, each replica appends that specific transaction to its unique log of transactions. Afterward, the consensus algorithm is executed by all replicas to persist the completed transactions in an orderly manner. Every replica has stored the following data: (1) a list of all client's unspent transactions; (2) a unique transaction log that stores all transactions executed inside Nomad's system. Replicas are the core entities that keep Nomad's state up and running.

Concerning Nomad properties, just confidentiality is being applied on the replicas side, because although they need to share information and reach the same state, each replica have different shares for the same value where a secret sharing scheme is applied. For that reason, each replica stores its shares internally without sharing with anyone. Regarding anonymity, this property is not achieved at the replica level, because all replicas must know each other to execute the consensus protocol. Only clients remain anonymous during a transaction.

3.1.5 Clients

A client in Nomad's context is responsible for creating a private network, where all clients talk to each other in a peer-to-peer manner, in which each client is known as a node. The client is the end-user responsible for invoking operations, by sending requests to replicas to be executed. There are two different types of nodes in Nomad, which are:

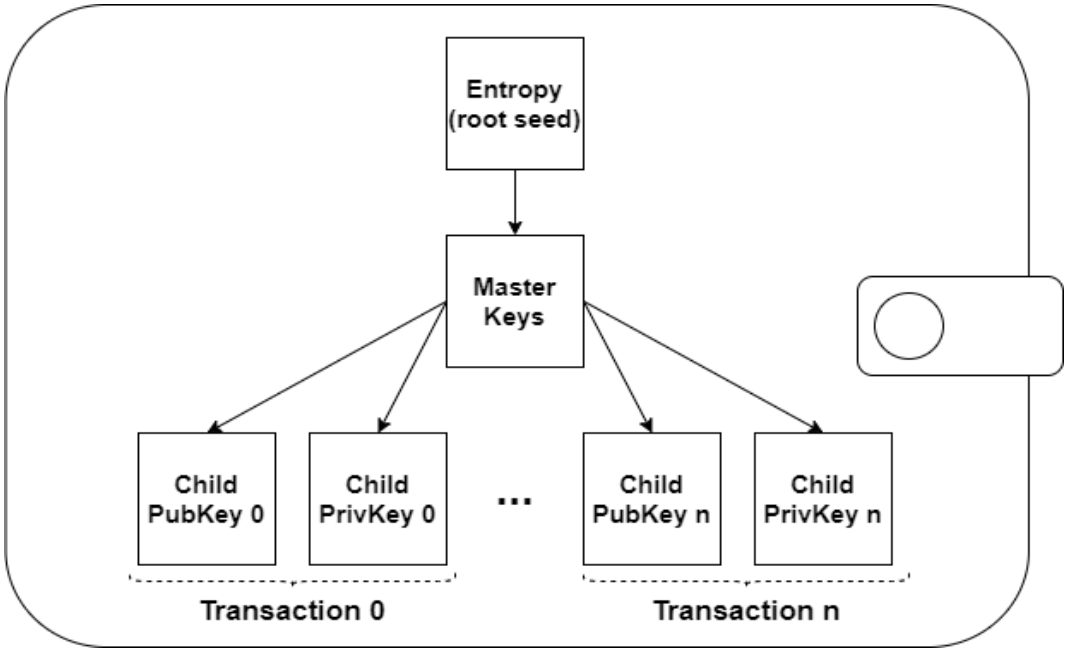

Figure 3.3: Wallet Structure

1. **Client:** this node represents a regular client, which can send and receive transactions. Regular clients don't send transactions directly to other clients but send transactions to validators. Validators then dispatch transactions to replicas, to be executed.

2. **Validator:** this node represents a particular client, which is responsible for validating transactions within Nomad. In addition, has all the features of a regular client. After validating transactions, then they are sent to the replicas to be executed and persisted.

In Nomad, each client has a dedicated wallet. Essentially, a client needs a wallet to store and send cryptocurrencies. Unlike physical wallets, a wallet in Nomad's ecosystem doesn't hold any money, but instead, it holds the master keys needed to access and control cryptocurrencies on Nomad. For that reason, when a new client enters Nomad's system, a new wallet is created with a pair of master private and public keys.

Concerning Nomad properties, the client applies anonymity. As explained, to achieve anonymity, every time two clients are involved in a transaction, their wallets create a one-time public key from their master public key to use for that specific transaction. Then, these newly created one-time public keys are stored in a data structure on the client side.

The public keys used in transactions are stored because when a client retrieves its balance from the replicas, the client needs to prove that the UTXOs belong to it. By providing the ID of the one-time public key, the client asks the replicas to deliver a specific UTXO. Then, the replicas send a secret message,

encrypted by an RSA key (i.e., presented in the actual UTXO), with the client decrypting the encrypted challenge, sending the challenge back to the replicas, and is then allowed to recover its balance.

Regarding Nomad's confidentiality property, this property is not directly enforced on the client side but over transaction data. In a later section, this procedure of enforcing confidentiality over transaction data is detailed. Figure 3.3 shows the wallet structure within Nomad.

3.1.6 Transactions

A transaction is the main functionality within Nomad. A transaction is a transfer of digital coins from one address to another. Every transaction must always be signed by the sender before the validation phase. This proves that the signer of that transaction is the rightful owner of the coins being transacted. When transactions are executed, they are logged in a local ledger maintained by all replicas. Figure 3.4 displays the different steps to perform a transaction.

Concerning transaction properties, Nomad ensures integrity, anonymity, and confidentiality. To achieve these properties, Nomad:

- **Integrity:** relies on public key cryptography to ensure the integrity of transactions created on the network. To transfer coins, each client has a master public key and a master private key that controls owned coins.
- **Anonymity:** each time a client sends a transaction, a new public key is created from the master public key, representing a unique address used once. In contrast, the private key is a secret key as it authorizes the spending of any funds.
- **Confidentiality:** we came up with the idea of using a technique of homomorphic encryption. This type of encryption allowed us to reach the level of confidentiality we were looking for. In this case, each time a transaction is sent, its value is previously encrypted with a public key generated from very large random prime numbers. For that reason, the value of the transaction is never decrypted until it reaches the recipient.

Each transaction conducted by Nomad's network is composed of four different phases:

- **Composition:** in the first phase, the composition phase, the transaction is built on the sender's side. This phase is divided into different steps, such as:

    1. The first step to define is the value being sent. The sender collects from its list of UTXOs (i.e., unspent outputs) the sufficient number of outputs that is equivalent to or higher than the value being transferred. If the sum of the results exceeds the amount being transferred, a new transaction is created with the correct exchange being delivered to the sender.

2. Then, all collected outputs are grouped in a list of inputs. It is important to mention that for every output being spent on a transaction, its value is encrypted using a homomorphic encryption algorithm instance, that is, a public key generated by the homomorphic cryptosystem.
3. Subsequently, as explained in the client's section, a one-time public key, representing an address, is created from the client's master public key (i.e., generated at the time of the wallet creation).
4. Then, the value to be transferred is encrypted with the same public key generated by the homomorphic cryptosystem instance used to encrypt each one of the outputs belonging to the transaction.
5. Finally, with all the components needed to form a transaction, the transaction is then signed with a one-time private key, created by the sender's master private key.

- **Validation:** in the second phase, the validation phase, as explained in the client's section, a special type of node is used to validate a transaction, called validator. A validator is a client but with an additional feature of validating transactions. Each time a client issues a transaction to be sent to another client, first it needs to be validated by a validator.

- **Dispatch:** in the third phase, the dispatch phase, the transaction is sent from the validator to the replicas, through a confidential proxy channel, where a secret sharing algorithm is going to be executed on top of the transaction, more specifically, on top of the encrypted value being transferred.

