# Digital Asset Management via Distributed Ledgers

*Dimitris Karakostas*


Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
The University of Edinburgh

2021

*"One must imagine Sisyphus happy."*

*Albert Camus*

# Abstract

Distributed ledgers rose to prominence with the advent of Bitcoin, the first provably secure protocol to solve consensus in an open-participation setting. Following, active research and engineering efforts have proposed a multitude of applications and alternative designs, the most prominent being Proof-of-Stake (PoS). This thesis expands the scope of secure and efficient asset management over a distributed ledger around three axes: i) cryptography; ii) distributed systems; iii) game theory and economics.

First, we analyze the security of various wallets. We start with a formal model of hardware wallets, followed by an analytical framework of PoS wallets, each outlining the unique properties of Proof-of-Work (PoW) and PoS respectively. The latter also provides a rigorous design to form collaborative participating entities, called stake pools. We then propose Conclave, a stake pool design which enables a group of parties to participate in a PoS system in a collaborative manner, without a central operator.

Second, we focus on efficiency. Decentralized systems are aimed at thousands of users across the globe, so a rigorous design for minimizing memory and storage consumption is a prerequisite for scalability. To that end, we frame ledger maintenance as an optimization problem and design a multi-tier framework for designing wallets which ensure that updates increase the ledger's global state only to a minimal extent, while preserving the security guarantees outlined in the security analysis.

Third, we explore incentive-compatibility and analyze blockchain systems from a micro and a macroeconomic perspective. We enrich our cryptographic and systems' results by analyzing the incentives of collective pools and designing a state efficient Bitcoin fee function. We then analyze the Nash dynamics of distributed ledgers, introducing a formal model that evaluates whether rational, utility-maximizing participants are disincentivized from exhibiting undesirable infractions, and highlighting the differences between PoW and PoS-based ledgers, both in a standalone setting and under external parameters, like market price fluctuations. We conclude by introducing a macroeconomic principle, cryptocurrency egalitarianism, and then describing two mechanisms for enabling taxation in blockchain-based currency systems.

# Acknowledgements

First and foremost, this thesis owes its existence to my advisor, Prof. Aggelos Kiayias. His calming presence, steady guidance, profound advices, and contagious passion for the scientific process have been staples of my journey through academia. He allowed me to work independently and on my preferred pace, putting trust in my abilities, always offering opportunities to better myself and never making me feel pressured. For all these and more, I am forever grateful.

Completing this thesis would have been impossible without the continuous discussions and exchange of ideas with multiple people. My co-authors and collaborators Nikos Karayannidis, Thomas Zacharias, Andriana Gkaniatsou, Myrto Arapinis, Christos Nasikas, and Kostis Karantias have been extremely helpful in tackling the questions we faced and helping me learn in practice how science is conducted. I am also grateful to Drs. Vesselin Velichkov and Arthur Gervais, for taking the time to review this thesis, posing intriguing questions during the viva voce examination, and providing me with constructive feedback. I consider myself lucky to have been member of a research group alongside Markulf Kohlweiss, Vassilis Zikas, Michele Ciampi, Thomas Kerber, Orfeas Stefanos Thyfronitis Litos, Giorgos Panagiotakos, Hendrik Waldner, Christian Badertscher, Lamprini Georgiou, Misha Volkhov, Lorenzo Martinico, Muhammad Ishaq, Yun Lu, Aydin Abadi, and Yiannis Tselekounis, who were daily companions in discussions that, more often than not, culminated in undeniably entertaining arguments. I am particularly thankful to IOHK, for funding my numerous conference travels all around the globe, and Mirjam Wester, for helping me tackle the oh-so dreaded bureaucracy. Finally, I feel the need to give special thanks to Mario Larangeira, who was a constant collaborator for the duration of my thesis and never failed to lift my spirits, and Dionysis Zindros, for teaching me the marvels of technology and motivating me by example.

To the extent that this thesis is a product of mine, I am a product of the relationships with the people closest to me. My mom, dad, and sister have, simply put, shaped who I am today more than any other people. Christos, Panagiotis, and Tea have been part of my life for more years than not; in our relationship, lies my definition of friendship. I was fortunate enough to have shared my adult life with friends in Thodoris, Konstantina, Nikolas, Fenia, Christos, Thanos, Foteini, Georgia, and Dorothea, who always made my living in Greece and abroad as relaxing and fun as I could hope for.

In culminating this journey, any attempt to compress in a few sentences what Mirella means to my life is futile; all I can offer is a humble thank you.

This work is licensed under the Creative Commons Attribution-Non-Commercial-Share-Alike 4.0 International License; to view a copy of this license, you may visit `https://creativecommons.org/licenses/by-nc-sa/4.0/`. This license is in addition to the copyright granted to the University of Edinburgh for the publication of this thesis, and does not preclude granting additional licenses. The LaTeX source files of this thesis are public and are available at `https://github.com/dimkarakostas/phd-thesis` under the same license as all parts authored by Dimitris Karakostas.

# Declaration

I hereby declare this thesis was written by myself to an overwhelming degree. A minority of the writing was originally composed by co-authors, namely Aggelos Kiayias, Mario Larangeira, Thomas Zacharias, Nikos Karayannidis, Andriana Gkaniatsou, Christos Nasikas, Dionysis Zindros, and Myrto Arapinis for joint research papers. Neither this work, nor any part thereof, has been submitted for any other degree or professional qualification.

*Dimitris Karakostas*

# Lay Summary

Distributed ledgers have been hailed as the "next big thing" for more than a decade. Bitcoin, which established the blockchain paradigm, paved the way for a technology that touches on a multitude of interdisciplinary domains. However, many subsequent blockchain systems have inherited various deficiencies of Bitcoin, most importantly in terms of energy consumption, the number of computations and storage needed to maintain the ledger, and the incentives offered to the ledger's maintainers. This thesis uses tools of cryptography, computer security, distributed systems, and economics to explore how users can participate in the maintenance of digital assets via distributed ledgers in a secure, efficient, and sustainable manner.

# Table of Contents


**1 Introduction** **1**

1.1 Motivation and Contributions . . . . . 6

1.2 Publications . . . . . 8

**2 Background** **10**

2.1 Cryptographic Primitives . . . . . 11

2.1.1 Cryptographic Hash Functions . . . . . 11

2.1.2 Digital Signatures . . . . . 12

2.1.3 The Universal Composability Framework . . . . . 13

2.2 Distributed Ledgers . . . . . 14

2.2.1 Consensus . . . . . 14

2.2.2 Reliable Broadcast . . . . . 15

2.2.3 Distributed Ledger . . . . . 15

2.3 Bitcoin and Blockchains . . . . . 16

2.4 Literature Overview . . . . . 20

2.4.1 Bitcoin Formal Models . . . . . 20

2.4.2 Proof-of-Stake Protocols . . . . . 20

2.4.3 Blockchain Incentives . . . . . 21

**3 Formalization of Hardware Wallets** **22**

3.1 Formal Model of Hardware Wallets . . . . . 24

3.2 The Ideal Functionality . . . . . 30

3.3 The Real-World Hybrid Setting . . . . . 32

3.4 Security Analysis . . . . . 35

3.5 Product evaluation . . . . . 40

**4 Account Management in Proof-of-Stake Ledgers** **42**

4.1 General Desiderata . . . . . 45

4.2 Address Malleability . . . . . 47
4.3 The Core-Wallet Functionality . . . . . 54
4.4 The Generic Core-Wallet Protocol . . . . . 58
4.5 Security Analysis . . . . . 58
4.5.1 Properties of the Generation Algorithms . . . . . 58
4.5.2 Security in the Sink Malleable Setting . . . . . 62
4.5.3 Security in the Fully Malleable Setting . . . . . 68
4.5.4 Attacking the Malleable Protocol in the Non-Malleable Setting . 68
4.6 PoS Addresses: Construction and Recovery . . . . . 70
4.6.1 Address Types and their Attributes . . . . . 70
4.6.2 Malleable Addresses . . . . . 73
4.6.3 A Posteriori Malleable Addresses . . . . . 74
4.6.4 Sink Malleable Addresses . . . . . 76
4.7 The Proof-of-Stake Wallet . . . . . 78
4.7.1 Payment . . . . . 78
4.7.2 Stake Pool Registration . . . . . 79
4.7.3 Delegation . . . . . 79
4.7.4 Protocol Participation . . . . . 81
4.7.5 Security in the Presence of Stake Pools . . . . . 83
4.7.6 Modes of Execution . . . . . 85
4.8 Discussion . . . . . 86

**5 Collective Stake Pools** **88**
5.1 Desiderata . . . . . 90
5.2 Execution Model . . . . . 91
5.2.1 Weighted Threshold Digital Signatures . . . . . 92
5.2.2 Transactions, Blocks, and the Global Ledger . . . . . 93
5.3 UC Weighted Threshold Signature . . . . . 95
5.4 The Collective Stake Pool . . . . . 95
5.4.1 Hybrid Protocol Execution . . . . . 97
5.4.2 Part 1: Stake Pool Management . . . . . 97
5.4.3 Part 2: Participation in Consensus . . . . . 99
5.5 Security Analysis . . . . . 104
5.6 Incentives Analysis . . . . . 109

**6 Efficient Global State Management 111**
6.1 A UTxO Model . . . 113
6.2 Transaction Optimization . . . 116
6.2.1 Transaction Logical Operators - Ledger State Algebra . . . 117
6.2.2 A Transaction Optimization Framework . . . 119
6.2.3 Transaction Optimization Techniques . . . 120
6.2.4 The Transaction Optimization Problem . . . 124
6.3 State Efficiency in Bitcoin . . . 127
6.3.1 A State Efficient Bitcoin . . . 130

**7 Blockchain Nash Dynamics 132**
7.1 The Setting . . . 136
7.1.1 Network Model . . . 136
7.1.2 Approximate Nash Equilibrium . . . 137
7.2 Compliance Model . . . 137
7.2.1 Basic Notions . . . 138
7.2.2 Compliant Protocols . . . 139
7.3 Blockchain Protocols . . . 141
7.3.1 The Setting . . . 141
7.3.2 Utility: Rewards and Costs . . . 143
7.4 Fair Rewards . . . 144
7.5 Block-Proportional Rewards . . . 149
7.5.1 Bitcoin . . . 150
7.5.2 Proof-of-Stake . . . 155
7.6 Externalities . . . 161
7.6.1 Utility under Externalities . . . 161
7.6.2 Compliance under Externalities . . . 163
7.6.3 Attacks and Market Response . . . 165
7.6.4 Penalties . . . 167

**8 Macroeconomic Principles 171**
8.1 Cryptocurrency Egalitarianism . . . 171
8.1.1 PoW vs. PoS . . . 173
8.1.2 A Formal Model of Crypto-Egalitarianism . . . 175
8.1.3 Discussion . . . 178
8.2 Tax Applications of Programmable Money . . . 179

8.2.1 Desiderata . . . . . 180
8.2.2 Tax Auditable Distributed Ledger . . . . . 181
8.2.3 A Tax-Auditing Extension for Provisions . . . . . 183

**9 Conclusion** **187**

**Bibliography** **190**

# Chapter 1

# Introduction

Atop Castle Rock, a land which has been occupied by humans since the Iron Age, stands Edinburgh's castle. The most famous attraction in a city full of those, the castle used to serve as the royal residence and repository of Scotland's official documents, since as early as the High Middle Ages. Initially, it comprised of only a fortified keep. This inner sanctum, a small and square stone building, was the place where, at each point in time, the ruler decided the fates of Scotland and its people. Centuries passed and towers were added, houses were constructed, extra lines of fortification were built. The confines of the castle now housed the king's advisors, that is the people who had gained his trust and helped him rule the land. Outside the castle's gates, a society formed, as potters supplied the castle with ceramics and stoneware, masons built or restored walls and roads, peasants and merchants travelled from across the land to trade in food and other products. These people lived on the periphery of the castle and were allowed to enter its premises by permission only, either from the king or his closed circle of confidants.

Eventually, the castle, both as a physical space and an idea, became too limited for the modern times. With the degradation of feudalism and the industrial revolution, the castle was no longer the fortified home of the ruler. More and more people gradually started participating in the governing of the country. First, the king was the lone ruler; then, he presided an inner council of hand-picked members; following, this council comprised of various lords and members of the upper class. Starting from $1802$ and the first British general elections, a few thousand aristocrats would elect a government. This was followed by granting voting rights to all male home owners and, eventually, universal suffrage for all citizens. Nowadays, the Scottish Government is elected by an open, fluid body of people, consisting of all *residents* of Scotland aged $16$ and above.[1]

[1] The above timeline should be perceived more as an allegory for the paragraphs to follow, rather than

## From a Digital Fortress to a Brave New World

In the course of human history, many devices vie for the title of "first computer". The abacuses of ancient Babylonia and Greece and the Antikythera mechanism are some primitive examples, while Charles Babbage's Analytical Engine is typically hailed as the first mechanical computer. The first design of a *modern* computer was given by Alan Turing [Tur37], with two marvelous machines paving the way: ENIAC and the Manchester Baby, the first Turing-complete and stored-program computers respectively.

In the $30$-odd years following the publication of Turing's pioneering work, computers became smaller, more efficient, and more versatile. A major breakthrough came in the $1960$s with the usage of direct-access storage, like magnetic disks, which resulted in the introduction of *databases*. This new technology enabled a more flexible, shared, and interactive storage and processing of information. Following, a rich body of literature focused on designing database standards and applications, like the navigational and relational database management systems and the SQL. Nonetheless, a ubiquitous element of this era was the centralized operation of database systems. From shared-time servers to personal computers, the designs assumed a single entity with complete control on data management. However, soon a need emerged to perform correctly in the presence of faults, be it benign, such as hardware failures, or malicious, such as bad actors trying to undermine the system.

The work of Lamport, Shostak, and Pease during the early '$80$s introduced the consensus problem to the world [PSL80a, LSP82]. As the title of their seminal paper suggests, their work considered a set of computers that should reach agreement, on the content of the information shared and the operations performed, in the presence of faults. To this day, the following beautiful analogy of the Byzantine generals, devised in that initial work, remains the best way to describe the consensus problem.

Imagine a group of generals of the Byzantine army, who siege a city and must decide whether to attack at a pre-defined time. Some generals might prefer to attack, others may not. Crucially, all generals should agree on a common decision, for divided troops are bound to be defeated; thus, splitting the forces is far worse than either attacking or retreating in a coordinated manner. The decision is made via remote voting, as the generals cannot meet in person. However, there is a caveat. Some generals may have defected to the enemy, so they might vote for a suboptimal strategy or, more importantly, vote selectively. For example, if the group consists of $5$ generals, $2$ of which are

a scientific exploration of Scotland's history and politics; for the latter, the reader may advise textbooks and works specializing on the matter, such as [Mac19, Bam14].

in favor of attacking while $2$ are against, the fifth — corrupted — general may send an "attack" vote to the former and a "retreat" vote to the latter; as a result, the four — honest — generals would split. Even worse, since the generals are physically separated and deliver their votes via messengers, it is possible that some messages are delayed or even fail to be delivered.

In distributed computing, processors take the place of generals and networks take the place of messengers. The system's designer defines a protocol $\Pi$ such that, if a processor $\mathcal{P}$ that follows $\Pi$ outputs a value $x$, then every other processor that follows $\Pi$ also outputs $x$. A well-known impossibility result showed that, if more than half of the processors are faulty, no protocol can solve the consensus problem. Subsequently, various protocols were proposed as solutions, each achieving different complexity bounds under various network assumptions [GK20b]. Nonetheless, all of these protocols were *federated*, thus restricting how many and which parties could participate. Without such restriction, it was unclear how an attacker could be prevented from mounting a *Sybil attack*, i.e., create a multitude of fake identities to gain an artificial majority.

$30$ years later came Bitcoin [Nak08a], which introduced "Nakamoto consensus". The major achievement of Bitcoin is solving [GKL15] the consensus problem in a completely open manner, i.e., without any restriction on who can participate when. It achieved this by combining two pre-existing elements, (a) a linked chain of data, (b) Proof-of-Work (PoW), resulting in a protocol whose quality was higher than the sum of its parts.

The former, which has since been dubbed a "blockchain", is a special database, which consists of an append-only log of data chunks ("blocks"). In this database, nothing gets deleted, i.e., a party can only add information, and, at any point in time, each processor outputs an ordered log of published data.

The latter (PoW) is a cryptographic mechanism, via which a party proves to others that it has performed a certain amount of computational effort. Invented by Dwork and Naor [DN93] and formalized (and christened) by Jakobsson and Juels [JJ99], PoW was originally proposed as a deterrent against Denial-of-Service (DoS) attacks and email spamming. However, Bitcoin's designer(s) repurposed it to counter sybil attacks. In the new setting, each unit of computational power is an individual party and, to gain a majority (and break the system), an attacker should possess more computational power than all honest participants *combined*.

With this new protocol an evolution had occurred, from centrally-controlled computer systems to completely open ones, much like the evolution described in the begin-

ning of this chapter. Now, one could design an application that retains as high a degree of decentralization as one could hope for. However, an issue still remained. PoW requires from each party to repeatedly perform a simple task (see Section 2.3 below). Still, simple as it is, each computation consumes some amount of energy. To perform millions, trillions, or more such computations per second comes at great cost, that is the consumption of significant amounts of energy. So, what kind of application could be built on this new paradigm and why would people care to pay the price of the costly PoW algorithm?