- **Received:** in the last phase, the received phase, replicas receive the transaction with the same information, but with a different transaction value. In this case, because of the secret sharing algorithm, replicas receive different shares. Therefore, all replicas append the transaction to the local ledger, updating the ledger, and at the same time, they execute the consensus protocol to reach the same state. Lastly, a confirmation is sent to the sender stating that the transaction was successful, and from now on, the recipient is eligible to receive the new output.

Each time that a client wants to issue a transaction, its composition is:

1. $\mathcal{H} \longrightarrow$ **h(pk_snd, pk_rct, tx_val):** transaction ID, which is a hash of the transaction, that is, we concatenate the sender's public key, recipient's public key, and the transaction value. Therefore, we apply a hashing algorithm on top of this concatenated data, in which the result is the id. There is no need to include a random number in this step because since the keys used in different transactions are always different, we don't run the risk of two transactions having the same id.

2. Sender's one-time public key, which is always generated by the sender's master public key in the sender's wallet.

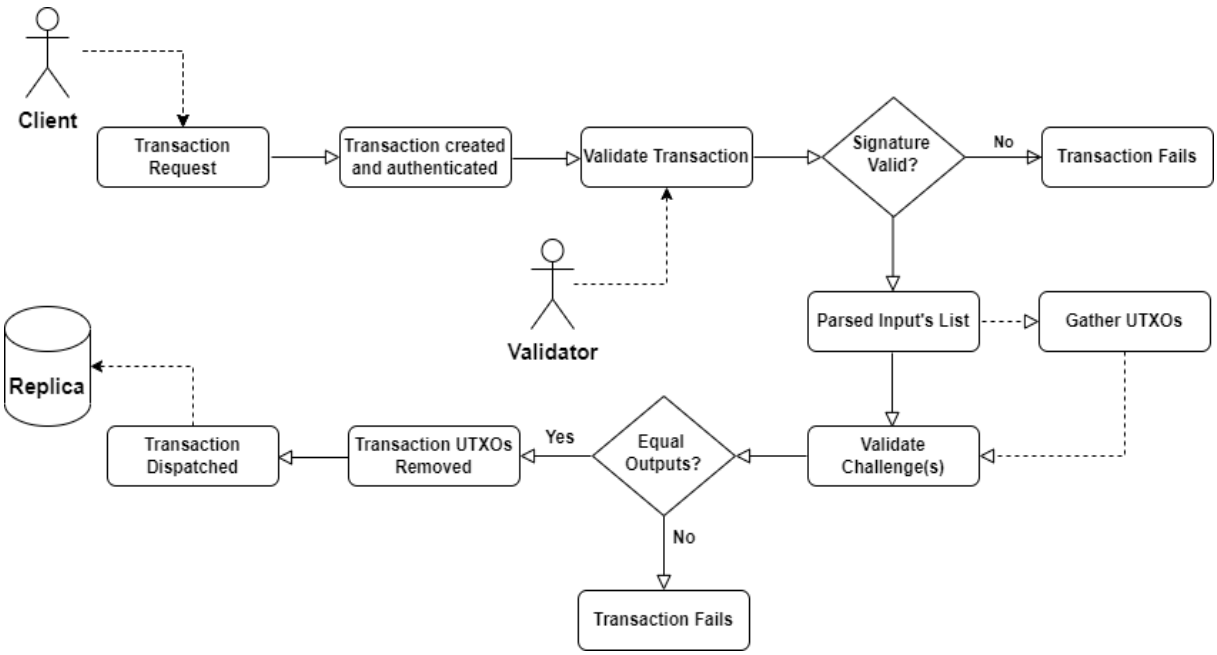

Figure 3.4: Transaction Flowchart

3. Recipient's one-time public key, which follows the same pattern as the sender's one-time public key, where this public key is constantly going to be created by the recipient's master public key in the recipient's wallet.

4. The amount being transferred, obscured by the homomorphic cipher scheme.

5. Inputs that are part of the transaction, which are encrypted with the same homomorphic scheme.

$$\mathbf{T} \longrightarrow [t(c) = t(pk_s, pk_r, enc_v, L_i)]$$

t(c) : transaction per client
pk$_s$: sender's one-time public key
pk$_r$: recipient's one-time public key
enc$_v$: encrypted transaction value
L$_i$: inputs list

Validation of a transaction encompasses different steps, such as:

1. Validation of the transaction signature. For this validation, the transaction signature is calculated

again with the sender's and recipients' public keys and the value being transferred. If the signature matches the received signature, then it's validated.

2. Check if the transaction outputs within the list of inputs of the transaction are unspent. In this step, the validator retrieves from the replicas, the UTXOs corresponding to the UTXOs contained in the received transaction. For this step to be accomplished, a proof must be given for each UTXO, that the requested UTXO (i.e., from the list of UTXOs maintained by the replicas) can be retrieved to the requester (i.e., more details about this proof step are given in the implementation section). Therefore, the validator checks if the received UTXOs weren't spent previously. In this step, the id and the value of the received UTXO are compared with the UTXO that's inside the transaction. This step allow us to address the "double-spending" problem.

3. Generate a new transaction output, which is the UTXO that is going to be sent to the replicas to be persisted in their list of UTXOs, and what the recipient is going to receive in its wallet. Additionally, as already explained, if the summed up value of the UTXOs within the transaction is higher than the real value that needs to be transferred, a new transaction output is created with the exchange to be delivered to the sender.

4. Send to the replicas the list of UTXOs being spent on that transaction, to be removed from the list of UTXOs maintained by the replicas.

5. Send the validated transaction.

3.1.7 Ledger

Nomad's ledger foundation is the same as the distributed ledger technology, which is a digital system that records transactions of assets in multiple places at the same time. In Nomad's case, transactions are recorded at the same time in each replica. A distributed ledger can record different types of data, from static data, for example, data objects, and documents, to dynamic data, such as financial data. Inside the Nomad network, we assume a zero-trust network in which trust isn't implicit. Each replica in Nomad has a copy of the same data in the form of a distributed private ledger. Because of the consensus protocol, if a replica's ledger is corrupted in any way, it will be rejected by the majority of the replicas when trying to update the ledger. This can be achieved by the execution of a snapshot of the system from time to time to keep Nomad's state running from faulty reasons. The importance of performing snapshots of the network occasionally is important to restore Nomad's state to a valid state and also acts as a recovery point when the snapshot was taken. The snapshot idea is to define where and how data is stored/organized inside Nomad.

Nomad ledger reflects Nomad state, in which every time changes are made to the ledger, these are reflected and copied to all participants. In Nomad's ledger, each replica generates a record of each transaction and creates a consensus on its veracity. In the end, is important to define a record-storage service, which keeps evidence of all executed transactions.

3.2 Protocols and Algorithms

In the following chapter, the main protocols that are the foundation of Nomad will be explained and how they were used in the context of Nomad.

3.2.1 State Machine Replication Protocol

The concept of State Machine Replication is a significant concept that served as the basis for the development of Nomad.

State Machine Replication is a general approach to implementing a replication model by replicating servers and coordinating a given client's interactions with them, i.e., an arbitrary number of client dispatch requests to a set of replicas. These replicas implement a service with a state that receives these requests and updates its state accordingly to the operation contained in the client requests. When a sufficient number of replicas transmit specific responses to the client, their invocation returns the result calculated by the service. The purpose of this technique is to consistently grow the state of the service maintained by each replica. To achieve this behavior, certain requirements must be satisfied:

1. Two replicas always start with the same state.

2. If these replicas apply the same operation to a given state, they will get the same state.

3. Both replicas execute the same sequence of operations.

The first two requirements can be easily satisfied if the service is deterministic, but the last one requires a full-order transmitted primitive, which is equivalent to solving the consensus problem.