## Freakonomics

To answer this question, Bitcoin's designers turned to classical economics, particularly the concept of *utilitarianism*. This notion came to prominence by Jeremy Bentham who, in his classic "Introduction to the Principles of Morals and Legislation", defined utility as "that property in any object, whereby it tends to produce benefit, advantage, pleasure, good, or happiness" [Ben70]. Based on this idea, late $19$th century economists devised the image of *Homo Economicus* [PSP+71], a being that consistently acts rationally and optimally, in order to increase its self-centered utility. People, Bitcoin's designers argued, are driven by the pursuit of wealth. Therefore, to convince them to participate in this new system, they should be compensated. Consequently, the first application to be built on this new paradigm was a financial one: the Bitcoin cryptocurrency (BTC).

Bitcoin is undoubtedly a product of its era. Although the real identity of Satoshi Nakamoto, its creator(s), remains unknown to this day, we can safely assume that, by $2008$, they were at least in their early $20$'s. Therefore, they grew up in, and were nurtured by, the globalized, financial, neoliberal capitalism. The platform on which Bitcoin's design was initially published, an online cryptography mailing list [Nak08b], suggests that Nakamoto were part of the discussion on Internet civil liberties and its culmination in the "cypherpunk" movement [Gre12, Lev01, AAMMZ12, Man11]. Hence, individual liberty against an oppressive state became the compass of Bitcoin's existence [Gol16]: i) on the computer science side, Bitcoin was designed as a global, censorship-resistant system, that can withstand attacks from any single entity with less power than the aggregate power of its participants; ii) on the economics side, it was based on the ideas of the Austrian School and the ideal of the gold standard, arguing for deflation and algorithmically-controlled, a-political money.

As a result, the novel blockchain-based paradigm was used as the database of a finan-

cial system with the following characteristics. The blockchain acts as a log of monetary transactions. The unit of transactions is a new currency, the Bitcoin (BTC). The parties that maintain the blockchain are called "miners" (in a not at all subtle nod to the gold standard). For each block that a miner $\mathcal{P}$ adds to the blockchain, $\mathcal{P}$ is rewarded with a certain amount of newly-issued BTC. The total amount of BTC in circulation is capped, converging over time to $21$ million. To transact with BTC, the sender pays some fees, which are awarded to the miner that includes the transaction in their created block.

Bitcoin's economic design has various interesting implications. The first type of rewards, newly-issued coins, deteriorates significantly over time, thus giving a disproportionate advantage to early participants. Additionally, it incentivizes miners to keep producing blocks, even if nobody uses it (as is typically the case with new systems). The second type, transaction fees, incentivizes miners to include as many transactions as possible to their blocks. However, Bitcoin restricts the size of each block to $1$MB. This bound creates a competitive market for space in each block, between miners (i.e., "sellers" of the space) and users (i.e., "buyers"). Consequently, if adoption of the system increases and more transactions are performed, users "compete" for the (limited) block space, so the fees increase and the miners are compensated more generously. Finally, if more people were to use the system for monetary transactions, that is if a Bitcoin-based economy of goods developed, the upper-bound of $21$ million would enforce deflation, as each product would cost less in BTC (conversely, each BTC would be worth more). As a result, the system encourages people to hoard BTC, rather than transact with them.

During the first $11$ years of Bitcoin's existence, these implications often manifested in practice in spectacular fashion. Bitcoin was initially presented as a cheap method of transacting, often compared to overseas wire transfers. Indeed, for the greater part of this period, a single transaction's fees averaged a few USD cents [Bit21a]. However, during periods of intense use, when many users tried to transact as fast as possible, the average fees increased to as much as $\$60$. As Bitcoin gained fame (and notoriety), its price increased, at times in clear bubble-like rate. However, even at the peak of its mainstream adoption, as few as $2160$ entities (represented by unique addresses on Bitcoin's ledger) controlled $42.17\%$ *of the total BTC in circulation* [Bit21b], a level of wealth centralization unprecedented in real-world economies. Consequently, almost all of Bitcoin's usage thus far has been in: i) savings and financial speculation (due to its deflationary nature), or ii) illegal trades (due to its censorship resistance) [Ger17].

More importantly, Bitcoin is quickly turning into an environmental disaster. As BTC's price increased, partially due to widespread and questionable speculation [GS20], more

people would join the (profitable) mining business. As more and more people started mining, the network's aggregate PoW computations, and the total energy consumed by the system, increased. Since the end of $2017$, when BTC's price bubbled to thousands of USD, Bitcoin's energy consumption has reached ridiculous levels. In an era when the planet is *on fire* [Kle20], the carbon footprint of Bitcoin is comparable to that of the Republic of Serbia, the $CO_2$ emissions of processing *a single Bitcoin transaction* are equivalent to processing $1,808,913$ VISA transactions, and the energy corresponding to this single transaction could power a USA household for $58$ days [Dig21].

## 1.1 Motivation and Contributions

Evidently, although the distributed ledger used by Bitcoin is a marvelous *technical* achievement, the application built on top of it is rather flawed. Specifically, as Bitcoin is a somewhat simple protocol, aimed at working on a basic level for its core application, i.e., transacting, there are various avenues of research on the usability, efficiency, and ecological and economic sustainability of distributed ledger-based applications.

Regarding usability, Bitcoin users control their assets via a cryptographic key. Using this key, they can only perform limited actions, such as transferring assets between addresses, possibly under some very basic conditions. Importantly, the ledger is immutable; once a transaction is complete, it is impossible to revert it. As a result, if the controlling key is compromised, the assets are at risk. This makes Bitcoin and cryptocurrencies a prime target for criminals, as is evident by the — almost daily — reports of thefts that are worth thousands of dollars.

Regarding efficiency, a decentralized system should be carefully designed to enable hundreds or thousands of participants to quickly process data under reasonable hardware requirements. Bitcoin requires of users and miners to store two data objects. First, the log of transactions, an ever-increasing list of historical data. Second, the state of the systems, i.e., a mapping of addresses and the amount of bitcoins each owns. As time passes, both objects increase. Therefore, if this trend continues, the system is bound to be unmaintainable without using specialized, expensive hardware.

From an ecological point of view, Bitcoin is clearly unsustainable. In the past years, this has become common knowledge, with alternative, more sustainable designs being proposed, of which Proof-of-Stake (PoS) is the most prominent. PoS is a variation of PoW where, instead of computational power, users are identified by stake in the system, i.e., the assets that they own. Consequently, the energy footprint of PoS-based ledgers

is minuscule, offering a rather appealing alternative to Bitcoin.

From an economic point of view, both in theory and in practice, Bitcoin has proven unable to sustain a real-world, productive economy. Instead, as shown above, it has been used mostly in gray areas of speculation and dubious transactions. A particularly interesting question then is how distributed ledgers could be better utilized, by applications that solve real problems and which are managed in an open, democratic manner by the whole of society, instead of a small group of insiders and early adopters.

Based on these principles, this thesis makes incremental steps in improving how digital assets, which are maintained via a distributed ledger, are designed, managed, and used. The contributions are split into the following main chapters, each based on one of the research papers output during the composition of the thesis:

- **Formalization of Hardware Wallets**: Chapter 3 analyzes hardware wallets, i.e., hardware modules that offer state-of-the-art security in storing and managing cryptocurrencies. We present a formal model of expressing the necessary properties of hardware wallets, which is then used to analyze a number of commercial products, identifying, in some cases, potential hazards.

- **Account Management in Proof-of-Stake Ledgers**: Chapter 4 articulates the necessary properties of managing assets on PoS-based ledgers. In doing so, it first identifies a malleability attack on cryptocurrency addresses, which is applicable against real-world deployed systems, and then proposes a formal model that captures the security of Proof-of-Stake wallets, which enables participation in the ledger, either as a user or a maintainer of the system.

- **Collective Stake Pools**: The model of Chapter 4 defines participation in the consensus mechanism of a Proof-of-Stake ledger via stake pools, i.e., collaborative entities of multiple parties. However, these pools are presumably operated by a single party. Chapter 5 relaxes this centralization assumption by introducing *Conclave*, a design that enables a group of parties to jointly manage a stake pool, in a competitive and highly efficient manner.

- **Efficient Global State Management**: The transactions that are created by a wallet are stored in a global state, which is shared across all participants. As such, efficient state management is imperative, if the system is to scale to thousands or millions of users. Chapter 6 introduces a framework for constructing such efficient transactions and describes how to incentivize both users (e.g., operators

of hardware wallets, as in Chapter 3) and maintainers (e.g., stake pool operators, as in Chapters 4 and 5) to avoid the unnecessary bloating of the shared state.

- **Blockchain Nash Dynamics**: Cryptographic treatment assumes that parties act either faithfully or arbitrarily and (possibly) maliciously; to evaluate whether it is in the parties' best interest to follow a protocol, we turn to game theory. Chapter 7 builds on the results of Chapter 6 and explores under which conditions parties remain compliant, i.e., do not exhibit a well-defined problematic behavior, even if slightly diverging from the prescribed protocol. We evaluate large families of deployed protocols and offer both positive and negative results, which showcase the differences between PoW and PoS, as well as the limits of system design in the presence of external market factors.

- **Macroeconomic Principles**: Chapter 8 explores some macroeconomic properties of blockchain-based financial systems. First, we show that wealth redistribution from large to small capital owners is impossible in anonymous decentralized financial systems. Building on this result, we define crypto-egalitarianism, a metric which identifies the rate at which wealthy investors accumulate capital and quantifies the identified limitation, with the best possible scenario being a linear reward rate with respect to the invested capital. Second, we consider how a taxation policy can be enforced in such environment, under such limitations. Although the first and, till now, primary application of blockchains has been hosting decentralized financial systems, an evolving line of research looks into integrating this technology in traditional systems, to create centrally-controlled digital cash. We builds on this idea by exploring how distributed ledgers can help solve a widespread problem in real-world economies, *tax gaps*. We contribute by presenting two ideas, via which a tax authority can identify differences between the assets reported by citizens and the actual assets these citizens own.

In addition, Chapter 2 reviews necessary background material and Chapter 9 offers concluding remarks and ties together the thesis's core results.

## 1.2 Publications

A large amount of the work presented in this thesis is based on the following co-authored publications:

- *"A Formal Treatment of Hardware Wallets"* [AGKK19]
  Myrto Arapinis, Andriana Gkaniatsou, Dimitris Karakostas,[1] Aggelos Kiayias
  $23^{rd}$ International Conference on Financial Cryptography and Data Security (FC) 2019

- *"Account Management in Proof of Stake Ledgers"* [KKL20]
  Dimitris Karakostas,[1] Aggelos Kiayias, Mario Larangeira
  $12^{th}$ International Conference on Security in Communication Networks (SCN) 2020

- *"Conclave: A Collective Stake Pool Protocol"* [KKL21]
  Dimitris Karakostas,[1] Aggelos Kiayias, Mario Larangeira
  $26^{th}$ European Symposium on Research in Computer Security (ESORICS) 2021

- *"Efficient State Management in Distributed Ledgers"* [KKK21a]
  Dimitris Karakostas,[1] Nikos Karayiannidis, Aggelos Kiayias
  $25^{th}$ International Conference on Financial Cryptography and Data Security (FC) 2021

- *"Blockchain Nash Dynamics and the Pursuit of Compliance"* [KKZ22]
  Dimitris Karakostas,[1] Aggelos Kiayias, Thomas Zacharias
  Unpublished manuscript

- *"Cryptocurrency Egalitarianism: A Quantitative Approach"* [KKNZ19]
  Dimitris Karakostas,[1] Aggelos Kiayias, Christos Nasikas, Dionysis Zindros
  $1^{st}$ International Conference on Blockchain Economics, Security and Protocols (Tokenomics) 2019

- *"Short Paper: Filling the Tax Gap via Programmable Money"* [KK21]
  Dimitris Karakostas,[1] Aggelos Kiayias
  $5^{th}$ International Workshop on Cryptocurrencies and Blockchain Technology (CBT) 2021

The following publication was also developed during the thesis's preparation, but does not form an integral part of it:

- "Securing Proof-of-Work Ledgers via Checkpointing" [KK20]
  Dimitris Karakostas,[1] Aggelos Kiayias
  $3^{rd}$ IEEE International Conference on Blockchain and Cryptocurrency (ICBC) 2021

[1] Author names are ordered alphabetically.

# Chapter 2

# Background

Security is typically expressed via formal provable statements expressed through so-called *security games*, i.e., time-limited interactions with an adversary. The adversary is modelled as a set of algorithms which, depending on the generality of the security argument, may be abstract or well-defined. In each game, an adversary has some winning condition, e.g., outputting a forged signature. The goal of our security proofs is to demonstrate that, *for any adversary*, the advantage in being successful, compared to outputting a purely random value, is small. If the maximum of the advantage, over all possible adversaries, is $0$, then we enjoy perfect security. Otherwise, the advantage is expressed with respect to the *security parameter* $\kappa$. Specifically, security proofs typically employ asymptotics, by demonstrating that the success probability of the adversary is negligible[1] in the security parameter, denoted by $\mathsf{negl}(\kappa)$.

In summary, the cryptographic treatment of computer systems is based on three cornerstones [KL20]:

1. *Clearly-articulated assumptions*, which distill the limitations of a system. The protocols covered in our work do not operate unconditionally, thus we need to identify the restrictions imposed on the environment, within which all parties operate, and the adversaries, against which our protocols are protected.

2. *Formal statement definitions*, which specify the desirable properties of our protocols, under the aforementioned assumptions.

3. *Rigorous proofs of security*, which guarantee that the formal statements hold, as long as the assumptions are satisfied. In this work, we consider *generic adversaries*,

[1] We say that a function $f$ is negligible in $\kappa$ if, for every $c, d$ in $\mathbb{N}$, there is a $\kappa_0 \in \mathbb{N}$ such that, for all $\kappa > \kappa_0$ and $x \in \{0,1\}^{\kappa^d}$, it holds that $f(x, \kappa) < \kappa^{-c}$.

instead of employing ad-hoc arguments, therefore the proofs are mathematical arguments which typically employ only computational bounds.

In the upcoming sections, we overview the foundations upon which the main body of this thesis is built, assuming a general computer science background on the reader's part. We first overview a list of core cryptographic primitives, which will serve as building blocks in many of our protocols. Following, we provide an overview of classic computing problems, namely consensus and distributed ledgers, and their modern variations, i.e., blockchain-based distributed protocols. This section offers only a brief review of the topics; for a full treatment, we refer to textbooks on: i) cryptography and computer security [KL20, Gol07, Gol09]; ii) distributed systems [CDK02]; iii) blockchain-based ledgers [NBF+16]; iv) game theory [NRTV07, SJ93, Rou16].

## 2.1 Cryptographic Primitives

Across the thesis we assume a peer-to-peer network, where parties use a gossip *diffuse* functionality [GKL15], without the need of a fully-connected graph and point-to-point connections. Consequently, when a party receives a message, they cannot know which party the message originated from.

**Synchronous Network.** Under a synchronous network, a message produced by an honest party at round $r$ is delivered as input to all other parties by at most round $r+t$, where $t$ is a threshold known a priori to a protocol's designer. In the simplest case, when $t=1$, each protocol round simulates a message passing round, i.e., the maximum time required to deliver a message at any computer in the world; in blockchain analyses, the message passing round is typically equal to $10-20$ seconds [GKL15, Woo14].

**Adversary.** The adversary $\mathcal{A}$ is an algorithm that aims at breaking the security of the protocol under review, i.e., violate one of its security properties. $\mathcal{A}$ is *adaptive*, i.e., can corrupt parties dynamically, and *rushing*, i.e., can decide its strategy after receiving, and possibly delaying, the honest parties' messages.

### 2.1.1 Cryptographic Hash Functions

Cryptographic hash functions, as in Damgård [Dam88], exhibit the properties outlined in Definition 1.

**Definition 1** (Hash Function)**.** *A cryptographic hash function* $\mathsf{H} : \{0,1\}^* \to \{0,1\}^l$ *is a function that, for some* $l$ *which is the length of the hash values, presents the following properties:*

- *Collision Resistance: Given* $h \leftarrow \{0,1\}^l$ *it should be computationally infeasible for a probabilistic polynomial algorithm to find a value* $x$ *such that* $h = \mathsf{H}(x)$.
- *Pre-image resistance: It should be computationally infeasible for a probabilistic polynomial algorithm to find two values* $x, y$ *where* $x \neq y$ *such that* $\mathsf{H}(x) = \mathsf{H}(y)$.
- *Second pre-image resistance: Given a value* $x$*, it should be computationally infeasible for a probabilistic polynomial algorithm to find a value* $y \neq x$ *such that* $\mathsf{H}(x) = \mathsf{H}(y)$.

### 2.1.2 Digital Signatures

A digital signature scheme $\Sigma$, as in Canetti [Can03] and Goldwasser *et al.* [GMR84], is a triple of algorithms $\Sigma = \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$, as described in Definition 2. Following Definition 3 describes a core property of signatures, which is resistance to Existential Unforgeability under Adaptive Chosen Message Attacks (EUF-CMA).