Within Nomad's context, all replicas execute the state machine replication protocol, receiving and executing the same requests. Using this protocol within Nomad allows us to achieve fault tolerance and implement a sort of decentralized control in a distributed system. Therefore, the purpose of using this protocol in Nomad is the ultimate goal of the replicas performing the same set of operations in the same order, via a consensus algorithm.

3.2.2 Secret Sharing Scheme Protocol

The main objective behind Nomad's system is to ensure confidentiality both in stored data and data in transit. Through a dynamic proactive secret sharing protocol, we preserve the confidentiality of stored data on Nomad through a secret sharing approach.

A secret sharing scheme takes a secret, for example, a message, breaking into multiple parts, known as shares, and distributes the shares among multiple parties. The secret can only be reconstructed if the parties bring together their respective shares. More specifically, if n shares are created at most t shares out of n (i.e., depending on the threshold defined in the algorithm) are needed to reconstruct the secret.

There are variations of secure secret sharing schemes in which, if an attacker has access to a certain number of shares, where their number is lower than the threshold defined by the algorithm, the attacker gains no information about the secret.

A secret sharing scheme is beneficial in the design of Nomad architecture since we wanted to enforce confidentiality on top of stored data. This scheme allows for more secure storage of sensitive data, such as persistent ledger transactions. Since data is separated into different parts, no single point of failure can lead to its loss.

Ultimately, regarding Nomad's environment, each time data needs to be stored, such as persisting a new transaction or a new UTXO, a secret sharing scheme is executed on top of these types of data to create a high level of security, providing then confidentiality. Therefore, each replica receives a different part to be stored. In Nomad's case, the secret (i.e., incoming data) is only reconstructed if the entity that owns that data, for example, a client retrieving a UTXO, wants to retrieve its data. Finally, the different shares that collectively make up the data, are reconstructed on the client side before being delivered to the client.

3.2.3 Homomorphic Encryption Protocol

To achieve confidentiality on data in transit, through a homomorphic encryption protocol we can perform computations on top of encrypted data.

Homomorphic encrpytion is a form of encryption algorithm designed for allowing computations to be performed on encryption data without decrypting the data (i.e., without access to the secret key used to decrypt the data). Most of these computations are mathematical operations, which suits Nomad's cryptocurrency use case. This means that data can be outsourced to different entities without the need to trust in them, because the decryption key is required to access the data. By using homomorphic encryption, the data stays encrypted at all times, which minimizes the likelihood that sensitive information ever gets compromised. The result of such a computation remains encrypted. Homomorphic encryption can be viewed as an extension of public-key cryptography.

The capacity to conduct mathematical operations on encrypted data means that, there is a relationship between plaintexts and ciphertexts. Therefore, if we add two ciphertexts together, the outcome should be

the same as doing the same operations on two plaintexts.

The ability to perform mathematical operations on encrypted data means that there has to be a relationship between plaintexts and ciphertexts. It has to be possible to add or multiply two ciphertexts together and have the same result as performing the same operation on the two plaintexts and then encrypting them. On the other hand, if an attacker can get access to the underground technology in which the homomorphic algorithm is executed on top of, the attacker could understand how the operations are performed, revealing information about the plaintexts, breaking the algorithm. Ultimately, by using a homomorphic encryption protocol we can increase the security of Nomad and use a strong encryption methodology.

Inside Nomad's context, there are specific cases where sensitive data is in transit, such as:

1. Transaction information to be validated. Although a homomorphic encryption scheme isn't used for validating data, but to perform mathematical operations on encrypted data, this scheme plays an impactful role in validating transactions. Therefore, it is during the validation phase of a transaction that encrypted UTXOs are used to verify that they haven't been spent before.

2. The balance of a client to be retrieved. In this example, the homomorphic cipher is just being used to decrypt the incoming UTXO values received by the client.

3. A UTXO to be added to a transaction. In this situation, the homomorphic cipher is used to cipher the UTXO value. Later on, the encrypted value is used during the validation phase.

At last, a homomorphic encryption algorithm is implemented on top of these confidential data to improve integrity and confidentiality. In Nomad, data in transit is never disclosed, only the owner of the data can decrypt and see its content.

3.3 Use Cases

3.3.1 Confidential European Central Bank

The European Central Bank (i.e., ECB) is the central bank responsible for monetary policy of the European Union (i.e., EU) member countries that have adopted the euro currency. The ECB's primary objective is price stability in the euro area. The way Nomad could be adapted within the European Central Bank, is having the replicas working and running in the central bank, and the clients would be each country that have adopted Nomad's currency. For this solution to work, it is necessary to have a large set of replicas, which would scale in order to receive requests concurrently and efficiently. The only difference Nomad offers is the level of confidentiality, where payments and currency usage by each country would be confidential. If the solution were to be implemented, although the European central bank would

know which countries are part of its membership, it wouldn't know how each country uses the currency. The disadvantage that this solution presents would be in terms of regulation. In this situation an operation could be implemented so that a given country would lend its cryptographic keys to the central bank to decrypt and read the contents of its transactions. But this idea would make Nomad's concept irrelevant. Another additional plausible idea would be that, each time a country needed to prove executed transactions, it would provide the unique public keys used as addresses in the transactions. In the end, the central bank would serve as a foundation to hold the network, where countries (i.e., clients) that are part of the network, could use the currency confidentially.

3.3.2 Confidential E-commerce Platform

An e-commerce platform is a software that enables purchasing and selling goods over the internet. Customers come to the online marketplace and purchase products using electronic payments. By receiving the funds, the merchant sends the goods or provides the service. This use case would be easier to adapt to Nomad, since both the vendors and the customers would be the network clients, and the replicas would be the servers running under the e-commerce platform, in order to execute the transactions between the clients. The advantage of implementing such a system would be to provide anonymity to all entities involved in the platform, and at the same time confidentiality, so that the content of the payment isn't revealed. The only proof needed that needs to be kept, is the proof of payment, which is persisted in the ledger of each replica.

3.3.3 Confidential Gaming Platform

A gaming platform is a system or program made for playing video games. Again, this example would be the same as the use case of the confidential e-commerce platform, where Nomad could be implemented so that players could confidentially purchase games or certain items to be used within the games. By using a system that protects the security and privacy of the players' data, more trust can be achieved when games are played. In this case, the clients would be the players, and the replicas would be the servers responsible for executing the transactions between them. In this way, players would have greater freedom. Therefore, the only proof needed is the payments executed within the platform, stored in a safe storage location, which is the ledger.

3.3.4 Confidential Food Delivery Platform

A food delivery platform is an online business that acts as an intermediary between consumers and multiple food facilities to submit food orders from a consumer to a participating food facility and arrange for

the delivery of the order from the food facility to the consumer. This example would be the same as the last two use cases, where Nomad could be used so that consumers could confidentially purchase food. In this case, the clients would be the customers ordering food and the different food facilities, and the replicas would be the servers responsible for executing and storing the transactions between them. In this case, to support a larger network, more replicas would be needed.

3.3.5 Confidential Crypto Exchange Platform

A cryptocurrency exchange platform is a marketplace that provides easy trading of cryptocurrencies for other assets, including digital and fiat currencies. Cryptocurrency exchanges act as an intermediary between a buyer and a seller. These exchanges have the task of connecting cryptocurrency buyers and sellers and support the network. If this use case were applied to Nomad, each time a certain asset was purchased, if a certain customer wanted to use another currency unit instead of Nomad's currency, an operation would have to be applied to make the conversion in real time. Thus, all payments within the platform would be private and confidential. In this context, the clients would be the buyers and the sellers, and the replicas would be responsible for supporting the network and persisting the payments executed between the customers.