**Definition 2** (Digital Signature)**.** *For a security parameter* $\kappa$*, a digital signature scheme* $\Sigma$ *is a tuple* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$*:*

- $\mathsf{KeyGen}(1^\kappa) \to (\mathsf{vk}, \mathsf{sk})$*: a randomized algorithm that, given the security parameter* $\kappa$*, outputs a pair of keys, the verification key* vk *and the* $\kappa$*-bit long private key* sk*;*
- $\mathsf{Sign}(m, \mathsf{sk}) \to \sigma$*: (possibly) randomized algorithm that, given a message* $m$ *and the private key* sk*, outputs a signature* $\sigma$*;*
- $\mathsf{Verify}(m, \mathsf{vk}, \sigma) \to \{0,1\}$*: a deterministic algorithm that, given a message* $m$*, a public key* vk*, and a signature* $\sigma$ *outputs* $1$ *if a signature is valid w.r.t. message* $m$ *and verification key* vk *(respectively* $0$ *if the signature is invalid).*

**Definition 3** (EUFCMA)**.** *A digital signature scheme* $\Sigma$ *is Existentially Unforgeable under Adaptive Chosen Message Attacks (EUF-CMA) if it presents the following properties:*

- *Completeness: For any message* $m$*, it holds:*

  $$\mathsf{Pr}[(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa), \sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}) : 0 \leftarrow \mathsf{Verify}(m, \mathsf{vk}, \sigma)] \leq \mathsf{negl}$$

  *where all probabilities are computed over the random coins of the generation and sign algorithms.*

- *Consistency: For any message $m$, the probability that two independent executions of* $\mathsf{Verify}(m, \mathsf{vk}, \sigma)$ *for a key pair* $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$*, output two different outcomes is smaller than* negl.

- *Unforgeability: For any PPT algorithm $\mathcal{A}_{forger}$, which can query the signature oracle* $\mathsf{Sign}(\cdot, \mathsf{sk})$ *for signatures on a polynomial number of messages $m_i$, it holds:*

$$\Pr\left[(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa) : \begin{array}{c}(m, \sigma) \leftarrow \mathcal{A}_{forge}^{\mathsf{Sign}(\cdot,\mathsf{sk})} \wedge \\ m \neq m_i \wedge \\ \mathsf{Verify}(m, \mathsf{vk}, \sigma) = 1\end{array}\right] \leq \mathsf{negl}(\kappa)$$

  *where all the probabilities are computed over the random coins of the generation algorithm and the adversary.*

### 2.1.3 The Universal Composability Framework

The Universal Composability (UC) Framework by Canetti [Can01] is a tool that enables us to capture the security properties of a distributed protocol. As a preparation for presenting the framework, consider two ensembles $X = \{X_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ and $Y = \{Y_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ of binary random variables. $X$ and $Y$ are said to be *computationally indistinguishable*, denoted by $X \approx_c Y$, if for all $z$ it holds that $|\Pr[\mathcal{D}(X_{\kappa,z}) = 1] - \Pr[\mathcal{D}(Y_{\kappa,z}) = 1]|$ is negligible in $\kappa$, i.e., negl, for every probabilistically polynomial-time (PPT) distinguishing algorithm $\mathcal{D}$.

The main idea of security proofs under the UC framework relies on the comparison between the execution of a concrete protocol, say $\pi$, and a security definition, named the *ideal functionality*. These two executions are, respectively, the *real world* and the *ideal world*. Both are controlled by an entity called the *environment*, denoted by $\mathcal{Z}$, which can submit actions and observe outputs from the executions. The environment controls the execution of $\pi$, through choosing the inputs of its participants, and also the actions of the adversary $\mathcal{A}$ in the real world. We note that our work is restricted on executions where every party $\mathcal{P} \in \mathbb{P}$ is activated on each time slot. $\mathcal{Z}$ also controls the activation schedule and the inputs of each party. The adversary $\mathcal{A}$ can read the messages exchanged between the protocol players and even delay them, to a degree that depends on the network model. Moreover, it is allowed to corrupt players per the environment's instructions, in which case the player's secret state is compromised and is available to the adversary.

More formally, every entity is modeled as a PPT Interactive Turing Machine (ITM), and the real world and ideal executions are respectively represented by the ensembles:

$$\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} = \{\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z, r)\}_{\kappa\in\mathbb{N}, z\in\{0,1\}^*}$$

and

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} = \{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z, r)\}_{\kappa\in\mathbb{N}, z\in\{0,1\}^*}$$

and uniform randomly chosen value $r$. We use $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z, r)$ to denote the output of the environment $\mathcal{Z}$ in the real-world execution of a protocol $\pi$ and the adversary $\mathcal{A}$ under security parameter $\kappa$, input $z$ and randomness $r$. Analogously, we denote by $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z, r)$ the output of the environment in the ideal interaction between the simulator $\mathcal{S}$ and the ideal functionality $\mathcal{F}$ under security parameter $\kappa$, input $z$ and randomness $r$. It is said that *the protocol $\pi$ securely realizes the functionality $\mathcal{F}$* when the environment cannot distinguish between the two worlds, i.e., for every $\mathcal{A}$ exists a simulator $\mathcal{S}$ such that for every PPT $\mathcal{Z}$ we have that $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \approx_c \mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

## 2.2 Distributed Ledgers

We now overview the problem of constructing a secure distributed ledger. Specifically, starting from the problem of consensus, a fundamental problem of distributed computing on which distributed ledgers are based, we provide formal definitions for the problem space that blockchain protocols aim to solve.

### 2.2.1 Consensus

The consensus problem, introduced in the seminal work of Shostak, Pease, and Lamport [LSP82, PSL80a], is the setting where a set of parties, each with its own input, need to reach agreement and output the same value. A protocol that solves the consensus problem demonstrates three core properties [CDK02], namely termination, agreement, and validity (Definition 4).

**Definition 4** (Consensus)**.** *A consensus protocol $\Pi$, which is performed among $n$ parties $\mathcal{P}_i$, each with input $v_i$, satisfies the following properties:*

- *Termination: eventually each correct process outputs a single value;*
- *Agreement: all correct processes output the same value;*

- *Validity: if all correct processes start $\Pi$ with the same input $v$, then every correct process outputs $v$.*

### 2.2.2 Reliable Broadcast

A problem close, but not identical, to consensus is that of Reliable Broadcast (RBC), which will prove useful in the construction of the collective pool of Chapter 5. Briefly, a RBC protocol ensures that a message output by an honest party is eventually broadcast and output by all other honest parties. Definition 5 describes RBC, following the formalization of [CJKR12, MR21].

**Definition 5** (Reliable Broadcast)**.** *Let $\mathcal{P}_s$ be a designated sender of its input $b_{in}$. A Reliable Broadcast (RBC) protocol, which is performed among $n$ parties, satisfies the following properties;*

- *Safety: i) Consistency: if two honest parties output values $b, b'$ respectively, then $b = b'$; ii) Integrity: if $\mathcal{P}_s$ is honest, no honest party outputs a value $b \neq b_{in}$.*
- *Liveness: i) Validity: if $\mathcal{P}_s$ is honest, then all honest parties output some value; ii) Totality: if an honest party commits a value, then all honest parties output some value.*

### 2.2.3 Distributed Ledger

A ledger is an append-only list of ordered transactions: $\mathcal{L} = [\tau_0, \ldots, \tau_j]$. A distributed ledger is a ledger that is maintained in a decentralized manner, i.e., by multiple parties without a single central authority. Intuitively, a ledger can be seen as a repetitive execution of a consensus protocol, where the input is a transaction. As Definition 6 shows, a distributed ledger is secure if all parties agree on the transaction ordering and the ledger expands over time.

**Definition 6** (Distributed Ledger)**.** *Let $\Pi$ be a protocol that is run by $n$ parties; $\Pi$ implements a secure distributed ledger if the following properties are satisfied:*

- *Safety: If two honest parties output $[\tau_0, \ldots, \tau_j]$ and $[\tau'_0, \ldots, \tau'_j]$ respectively, then $\forall i \in [0, \min(j, j')] : \tau_i = \tau'_i$.*
- *Liveness: If a transaction $\tau$ is provided as input to at least one honest party, eventually all honest parties output a ledger containing $\tau$.*

## 2.3 Bitcoin and Blockchains

Blockchains are a special family of protocols that solve the distributed ledger problem. Their name is derived from the mechanics of the underlying data structure. Specifically, in blockchain systems, transactions are grouped in blocks. Each block $\mathcal{B}$ contains an Merkle Tree [Mer88] of (ordered) transactions, as well as a hash pointer to another block. Each block points to exactly one other block, thus forming a tree of blocks. The root of the tree is typically a global parameter, called the "genesis" block. Each branch of the tree consists of a well-ordered chain of blocks $\mathcal{C}$, thus implementing a distributed ledger (cf. Section 2.2.3). Additionally, a block contains a timestamp and a nonce, which is used during the Proof-of-Work computation (see below). Figure 2.1 depicts a simple model of Bitcoin's blocks.