Chapter 4

Nomad Implementation

In this chapter, we describe the implementation of our confidential decentralized payment system, Nomad. All the features that are part of Nomad, from the configurations used to achieve the confidentiality we desired to the protocols that are the basis of this cryptocurrency, are going to be detailed and explained in this chapter. All these components together allowed us to develop a cryptocurrency that aims at anonymity, confidentiality, and integrity.

We implemented Nomad in Java on top of COBRA, a proactive secret sharing scheme, which employs a Byzantine consensus protocol to generate secret polynomials with shares distributed to at least $t + 1$ correct servers, being enough for their reconstruction. In addition, COBRA was implemented on top of BFT-SMaRt, a popular BFT SMR library supporting all features needed in practical BFT SMR systems that use a Byzantine consensus protocol. COBRA offers the confidentiality that we wanted for Nomad, in a way of hiding data being stored, whereas BFT-SMaRt delivers a consensus protocol to allow the operation of the state machine replication protocol. Ultimately, Nomad uses Paillier, a homomorphic encryption system that allows operations to be performed on top of encrypted data without being decrypted.

4.1 Overview and Structure

As already explained in the last chapter, Nomad's architecture has clients and servers. In the end, clients don't communicate directly with each other, but through a server. As Nomad follows a client-server model, every time a transaction is sent, it must always be executed by the replicas. Therefore, to recover the amount received by the transaction, the recipient must communicate with the replicas. Consequently, replicas are the middle layer responsible for receiving, processing, and delivering transactions. Clients are just the end users in charge of sending and storing coins. Replicas execute the consensus protocol supplied by BFT-SMaRt to achieve fault tolerance. COBRA and Paillier are only executed on the client side.

Nomad's ledger is immutable and shared between replicas. Replicas are responsible for updating the ledger between them. The biggest difference from a typical blockchain system is that Nomad isn't totally decentralized nor implements a mining process. Nomad follows a more typical private blockchain system, where we have a set of participants that control the network.

At its core, a private blockchain isn't decentralized and is a distributed ledger that operates as a closed, secure database based on cryptography concepts. Within Nomad, not everyone can run a full node, create transactions, or validate/authenticate the system changes.

The method by which Bitcoin and other cryptocurrencies are generated and the transactions involving new coins are verified is known as mining. In Nomad, we have another approach, where we have a set of special nodes responsible to validate transactions. These nodes are called validators, and each time a validator node validates a transaction, it receives a reward. That's how new coins are generated within Nomad.

In the following sections, all the details that were thought out and implemented to build Nomad are explained.

4.2 Wallet Implementation

The wallet was the first feature implemented in Nomad. Each client who joins the network, has a wallet created with a pair of private and a public key. The purpose of having a wallet implemented on Nomad, was to give the client a repository to store its cryptocurrencies. In addition, a wallet was needed because it's from the wallet where a transaction is first initiated.

4.2.1 Master Keys

For the wallet implementation, more specifically, for the creation of the keys, we followed the BIP32 proposal, which is the Bitcoin Improvement Proposal. The BIP32 introduced the standard of Hierarchical Deterministic Wallets. A hierarchical deterministic wallet is a digital wallet commonly used to store the digital keys for holders of cryptocurrencies. Anyone with public and private keys can control the cryptocurrency in the wallet. A seed is used within the wallet to derive many public and private keys. That's how we achieve anonymity in Nomad, by using a different public key every time a transaction is sent.

To create the keys, the Web3j library is used. The Web3j is a lightweight Java and Android library for working with smart contracts and integrating with clients (i.e., nodes) on the Ethereum network. In our case, we aren't interested in using the library for that specific use case, but instead, we are using one of the modules of the library to create the wallet keys. Therefore, we use an algorithm that derives child keys from a parent key following the BIP32 proposal, through an Elliptic Curve key pair. Every time a wallet is created, a BIP32 master keypair is then created. Algorithm 1 shows how we can generate a

master key pair using the BIP32ECKeyPair algorithm provided by the web3j library.

Algorithm 1 Bip32ECKeyPair

procedure generateKeyPair($seed$)
$i \leftarrow hmacSha512(\text{"}Bitcoinseed\text{"}.bytes, seed)$
$il \leftarrow Arrays.copyOfRange(i, 0, 32)$
$ir \leftarrow Arrays.copyOfRange(i, 32, 64)$
$keypair \leftarrow Bip32ECKeyPair.create(il, ir)$
$Arrays.fill(i, (0))$
$Arrays.fill(il, (0))$
$Arrays.fill(ir, (0))$
return $keypair$
end procedure

4.2.2 Child Keys

Each time a transaction is sent from the sender's wallet, a unique public key is created. This public key is derived from the BIP32 master key pair to be used for that specific transaction. All keys generated by a client's wallet are stored in a local keystore. The reason why we store these keys is to, later a client when retrieving its balance can send to the replicas all of the keys that were used on different transactions. Algorithm 2, using the same algorithm and library as Algorithm 1, describes how child keys are generated from a master key pair.

Algorithm 2 Bip32ECKeyPair

procedure deriveKeyPair($master, path$)
$curr \leftarrow master$
if $path \neq null$ **then**
for childNumber in path **do**
$curr \leftarrow curr.deriveChildKey(childNumber)$
end for
end if
return $curr$
end procedure

4.2.3 Transaction Dispatch

It's from the wallet where a transaction is generated. Consequently, the sender:

1. First gathers sufficient funds to create the transaction. If the funds aren't acceptable, the transaction fails.
2. Therafter, the client gathers its UTXOs list, and create an empty inputs list. Then, the UTXOs list is iterated, and for every visited UTXO, a new input is created and added to the list of inputs. If the total sum of all traversed UTXOs is equal to or greater than the value to be sent in the transaction, then no more inputs need to be created. It is at this point that the actual transaction is created with the receiver key, the recipient key, the value to be sent, and the new list of inputs.
3. Finally, the transaction is signed with the private key of the client that is sending the transaction. Consequently, the UTXOs used in that transaction are removed from the client's list of UTXOs.

Algorithm 3 demonstrates how a transaction is created within a client's wallet in Nomad.

4.3 Client Implementation

4.3.1 Node

A node is an alias that represents a client and a validator. In the world of the state machine replication paradigm, a client is responsible for invoking specific operations, that's why we refer to the client as the real client agent. But in our implementation, the client is known as a node, and it extends as a regular client and a validator. In Nomad different nodes are created.

4.3.2 Client

The client represents a regular client within Nomad's environment. This type of client extends from the node's class, and when created, it has implemented three different operations:

1. After the client creates the transaction on its wallet, it sends the transaction to be processed by a validator. Therefore, the client implements an operation of sending the new transaction to be processed and validated by a random validator.
2. Whenever a new client is created, we need to assume that it already possesses some funds in its account. In this case, the client implements an operation that sends its UTXOs to be persisted by the replicas in the UTXOs list maintained by them.
3. Every time a client is sending a new transaction, it must always verify its balance. Here, the client implements an operation of retrieving its balance from the replicas. As already explained, for the client to retrieve its balance, it needs to give a proof to the replicas that it is the rightful owner of the UTXOs.

Algorithm 3 CreateTransaction

```
procedure sendFundsToRecipient(paillier, pubKey, value)
    if balance ≤ value then
        return null
    end if
    inputs ← arrayList
    totalAmount ← 0
    tempUTXOs ← UTXOs
    for elem in tempUTXOs do
        utxo ← elem.key
        totalAmount += utxo.value
        recPubKey ← utxo.pubKey
        encVal ← paillier.encrypt(utxo.value)
        parTrans ← utxo.parent
        key ← utxo.key
        encOutput ← newTransactionOutput(recPubKey, encVal, parTrans, key)
        inputs.add(encOutput)
        if totalAmount ≥ value then
            break
        end if
    end for
    encVal ← paillier.encrypt(value)
    t ← newTransaction(address, pubKey, encVal, inputs)
    to ← newTransactionOutput(t.pubKey, t.value, t.id, rsaKey)
    t.genSignature(privKey))
    for input in inputs do
        UTXOs.remove(input)
    end for
    return t
end procedure
```

4.3.3 Validator

A validator within Nomad is responsible for processing and validating transactions. It is important to mention that they aren't responsible for executing the transactions because that's a task assigned to replicas.