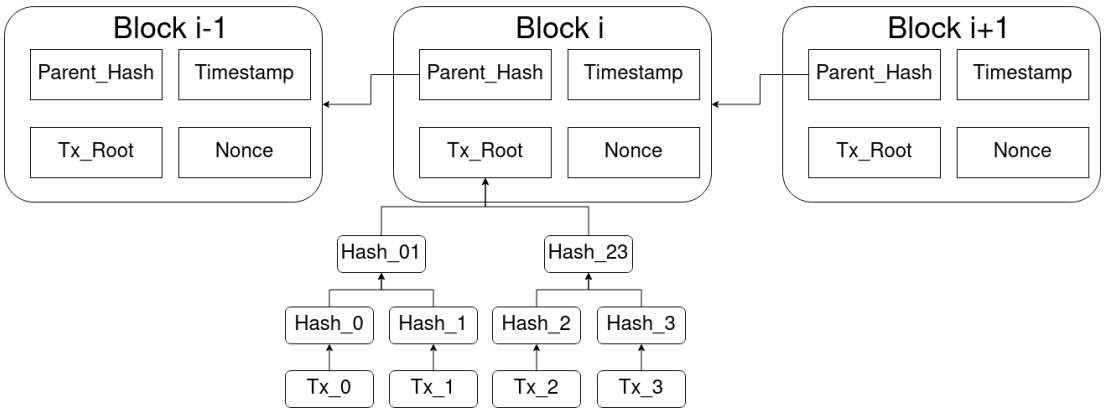


Figure 2.1: Bitcoin's blockchain data structure.

For the rest of this thesis, we will use the following notation:

- $\mathcal{B}$ denotes a block;
- $\mathcal{C}$ denotes a chain of blocks;
- $\mathcal{C}||\mathcal{B}$ denotes the concatenation of a chain and a block;
- head$(\mathcal{C})$ denotes the last block of a chain $\mathcal{C}$, i.e., the block farthest from genesis s.t. head$(\mathcal{C}||\mathcal{B}) = \mathcal{B}$;
- $(\prec)$ denotes the prefix operation between two chains; so, if $\mathcal{C}' \prec \mathcal{C}$, then there exists a chain of blocks $\mathcal{B}_0||\ldots||\mathcal{B}_j$ such that $\mathcal{C} = \mathcal{C}'||\mathcal{B}_0||\ldots||\mathcal{B}_j$;
- $(\backslash)$ denotes the difference of two chains, e.g., if $\mathcal{C} = \mathcal{C}'||\mathcal{C}''$ then $\mathcal{C}'' = \mathcal{C}\backslash\mathcal{C}'$;

- $\mathcal{C}^{\lceil k}$ denotes the chain obtained by removing the last $k$ blocks of chain $\mathcal{C}$, i.e., $\mathcal{C} \backslash \mathcal{C}^{\lceil k} = \mathcal{C}'$ where $|\mathcal{C}'| = k$;
- $|\mathcal{C}|$ denotes the length of chain $\mathcal{C}$ in blocks;
- $\mathcal{C}[k]$ denotes either the $k$-th block or the $k$-th transaction in $\mathcal{C}$ (depending on the context).

Bitcoin Backbone [GKL15, GKL17] and subsequently [KP15] distilled the properties that a secure blockchain protocol must satisfy (Definition 7). Based on these properties and Definition 6, Definition 8 describes the combined, high-level properties of *persistence* and *liveness* for blockchain-based distributed ledgers.

**Definition 7.** *Assume $n$ parties, each party $\mathcal{P}_i$ locally holding a chain $\mathcal{C}_i = [\tau_{i,0}, ..., \tau_{i,j}]$. A secure blockchain protocol must satisfy the following properties:*

- *Common Prefix: For parameters $k, r_0 \in \mathbb{N}$, the chains $\mathcal{C}_1, \mathcal{C}_2$ held locally by the honest, but not necessarily distinct, parties $\mathcal{P}_1, \mathcal{P}_2$ at rounds $r_0 < r_1 \leq r_2$ respectively, satisfy: $\mathcal{C}_1^{\lceil k} \prec \mathcal{C}_2$.*
- *Chain Quality: For parameters $l, \mu \in \mathbb{N}$, any consecutive sequence of $l$ blocks, in a chain held locally by any honest party at any point in time, comprises of at least $\mu$ blocks created by an honest party.*
- *Chain Growth: For parameters $s, t \in \mathbb{N}$, if at round $r$ an honest party $\mathcal{P}_i$'s chain is $\mathcal{C}_{i,r}$, then, at round $r + s$, the chain $\mathcal{C}_{i,r+s}$ locally held by $\mathcal{P}_i$ satisfies $|\mathcal{C}_{i,r+s}| \geq \mathcal{C}_{i,r} + t$.*

**Definition 8.** *Assume $n$ parties, each party $\mathcal{P}_i$ locally holding a chain $\mathcal{C}_i = [\tau_{i,0}, ..., \tau_{i,j}]$.*

*We call a transaction $\tau$ stable if, assuming that for some honest party $\mathcal{P}_i$ and some $k \in \mathbb{N}$ it holds $\mathcal{C}_i[k] = \tau$, then for every honest party $\mathcal{P}_l$ it holds $\mathcal{C}_l[k] = \tau$.*

*A blockchain-based distributed ledger protocol is secure, with parameters $k, u$, if it satisfies the following properties:*

- *Persistence: For each party $\mathcal{P}_i$, a transaction which is part of a block in a prefix chain $\mathcal{C}'_i \prec \mathcal{C}$, such that $|\mathcal{C}_i| - |\mathcal{C}'_i| \geq k$, is stable.*
- *Liveness: A transaction which is provided as input to an honest party at round $r$ has become stable by round $r + u$.*

The major innovation of Bitcoin [Nak08a] was to use a blockchain to implement a secure distributed ledger under open and dynamic participation [GKL15, GKL17, PSs17].

Up to that point, distributed ledger protocols assumed a known number of participating parties; instead, Bitcoin allows parties to join or leave the protocol as needed. However, in such open-participation setting, Bitcoin faced two questions: i) how to identify the party responsible for producing a new block at any stage of the protocol; ii) how to prevent sybil attacks [Dou02]. Briefly, in a sybil attack an adversary creates a large number of pseudonymous identities to gain disproportionate power in a system and subvert its security. Bitcoin answered both these questions with its Proof-of-Work, with other blockchain protocols following suit with alternatives like Proof-of-Stake.

**Proof-of-Work (PoW).** The core idea behind PoW [DN93] is performing some amount of computations and then, in order to participate in a protocol, publish a proof of this "work" performed by the hardware. In cryptocurrencies like Bitcoin, the PoW work is called "mining". The mining hardware is provided with two constants, previd and data, i.e., the id of the tip of the adopted blockchain and the data which need to be appended to it. The mining device then brute-force searches for some string nonce, such that $\mathsf{H}(\mathsf{previd}||\mathsf{data}||\mathsf{nonce}) \leq T$ for some hash function H defined by the system. $T$ is a — relatively — small number, called the *difficulty target*, which is adjusted to ensure a stable block production rate, although typically remains constant for periods of consecutive blocks called epochs. For example, in Bitcoin, epochs are $2016$ blocks long [BMC+15], while each new block is produced approximately every $10$ minutes. Because the search for solutions is exhaustive, the expected number of solutions found by a given miner is proportional to the number of evaluations of the hash function H they can obtain in a given time frame.

**Proof-of-Stake (PoS).** PoW's deficiencies, particularly its egregious environmental cost, have driven research towards alternative designs, most prominently Proof-of-Stake (PoS). In PoS, a minter is selected in proportion to the stake they hold, which is to say proportionally to the amount of assets they own. These assets are managed by the distributed ledger and serve as both the system's internal currency and consensus participation tokens. There exist a number of flavors of this process. In one case, e.g., Ouroboros [KRDO17], all coins automatically participate in the leader election process. In a second flavour, the stake has to opt-in to participate in the election by a special process, such as purchasing a ticket or becoming a delegate of the stake of other users; this is the case for cryptocurrencies like Decred [dec19] and EOS [Com18]. PoS systems are almost energy-free, but often rely on complex cryptographic primitives, e.g., secure Mul-

tiparty Computation [KRDO17], Byzantine Agreement [DPS19, GHM+17a, KJG+16], or Verifiable Random Functions (VRFs) [DGKR18, GHM+17a].

Starting with Bitcoin, blockchain-based distributed ledgers are typically used for financial applications. The ledger acts as a database which records the ownership of digital assets at any point and the transactions, i.e., transfers of asset ownership, that are performed between users. Each user interacts with the assets recorded on the ledger via *addresses*.

An address $\alpha$ is a string chosen from the set $\mathbb{A} \subseteq \{0,1\}^{\ell}$, where $\ell$ is the — protocol-specific — address length parameter. At any point in time, the ledger records the assets that each address owns. A user controls a single *account*, which may in turn control multiple addresses; therefore, a user participates in the ledger, via the account's addresses, in a pseudonymous manner.[2] Intuitively, an address is akin to an IBAN, in traditional bank accounts. Typically, each address is associated with a payment key pair $(\mathsf{vk}, \mathsf{sk})$, where vk is the public and sk the private key. As explored in Chapter 4, an address may be associated with additional data, such as a staking key or metadata. Additionally, in contract systems like Ethereum, a smart contract may not be associated with a key pair, but rather only with a piece of software. However, in all cases, addresses that are maintained by users of the system are associated with at least one key pair, which is used for receiving and issuing payments.

To transfer assets between addresses, a user creates a transaction $\tau$. In the simplest case, a transaction comprises of the following objects: i) $\alpha_s$: the address of the sender, i.e., the current owner of the assets; ii) $\alpha_r$: the address of the receiver of the assets; iii) $\theta$: the amount or set of assets which are transferred. Typically, the transaction is also signed by the private key associated with $\alpha_s$, such that the sender can prove ownership of the assets. Optionally, the transaction may also contain additional information, such as a number of fees, a change address (i.e., to receive surplus assets), or metadata (as explored in Chapters 3 and 4). $\theta$ may be a simple number, if the exchanged assets are fungible, or a set of individual asset identifiers, in the case of non-fungible assets.

**Asset Fungibility.** An asset is fungible if each unit is indistinguishable from the others; in contrast, each unit of a non-fungible asset is identified by a unique identifier, which can be used to track it and trace its history. For example, the US Dollar (USD) is a fungible asset, as citizens transact in amounts of USD and each single unit of USD is

[2] The literature has also seen a number of fully anonymous blockchain protocols [BCG+14, MGGR13, CGL+17, KKKZ19], but these are outside the scope of this thesis.

indistinguishable. However, USD bills or banknotes are non-fungible, since each bill is identified by a unique serial number, which is often used to track forged or stolen assets. In our work, a non-fungible asset is identified by a natural number $\theta$, while for fungible assets we consider only amounts denoted by $\Theta$.

## 2.4 Literature Overview

We now overview notable works that pertain to the entire scope of the thesis. Following, each individual chapter focuses on the body of literature related to its specific research questions.

### 2.4.1 Bitcoin Formal Models

The importance of formal methods for analyzing Bitcoin is well understood, with existing literature showcasing different approaches. Garay *et al.* [GKL15, GKL17], after extracting and analyzing the core Bitcoin blockchain protocol, presented a formal abstraction to prove that Bitcoin satisfies a set of security and quality properties. Pass *et al.* [PSs17] analyzed the consistency and liveness properties of the consensus protocol in an asynchronous setting, proving Bitcoin secure assuming an upper bound on the network delay. Badertscher *et al.* [BMTZ17] suggested a Universally Composable treatment of the Bitcoin ledger, defining Bitcoin's goals and proving that their model is securely realized in the UC framework. Transactions, being a core part of Bitcoin, have also attracted attention; notably, Atzei *et al.* [ABLZ18] proposed a formal model of Bitcoin transactions, to prove security e.g., against double-spending attacks.

### 2.4.2 Proof-of-Stake Protocols

The cryptographic literature has seen a number of PoS protocols in the past years. A family of such protocols, which was initiated with Ouroboros [KRDO17] and continued with Ouroboros Praos [DGKR18], Ouroboros Genesis [BGK+18], and Ouroboros Crypsinous [KKKZ19], provides a wide range of threat model considerations, relevant to PoS systems, and offers eventual guarantees of liveness and persistence, similar to Bitcoin. Algorand [CGMV18, GHM+17b] is a protocol which employs Byzantine Agreement to achieve the necessary properties of a PoS setting, as well as transaction finality in (expected) constant time. Snow White [DPS19, PS17b] similarly uses the notion of

"robustly reconfigurable consensus", which is specially designed to cope with the lack of participation of users in the consensus protocol.

Real-world PoS implementations often opt for stake representation and delegation, similar to our work in Chapter 4. Systems like Cardano [Car], EOS [Com18], and (to some extent) Tezos [Goo14], employ different consensus protocols, but all enforce that a (relatively small) subset of representatives is elected to participate. Decred [dec19] takes a somewhat different approach, where stakeholders buy a ticket for participation, akin to PoS with optional participation. However, these systems typically assume single parties that act as delegates, either individually or as pool operators, a restriction that Chapter 5 directly aims at relaxing.

### 2.4.3 Blockchain Incentives

The seminal work of Selfish Mining [ES14, SSZ16, KKKT16a] showed that honest behavior is not incentive-compatible. However, alternative reward sharing mechanisms in the PoS setting may make it feasible to perform better in terms of incentive compatibility. For instance, Ouroboros [KRDO17] can be designed from the ground up to be a Nash equilibrium under certain plausible conditions. The question of how to incentivize parties in PoS systems to form a desired number of stake pools was further studied in [BKKS20]. The problem that these works study, and which we also explore in Chapter 7, is designing the incentives of blockchain systems from the designer's point of view, so that participants do not deviate from the prescribed protocol. One related question is how fair the protocol is to participants themselves, particularly to honest participants. The Bitcoin Backbone and Selfish Mining works include attacks in which an adversary can strategically the ledger's performance, causing the number of blocks and, in turn, the respective rewards, to be disproportionate to their contributed computational power, thereby harming fairness towards honest participants. In the PoW setting, Fruitchains [PS17a] proposes a protocol which solves this problem via the notion of "fair" rewards, i.e., rewards in exact proportion to the computational power of each party.

# Chapter 3

# Formalization of Hardware Wallets

Bitcoin, being the most successful cryptocurrency, has been repeatedly attacked with many users losing their funds. As wallets are the only way for a user to access their funds, they are repeatedly targeted for attacks that aim to access the account's keys or redirect the payments, ranging from clipboard hijacking [Par16] and malware [HDM+14] to implementation bugs, e.g., the Parity hack in Ethereum [Alo17], and more specific attacks, e.g., brain wallets [VBC+16]. In order to address such threats, different ways to harden the wallet's security have been proposed, with the most notable one being the utilization of cryptographic hardware, i.e., *hardware wallets*. These specialized devices currently dominate the market and, as demand grows, so does the number of commercially available products. However, they are also arguably the least formally studied part of cryptocurrency ecosystems, with their specifications and security goals remaining unclear and understudied.

The module known as a hardware wallet is responsible for the account's key management and the execution of the required cryptographic operations. The remaining operations are completed by a dedicated software, either provided together with the hardware or by a third party, with which the hardware communicates. However, incorporating expensive hardware as a wallet is bound to bring some security guarantees. Currently, the security of commercially available products can only be checked through manual inspection of their implementation, a process that requires a strong engineering and technical background, and a significant effort and time commitment. In turn, proprietary assumptions and lack of a universal threat model frequently lead to implementations prone to attacks.

Our work aims at bridging the gap between formally modeling and verifying the wallet's properties and claimed specifications. In this chapter, we devise a formal model

which defines the characteristics, specifications, and security requirements of hardware wallets. Instead of capturing a hardware wallet as a single module, we conceptualize it as a system of different modules that communicate with each other to perform the wallet's operations. This approach allows us to identify a greater set of potential attacks and the conditions for them to be successful. To that end, we manually inspect commercial products and extract their implementations, which we then map to our model. As we show, hardware wallets are prone to a set of attacks and are secure only under specific, well-defined assumptions. Therefore, our model can not only prove the security of existing implementations, but also act as a reference guide for future implementations.

**Related work.** Until now, Bitcoin hardware wallets have only been empirically studied, with research focusing on the integrity of transactions. Gentilal *et al.* [GMS17] stressed the necessity of separating the wallet into two environments, the trusted and the non-trusted, and proposed that a wallet remains secure against attacks by isolating the sensitive operations in the trusted environment. Similarly, Lim *et al.* [LKL+14] and Bamert *et al.* [BDWW14], argue that security in Bitcoin wallets equals with tamper-resistance and propose the use of cryptographic hardware. Hardware wallets have not yet been extensively studied, since no formal attempt to specify the functionalities and the security properties of such wallets exists so far. As of September 2018, research has only focused on attacking commercially available implementations. Gkaniatsou *et al.* [GAK17] showed that the low-level communication between the hardware and its client is vulnerable to attacks which escalate to the account management. Their research concluded to a set of attacks on the Ledger wallets, which allowed to take control of the account's funds. Hardware wallets have also been studied against physical attacks. Volotikin [Vol18] showed that specific parts of the Ledger's flash memory are accessible, exposing the private keys used for the second factor verification mechanism. Datko *et al.*presented fault injection, timing and power analysis attacks on KeepKey [Kee18] and Trezor [Tre18a], which allowed them to extract the private key.