The validator's initial idea is to follow the same concept as having miners in a blockchain environment. The difference is that in Nomad, every time a transaction is issued, a random validator is chosen

to validate the transaction. Upon validating the transaction, the validator responsible for that transaction receives a reward. Then, the validator sends the transaction to the replicas to be executed. On the other hand, miners in Bitcoin are responsible for resolving complex mathematical problems and appending the new block to the blockchain. In Nomad, each transaction is validated individually and sent to the replicas to be appended to the local ledger, storing all of the transactions executed since Nomad began to operate.

Furthermore, when implementing validators, we define that validators have different levels of reputation depending on how many transactions they execute. That being said, the amount of reward that a validator receives after validating a transaction depends on the level that's assigned to that validator. Consequently, if a validator validates a certain number of transactions, its level increase, and so is the amount of reward it's receives.

Within Nomad, the way we choose a validator to validate a specific transaction is purely random. For example, a validator that's assigned with a level lower than a validator which is two or three levels above, both of them have the same probability of being chosen to validate the transaction. A validator's level is just to know the amount of reward a validator is receiving.

4.3.4 Client Controller

The client controller is a class that we define within Nomad's system responsible for controlling the data flow on the client's side.

Every operation that contains data that needs to be sent from the client/validator to the replicas, to be executed, needs to pass through the client controller class. This class is responsible for establishing a connection to the replicas, invoking the operations, requested by the clients/validators. By using a confidential proxy (i.e., delivered by COBRA), a secure connection is established between clients and replicas, in which data in transit is encrypted. It's within this class where specific functions from COBRA are called to execute the secret sharing algorithm upon confidential data. In addition, all data is previously serialized before being sent to the replicas. Later responses, delivered by the service executed by the replicas, are sent to the client controller class, being forwarded to the proper entity (i.e., client or validator).

4.3.5 Storage

As explained before, each client stores locally its master and child keys. For achieving this behavior, we have created two different files. The files are:

- **Keystore File:** file responsible for saving two different secret keys, that is, symmetric keys.
- **Encrypted Log File:** encrypted file accountable to store a client's keys.

Regarding the secret keys, one of the two secret keys is responsible for encrypting all the client public keys, from its master public key to all of its child keys. The second secret key is responsible for

encrypting the client's master private key. Since security is what we want, these two secret keys are password-protected into a Keystore instance file. Therefore, each time a child's public key is generated, the secret key is retrieved from the Keystore, encrypts the child's public key, and ultimately the encrypted child public key is appended to the encrypted file. For that reason, we have implemented two different classes:

- **KeystoreFile class:** class responsible for managing all operations performed on top of the Keystore file.
- **CipherFile class:** class responsible for managing all operations on top of the encrypted file.

All of the operations from the KeystoreFile and CipherFile class are performed within the wallet class.

Considering that this kind of implementation can be sometimes costly and heavy because of writing/reading from a file and performing cryptographic operations, we developed a light version of the client's wallet. Instead of relying on complex operations, we simply stored all keys within the wallet's class. With these comes the paradigm of security vs performance.

4.4 Server Implementation

The server represents the entity responsible for updating Nomad's ledger and for executing and storing transactions. In the end, the server is a replica within the state machine replication paradigm. Replicas are the ones that keep the network running and are accountable for the execution of the different operations within Nomad.

4.4.1 Server

The server represents a replica within Nomad's environment. All operations invoked by a client/validator are executed and persisted by the replicas. Due to the state machine replication protocol, all replicas execute the same set of operations, sequentially, to reach the same state. Replicas execute different types of operations, such as:

- **SEND_TRANSACTION:** Transactions received are parsed and stored in each replica.
- **SEND_UTXO:** UTXOs received are parsed and stored.
- **GET_UTXO:** UTXOs are returned to clients depending on the requested UTXO.
- **PROVE_UTXO_OWNERSHIP:** UTXOs are proven to belong to certain clients.
- **REMOVE_UTXO:** UTXOs are removed from storage.

- **PROVE_BALANCE_OWNERSHIP:** UTXOs are proven to belong to a specific client, in order to calculate its balance.
- **GET_BALANCE:** A client's UTXOs are retrieved to the client to calculate its balance.

Depending on the type of operation executed, sometimes it is important to know the order in which requests are made. Therefore, within the server class, we have implemented two different functions, one for executing the operations in which order is important to maintain, and the other function in which order isn't relevant. For example, operations like persisting transactions and UTXOs are important to maintain order. On the other hand, when a client is retrieving a specific UTXO, a proof is calculated, or getting a client's balance, these types of operations don't need order. In all operations, a response is always sent back to the client. Responses can be different depending on the type of the executed operation. For example, when persisting transactions/UTXOs or removing UTXOs, replicas just send a message stating that the operation was executed successfully. When retrieving a UTXO or retrieving a client's balance, the response differs on the type of use case.

4.4.2 Storage

Regarding replica's storage, it was previously stated that replicas store transactions and UTXOs. The way replicas are storing transactions, is by creating a log file, which is serving as our ledger. The ledger here is just a simple file where every transaction executed, is appended to that file. When a replica receives a transaction, it parses the transaction and writes the content of that transaction to the log file.

Concerning UTXOs, each time a new UTXO is sent by clients to the replicas, the UTXO is persisted in a local list. The objective of having this list is to persist all UTXOs created by all clients within Nomad. Every time a transaction is executed, some UTXOs are removed from the list, and new UTXOs are created and added to that list. The method we are using for a client to return a UTXO from the replicas is through a challenge-response protocol (i.e., this method is explained in the transaction section).

4.4.3 Snapshot

The snapshot feature is important within a distributed system following a state machine replication protocol. To keep Nomad's state operating despite faulty motives, snapshots are necessary because they act as a recovery point for Nomad when a snapshot is taken. For achieving the snapshot functionality we have two functions:

- **Install Confidential Snapshot:** this function installs a confidential snapshot, which is an operation that is executed from time to time to install a new state of Nomad's system.
- **Get Confidential Snapshot:** this function is called from time to time to retrieve the current state of Nomad's system.

4.5 Transaction Implementation

As explained in the Solutions section, a transaction is a significant feature within Nomad. A transaction is a transfer of coins between clients. This type of data is just bits of information passing over the network, which there is value in this type of information. At Nomad, transactions are composed of different parameters, which are explained throughout this section.

4.5.1 Transaction

At Nomad, different transactions contain the same parameters but different values. In this way, we ensure that every transaction performed within Nomad is always different, despite if in some cases the amount of the value being transferred is equal. Within Nomad, a transaction is composed of different inputs and outputs, which we refer to as transaction inputs and transaction outputs.

4.5.2 Transaction Output

Nomad transactions produce outputs which are then stored in a distributed list holding the UTXOs of all clients. Therefore, these outputs are then utilized in subsequent transactions as inputs. Outputs create new coins (i.e., they create new outputs) that can be spent by a client, and these outputs are broadcasted to the replicas so that they recognize that outputs belong to a specific client (i.e., replicas don't know the client's identity). Additionally, these outputs are stored by replicas and later retrieved by different clients to return their balance. For example, during a transaction whereby a client A sends to a client B a certain amount of coins, the UTXOs contained in that transaction are consumed and new UTXOs are created. Then, these new UTXOs are persisted in the replicas distributed UTXOs list, and registered to client Bs wallet address, making B eligible to use the newly UTXOs. Within Nomad, transaction outputs consist of different components:

1. Transaction ID, which is a hash of the transaction output. We concatenate the client's public key, the output value, and the parent transaction ID. Therefore, we apply a hashing algorithm to this concatenated data, where the result is the id.
2. Client's public key, which is gathered from the transaction and identifies that the client is the owner of that specific UTXO.
3. Output's value, which is encrypted using a homomorphic encryption scheme.
4. Parent transaction ID, which is the transaction where this output was created.
5. RSA secret key, which is unique for every output.

Ultimately, when a transaction is executed, a client receives a transaction output that is created after the transaction is validated. So how does a client acting as the recipient in a transaction gather the value to be received?

Within Nomad we have implemented an operation where a client can prove to the replicas that a specific UTXO belongs to it. We achieve this through a challenge-response protocol. This operation is invoked when a validator is validating a transaction or when a client is retrieving their balance. In both cases, it's always the client the entity responsible for receiving the challenge by the replicas, resolving it and dispatch the response to the challenge to the replicas.

About the first use case, the validator is responsible for invoking within the client's class, a private class that has the method to decrypt and resolve the challenge. Therefore, the question arises, does the validator have to know the client's identity? No, because in this scenario, the validator only has to call the correct method and pass the associated parameters, which is the encrypted challenge. Even if the validator has access to the UTXO, it won't be able to forge and create a fake challenge response. Since the validator doesn't have access to the private parameter of the RSA key (i.e., explained in the Solution section), it can't decrypt the challenge. Thus, it should always be the client who decrypts the challenge and sends it to the replicas at the behest of the validator. There is also no issue with the validator seeing the value of the decrypted challenge because the challenge is always different for each UTXO.

Concerning the second use case, the client it's the entity responsible to communicate directly to the replicas. In this case, the validator isn't required to be used as an intermediary.

The challenge sent by the replicas is a one-time security token. This token is a random big number created taking into account the public key presented in the UTXO and some random values. As already explained, the steps to perform this operation are already detailed in the Solution section, more specifically, in the Accounting Model section.

4.5.3 Transaction Input

Transaction inputs consist of UTXOs that are to be sent within a transaction. One and only one input is associated with one output. When a transaction is validated, the transaction inputs are traversed one by one, and each UTXO that belongs to a specific input is consumed. Within Nomad, a transaction input contains an ID and a UTXO. In addition, transaction inputs are only created in a client's wallet, when gathering the sufficient amount of outputs to be inserted in a transaction.

4.5.4 Transaction Signature

A transaction signature is a fundamental component within Nomad's environment. This function serves to verify the integrity and non-repudiation of transactions. Within Nomad, this digital signature is employed through an elliptic curve cryptography scheme.

Signatures used to sign transactions are generated from the client wallet's private key. Therefore, we use an Elliptic Curve Digital Signature Algorithm (i.e., ECDSA), which is a secure and complex public

key cryptography encryption algorithm, used to digitally sign transactions. From the client's master private key, we create an elliptic curve private key and then sign the hashed transaction with the key. Therefore, to verify the signature, from the elliptic curve private key, the elliptic curve public key is extracted and used to verify the digital signature. We are using the Java "ECKey" class, which is implemented by the bitcoinj library, a Java implementation of the Bitcoin protocol. By using a digital signature, we prove that the transaction's sender is the owner of the one-time public key used on that specific transaction.

4.5.5 Transaction Validation

As already explained in the Nomad's solution transaction section, it's the validator's responsibility to process and validate an incoming transaction. It is important to mention that the transaction value is never decrypted, nor is the value of the outputs, allowing us to achieve confidentiality. We achieve this by using the Paillier homomorphic encryption scheme. During the validation process, the validator needs to continuously be communicating with the replicas to verify the integrity of the information received. If any of these steps fails, the transaction is immediately stopped, and an error is returned, and the transaction fails. Ultimately, validators only have access to the public keys used on a transaction. Even if a validator is a malicious agent, these public keys are used only once. Therefore, outputs are only decrypted when returned to the client's wallet. After the transaction is validated, the reputation level of the validator is updated, and then, the transaction is sent to the replicas. Algorithm 4 illustrates how a transaction is validated by a Validator within Nomad.

Algorithm 4 ValidateTransaction

```
procedure processTransaction(p, cc, t, ic, utr, uts)
    if t.signature = false then
        return false
    end if
    numUtxos ← t.inputsSize
    numStoredUtxos ← 0
    for i in t.inputs do
        utxo ← i.utxo
        mo ← methodOne(po, cc, to)
        challenge ← mo.invoke(ic, cc, utxo)
        mt ← methodTwo(pc, bi, to)
        response ← mt.invoke(ic, challenge, utxo)
        storedUtxo ← getUtxo(cc, utxo, response)
        if utxo.value = 0 then
            return false
        end if
        if utxo.id = storedUtxo.id then
            if utxo.value = storedUtxo.value then
                numStoredUtxos ++
            end if
        end if
    end for
    if numStoredUtxos = numUtxos then
        if uts ≠ null then
            sendUtxo(p, cc, uts)
        end if
        sendUtxo(p, cc, utr)
    end if
    for input in t.inputs do
        removeUtxo(cc, input.utxo)
    end for
    processTrans ++
    sendTransaction(cc, t)
    return true
end procedure
```

Chapter 5

Experimental Evaluation

In the following chapter, we give an overview of the results gathered testing Nomad's application. We performed an experimental evaluation of Nomad, aiming to know the overall performance of Nomad, the overall impact of integrating Nomad on top of COBRA, and the costs of having a significant number of confidential operations.

5.1 Setup and Methodology

The experiment was executed in a local cluster composed of 14 physical machines connected through a switched gigabit ethernet. All these machines are Dell PowerEdge R410 servers, with 32GB of memory and two quadcore 2.27 Intel Xeon E5520 processors with hyperthreading (i.e., supporting thus 16 hardware threads). The machines run Ubuntu Linux 20.04.1 LTS and JRE 1.8.

We present two types of experiments where we measured the throughput of Nomad, in a way of knowing how many operations and transactions could be executed per second. Additionally, we also measured the latency between the execution of transactions. For this experiment, we consider small groups of up to ten replicas (i.e., $n <= 3f + 1$), each deployed in a separate physical machine, with the four other machines running up to 1500 clients issuing transactions to the system.

Within Nomad's environment, throughput is a measure of how many actions are completed within a time frame. Transaction throughput refers to how fast Nomad's system processes transactions, including operations like sending a UTXO, getting a UTXO to verify a transaction, removing UTXOs, proving UTXO ownership, etc. That is, all of these operations mentioned are needed together to execute a transaction. For the experiment, the throughput was measured at the replicas side at regular intervals. At the end of the experiment, the file that had the throughputs recorded was parsed and some percentage of the values with greater variance were discarded where the average throughput value was presented. To measure the latency, depending on the number of clients, after the throughput has been stabilized, we always run an additional client to measure the latency between transactions.

To evaluate Nomad's performance, we have developed a script whereby providing a certain number of clients, the script forms groups of client pairs in which they send transactions to each other. To improve the performance of Nomad so that it can execute more transactions in less time, instead of having one pair of clients execute, and then another pair executed, threads are used to speed up this process. In each thread, a client pair sends transactions to each other. Our goal was to have the maximum number of threads running and at the same time have the maximum number of transactions being sent between clients.

In all experiments, to reach the maximum throughput we needed to have the most amount of script instances running in the machines to get the highest number of clients and transactions executing.

5.2 Operations Overall: Updates, Proves, Reads

Our first set of experiments aimed to quantify the performance cost of having confidentiality on a cryptocurrency application, that is Nomad, built on top of COBRA during normal operation. In this experiment, our objective was to measure the throughput of all operations executed together. Update operations involve mostly operations such as writing transactions to a log file, writing/removing UTXOs from a UTXO list maintained by the replicas, etc. Read operations concern mostly fetching UTXOs from replicas. Regarding the number of operations in terms of percentage, 8 operations were performed, in which 50% were read operations and 50% were write operations.Although comparisons with other systems aren't being assembled here, the results of this experiment was to check Nomad's overall performance in terms of the number of operations executed depending on a different number of replicas and clients. Table 5.1 presents the overall throughput of Nomad operations considering values of up to 1500 clients and groups of 4, 7, and 10 replicas.

As the number of replicas increases (i.e., the system tolerates more faults), the throughput of Nomad gradually decreases. This is due to the network becoming a bottleneck during consensus because when there are more replicas, more messages are exchanged and the replicas have to wait for more messages to have a quorum and confirm the operations.

Table 5.1: Throughput and latency for Nomad operations.

Nomad	**n = 4**	**n = 7**	**n = 10**
Throughput	6900	5200	3480
Latency	0.90	1.1	1.2

5.2.1 Transaction Operation

Our second experiment goes towards our evaluation goal which was to measure Nomad's transaction throughput, dividing this section into two different subsections. This second set of experiments, like the first ones, aimed to quantify the performance cost of having confidentiality on Nomad built on top of COBRA.

5.2.1.1 Comparing different underlying systems

We compared Nomad's performance with the underlying systems that are Nomad's foundation such as COBRA and BFT-SMaRt. This experiment compares Nomad's transaction operation throughput with a KV store application built on top of COBRA and a KV store built on top of BFT-SMaRt. The operation isn't the same, but the functionality is similar, persisting data with the intent of updating a data structure stored on the replicas side. Table 5.2 presents the transaction/update throughput considering up to 1500 clients and groups of 4, 7, and 10 replicas.

As expected, BFT-SMaRt (without confidentiality) achieves the best throughput among the compared systems, with more than 15k updates processed per second with four replicas. COBRA throughput, using its secret sharing scheme, compared to BFT-SMaRt decreases due to heavy protocol operations like share verification and resharing. As expected, Nomad throughput, with confidentiality embedded in all its operations, drops especially to costly cryptographic operations such as the use of homomorphic encryption, that is, Paillier. The Paillier cryptosystem is actually slower, primarily because the computations are done modulo n^2, which is twice as many bits as the modulus in RSA (i.e., also used within Nomad's operations) for similar security. As explained before, the choice of using Paillier is to give Nomad the most amount of confidentiality possible, although we decided to give more importance to security over performance. In addition, Nomad's throughput is also slow compared to BFT-SMaRt and COBRA, because COBRA's secret sharing scheme is being used within the transaction operation. Nonetheless, Nomad's throughput still achieves a reasonable throughput, compared to other cryptocurrencies (next section).

As stated, Nomad's throughput for bigger replica groups decreases due to the lack of scalability of BFT-SMaRt consensus protocol, making it a bottleneck.

Table 5.2: Throughput for different systems considering 4,7 and 10 replicas.

System	Limitation	n = 4	n = 7	n = 10
BFT-SMaRt	No confidentiality	15353	10604	8028
COBRA	None	2450	2114	1761
Nomad	None	1002	752	540

Table 5.3: Throughput and latency for different cryptocurrencies.

Cryptocurrency	Throughput (txs/sec)	Transaction Time (sec)
Nomad	540 - 1002	0.89 - 1.32
Monero	1500	60 - 120
Zcash	6 - 25	75
Dash	39 - 56	1 - 2
Beam	17	60

5.2.1.2 Comparing with other cryptocurrencies

We compared Nomad's transaction performance throughput with different cryptocurrencies. This second experiment compares Nomad's throughput and transaction time (i.e, latency) with different cryptocurrencies (i.e., described in the related work section).

Nomad's transaction throughput depends on numerous factors, from the time the transaction is created until it's persisted. Although we didn't test Nomad without confidentiality, Nomad's implementation over COBRA and the usage of the Paillier cryptosystem are the major factor that impacts performance, followed by the number of operations executed between. Unlike other coins, Monero offers the best results in terms of throughput and privacy, relying on strong cryptographic protocols. However, Nomad compared to other private cryptocurrencies offers better results in terms of throughput and transaction time (i.e., although we are persisting our transactions in a log file) as seen in Table 5.3.

Figures 5.1, 5.2, and 5.3 show the throughput and latency of Nomad according to the number of replicas, which varies between 4, 7, and 10 replicas respectively. In Figure 5.1 it is possible to observe the respective throughput and latency with a number of clients ranging from 2 to 1600. With this number of replicas, Nomad achieves the best results, being able to execute 1000 transactions per second. As the number of transactions increases, latency grows with it, which is normal because the number of clients sending transactions increases, so the replicas have more operations to process. In Figure 5.2 the throughput gradually decreases reaching an average execution of 750 transactions per second with a client number that also ranges from 2 to 1600. With the larger number of replicas, it can be seen that the throughput is lower and the latency is higher, this is again due to the scalability problem present in the BFT-SMaRt consensus protocol. Finally, in Figure 5.3, the throughput decreases significantly compared to Figure 5.1, by about a half, which makes the performance of Nomad for larger groups slower. Average throughput of 540 transactions executed per second is achieved with several clients ranging from also 2 to 1600. The number of clients used to achieve the maximum throughput is different for different numbers of replicas. That is, with fewer replicas, for example, 4 replicas, the maximum throughput is achieved with about 1500 clients, while with 10 replicas the maximum throughput is achieved with 1200 clients.

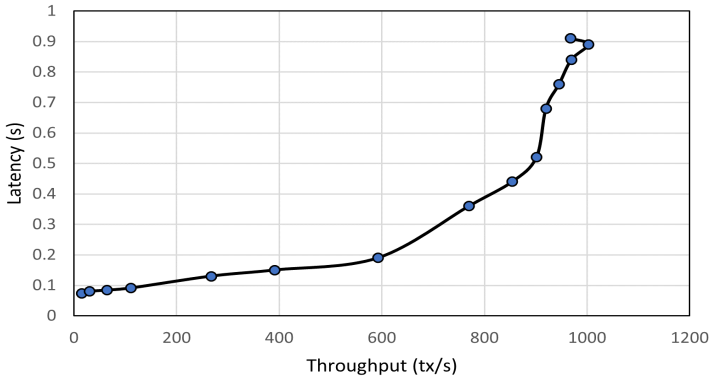

Figure 5.1: Nomad's Throughput for 4 replicas

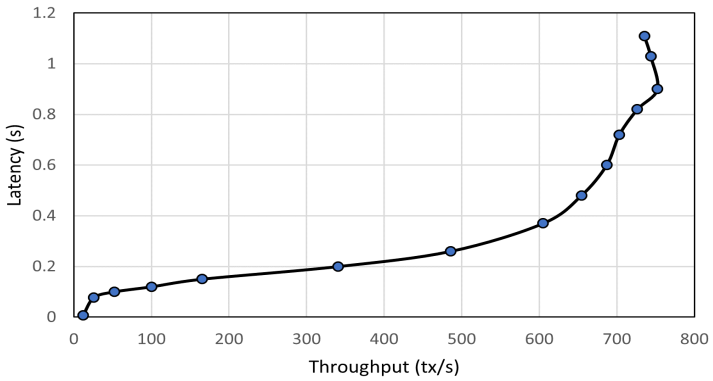

Figure 5.2: Nomad's Throughput for 7 replicas

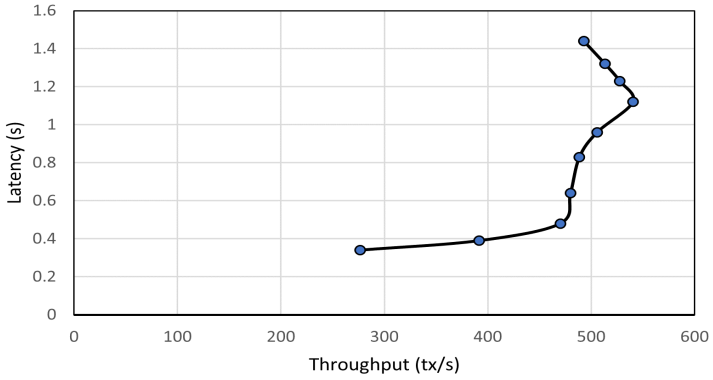

Figure 5.3: Nomad's Throughput for 10 replicas

Chapter 6

Conclusion

This dissertation presents Nomad, a privacy-preserving confidential cryptocurrency application for solving the privacy problem imposed by blockchains nowadays. The solution leverages a state machine replication approach and a secret sharing scheme, offering practically full confidentiality in all its operations. Nomad is a confidential decentralized payment system that can sustain 1000 payments/sec in a batch of 1500 clients while exhibiting an improved transaction time. It can do so by relying upon strong frameworks such as: (1) BFT-SMaRt, a state machine replication library providing a consensus scheme; (2) COBRA, a proactive secret sharing framework delivering confidentiality on top of stored data; (3) Paillier Cryptosystem, a homomorphic encryption scheme which allows confidentiality on top of transactional data.

Nomad didn't reinvent the wheel, we only focused on providing a confidential decentralized payment system following a state machine replication approach and implementing on top of a secret sharing scheme, focused solemnly on providing the best confidentiality we could within Nomad's operations. We also introduced a way of modifying Paillier's algorithm, offering a way to compare encrypted data. Yet, although performance wasn't our top main priority, we could achieve results that compared to known private cryptocurrencies are more promising.

We conclude that Nomad's solution is mostly right in terms of objectives, and when it functions well, provides good security properties such as confidentiality, anonymity, privacy, integrity, and availability.

6.1 Future Work

For future work, we believe that the solution should be improved, especially in the way where some operations are conducted. For example, the way how proves are conducted within Nomad between clients and replicas to return UTXOs. One possible solution could be to implement zero-knowledge proofs on top of Nomad to increase security. Another operation to improve is the anonymity of clients involved in a transaction. Although our solution works well, maybe a more robust implementation should be addressed [31]. One possible solution could be the use of ring signatures like Monero. Another solution passes by

the use of blind signatures [32], which is a kind of digital signature in which the message is disguised before it is signed so that even the signer itself won't learn the message content.

Furthermore, addressing new features that may be added in the future to Nomad's solution to improve the system and make it more realistic could be:

- An implementation of a blockchain platform, where we could organize transactions in blocks and implement a simple mining algorithm on top of COBRA and BFT-SMaRt.
- To increase Nomad's functionality, smart contracts [33] could be developed, added and be used for different purposes such as: (1) an authentication layer to authenticate clients into Nomad; (2) for financial purposes like trading, investing, lending, etc.
- Develop a prototype of a wallet application, web or desktop based, in which real clients could use this application to store their coins and send transactions between clients across the network.

References

[1] Wai Wu and Brett Falk. ”How the Blockchain Revolution Will Reshape the Consumer Electronics Industry”. 07 2017. 1

[2] Mostafa Al-Emran Mohammed AlShamsi and Khaled Shaalan. ”A Systematic Review on Blockchain Adoption”. 04 2022. 1

[3] Burcu Sakız and Aysen Hic Gencer. ”Blockchain Technology and its Impact on the Global Economy”. 10 2019. 1

[4] Tobias Scherer Elli Androulaki, Marc Roeschlin and Srdjan Capkun. ”Evaluating User Privacy in Bitcoin”. 04 2013. 1

[5] S. Nakamoto. ”A peer-to-peer electronic cash system”. *[Online]*, 2009. 2, 9, 27

[6] Harvey J and Branco-Illodo. ”Why Cryptocurrencies want Privacy: A Review of Political Motivations and Branding Expressed in ‘Privacy Coin’ Whitepapers”. 12 2020. 2

[7] Wai Wu and Brett Falk. ”Limitations of Privacy Guarantees in Cryptocurrency”. 04 2018. 2, 13

[8] George Kappos, Haaroon Yousaf, Mary Maller, Sarah Meiklejohn, and University College London. ”An Empirical Analysis of Anonymity in Zcash”. 08 2018. 2, 11

[9] Bitcoin Insider. ”Masternode Running 101: Advantages and Disadvantages of Staking Tokens”, 2018. 2

[10] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, Athanasios Xygkis, Matej Pavlovic, Petr Kuznetsov, Yvonne-Anne Pignolet, and Dragos-Adrian Seredinschi amd Andrei Tonkikh. ”Online Payments by Merely Broadcasting Messages”. 2020. 2, 3

[11] Alysson Bessani, Eduardo Alchieri, Joao Sousa, Andre Oliveira, and Fernando Pedone. ”From Byzantine Replication to Blockchain: Consensus is only the Beginning”. 2020. 2, 3

[12] A. Bessani, J. Sousa, and E. E. P. Alchieri. ”State machine replication for the masses with BFT-SMaRt”. *DSN’14*, 2014. 3, 5, 20

[13] COBRA: Dynamic Proactive Secret Sharing for Confidential BFT Services. 2021. 3, 5, 16, 21

[14] Alysson Neves Bessani and Eduardo Alchieri. ”A Guided Tour on the Theory and Practice of State Machine Replication”. 11 2014. 3, 15

[15] Fred B. Schneider. ”Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. 07 2016. 3, 15, 20

[16] Nicolas van Saberhagen. ”CryptoNote v 2.0”. *DSN’14*, 2014. 6

[17] Ronald L. Rivest, Adi Shamir, and Yael Tauman. ”How to Leak a Secret”. 2001. 6

[18] Shen Noether. ”RING CONFIDENTIAL TRANSACTIONS”. 2016. 7

[19] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. ”An Empirical Analysis of Traceability in the Monero Blockchain”. 4 2018. 8

[20] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. ”Zcash Protocol Specification”. 19 2018. 8

[21] Christian Reitwiessner. ”zkSNARKs in a Nutshell”. 12 2016. 8

[22] Evan Duffield and Daniel Diaz. ”Dash: A Privacy-Centric Crypto-Currency”. 11 2017. 11

[23] gmaxwell. ”CoinJoin: Bitcoin privacy for the real world”. 08 2013. 12

[24] LATANYA SWEENEY. ”k-Anonymity: A model for protecting privacy”. 05 2002. 12

[25] A. Romanov. ”BEAM - THE SCALABLE CONFIDENTIAL CRYPTOCURRENCY”. 01 2019. 13

[26] Tom Elvis Jedusor. ”MIMBLEWIMBLE”. 07 2016. 14

[27] Leslie Lamport, Robert Shostak, and Marshall Pease. ”The Byzantine Generals Problem”. 07 2016. 16

[28] Miguel Castro and Barbara Liskov. ”Practical Byzantine Fault Tolerance”. 02 1999. 20

[29] Tobias Distler and Friedrich-Alexander. ”Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective”. 02 2021. 20

[30] Yu Chen, Xuecheng Ma, Cong Tang, and Mang Ho Au. ”PGC: Decentralized Confidential Payment System with Auditability”. 09 2020. 22

[31] Matthew Green and Ian Miers. ”Anonymous Payment Channels for Decentralized Currencies”. 10 2017. 59

[32] David Chaum. ”Blind Signatures for Untraceable Payments”. 06 1983. 60

[33] Massimo Bartoletti and Livio Pompianu. ”An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns”. 03 2017. 60