**Contributions.** This chapter provides a holistic formal treatment of hardware wallets, identifying their core specifications and security properties. The proposed model allows to reason about the offered security of existing wallets and acts as the foundation for designing and implementing new ones. In particular, this chapter: i) defines the properties and requirements of hardware wallets; ii) provides a formal model and security guarantees of such wallets; iii) evaluates the security of commercial products under a

formal model.

In more detail, we identify the properties and security parameters of a Bitcoin hardware wallet and formally define them in the Universal Composition (UC) Framework. We present a modular treatment of a hardware wallet, by realizing the wallet as a set of protocols. This approach allows us to capture in detail the wallet's components and interactions, and the potential threats. We deduce the wallet's security by proving that it is secure under common cryptographic assumptions, provided that there is no deviation in the protocol execution. Finally, we define the attacks that are successful under a protocol deviation, and analyze the security of commercially available wallets.

## 3.1 Formal Model of Hardware Wallets

Bitcoin relies on the Elliptic Curve Signature Scheme (ECDSA) for signing the transactions and proving ownership of the assets. A Bitcoin account is defined by a key pair $(\mathsf{vk}, \mathsf{sk})$. The public key vk is hashed to create an address $\alpha$ for receiving assets, while the private key sk is used to sign transactions that spend the assets that $\alpha$ received. Unauthorized access to sk can result in a loss of funds, thus raising the issue of securing the account's private keys. A Bitcoin wallet, which offers access to the network and management of an account, is either implemented via software, i.e., is hosted online by a third party or run locally, or hardware.

The first threat arises from broken cryptographic primitives. For instance, a broken hash function may allow an attacker to correlate an address with multiple keys, whereas a broken signature scheme may enable an attacker to forge transaction signatures, for addresses which it does not own. Additionally, protecting against unauthorized access to the wallet's operations has been shown to be as important as using secure cryptographic primitives [BMC+15, GAK17]. To that end, hardware wallets offer a tamper-resistant and isolated environment for the cryptographic primitives.

However, a hardware wallet cannot exist in a standalone setting, but rather requires a client via which it connects to the network. Thus, if the hardware is connected to a compromised client, any inputs/outputs of the hardware can potentially be malicious. For example, consider Bob, whose account is defined by the key pair $(\mathsf{vk}, \mathsf{sk})$, and an adversary $\mathcal{A}$, who is able to forge Bob's signature. In this case any signature $\sigma_{\mathcal{A}}$ of a message $m_{\mathcal{A}}$ chosen by $\mathcal{A}$ can be verified by vk; hence the adversary can spend all assets that Bob has previously received, i.e., the assets sent to the hash of vk. Let us now assume that Bob's signature is unforgeable but $\mathcal{A}$ controls the signing algorithm

inputs, such that for any message $m$ that Bob wishes to sign, $\mathcal{A}$ substitutes it with $m_{\mathcal{A}}$. Even though the signature is unforgeable, the adversary can still spend Bob's assets by tampering with the message.

Our model captures the family of attacks which tamper with the inputs/outputs of the wallet's operations. Thus, as we show, the security of a Bitcoin hardware wallet is reduced to the security of the underlying cryptographic primitives *and* the honesty of the three communicating parties, i.e., the hardware, the client, and the user who operates them.

We note that our model does not capture attacks on the network level. In particular, we assume synchronous communication channels between the participants of the hardware wallet system, i.e., the user, client, and hardware; this is a natural assumption in this setting, as we expect a direct connection both between the client and the hardware (e.g., via USB) and with the user (via I/O devices). However, we also assume that, once a transaction is signed and transmitted by the client, it is published on the ledger within a short time span. Therefore, our model does not capture possible liveness violations, e.g., delays due to congestion, but rather abstracts the ledger as an ideal module that satisfies the necessary properties. Nonetheless, a hardware wallet model that connects to a possibly temporarily insecure ledger would be a useful enhancement.

**The Hardware Wallet Setting.** Software, not being tamper-resistant, cannot guarantee a secure environment for the wallet's operations. Instead, hardware wallets offer such an environment by separating the wallet's cryptographic primitives from the other operations, e.g., connection to the Bitcoin network. These devices do not offer network connectivity; instead they operate in an offline mode. Due to their limited memory capabilities and the absence of network access, they cannot keep track of the account's activities, e.g., past transactions. Thus, they connect to a dedicated software, *the client*, which records the account's actions and provides a usable interface to the user. Hardware wallets operate under the assumption of a malicious host and provide a trusted path with the user. Specifically, the hardware displays transaction-related data, which the user compares to the data presented by the (potentially compromised) client. As such, the user becomes part of the system and is responsible for identifying malicious actions by the client.

The wallet operations are initiated by the user, they are executed by the client, the hardware, or both, and are as follows:

- *Setup*: the hardware generates a master key pair $(\mathsf{mvk}, \mathsf{msk})$ and returns the pri-

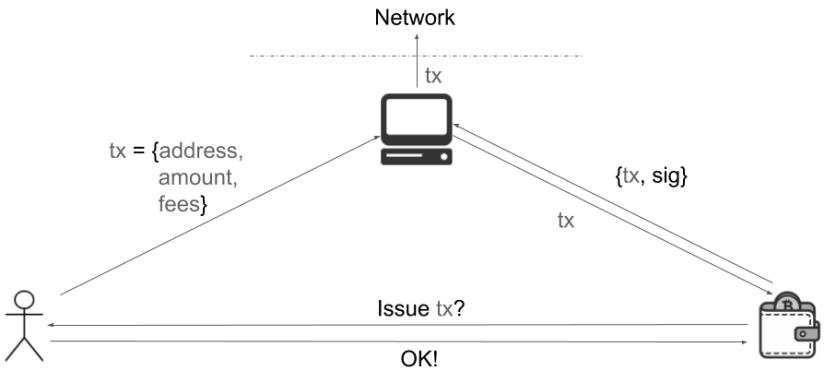


Figure 3.1: Transaction issuing in the hardware wallet setting.

vate key msk, i.e., the wallet's *seed*, to the user; currently all wallets are Hierarchical Deterministic [DFL19], mostly based on the BIP32 standard [Wui18].

- *Session Initialization*: the hardware connects to the client and sends to it the master public key mvk;
- *Generate Address*: both the client and the hardware generate a new address and return it to the user. The hardware derives a key pair $(\mathsf{sk}_i, \mathsf{vk}_i)$ from $(\mathsf{mvk}, \mathsf{msk})$, generates the address $\alpha_i$ and returns it to the user. The client may either generate $\mathsf{vk}_i$ using mvk or receive $\mathsf{vk}_i$ from the hardware, then generates and stores the corresponding address, and finally returns it to the user.
- *Calculate Balance*: given a list of the account's addresses, the client iterates over the ledger's transactions, calculates the account's available assets and returns this amount to the user.
- *Transaction Issuing*: the user provides the payment data to the client, i.e., the amount and receiver's address, which then forwards them to the hardware together with the available inputs, i.e., the account's addresses and balance, and requests its signature. The hardware checks whether the inputs belong to the managed account and generates a *change address* upon demand, i.e., if the balance is larger than the payable amount. Then, it requests the user's approval of the payment data; if the user confirms the payment, the hardware signs it and returns the signature together with the corresponding public key to the client in order to publish it on the network.

Figure 3.1 illustrates the issuing of a transaction in the hardware-enhanced wallet setting.

Evidently, a hardware-assisted wallet is not a single module, but rather an "ecosystem" of modules that communicate during an operation: the user, the client, and the hardware. To treat it under the UC framework, we will next describe an ideal functionality, that defines the wallet's properties, as well as its real world implementation. In the ideal world, the wallet is a monolithic component-functionality, responsible for all aforementioned operations. In the real world, the wallet emerges through the interaction of the human operator, the client (i.e., a desktop computer, tablet, or smartphone), and a tamper resistant hardware component. To prove the security of the real-world protocol, we will compare the two executions (in the real and ideal settings) and show that they are indistinguishable.

**Ideal World.** The wallet functionality, $\mathcal{F}_{\text{w}}$, defines the wallet's operations in the ideal setting. $\mathcal{F}_{\text{w}}$ interacts with the global Bitcoin ledger functionality $\mathcal{G}_{\text{LEDGER}}$, as defined in [BMTZ17], in order to execute operations requiring access to the decentralized system. $\mathcal{G}_{\text{LEDGER}}$ is the ideal functionality that models the Bitcoin ledger and allows a wallet to register itself, publish transactions and retrieve the state of the ledger, i.e., all published transactions. $\mathcal{F}_{\text{w}}$ generates a unique address per public key and also incorporates a signature functionality, $\mathcal{F}_{\text{SIG}}$, as defined in [Can03]; for ease of notation, we show $\mathcal{F}_{\text{SIG}}$ as a separate component in the ideal world, although in fact $\mathcal{F}_{\text{w}}$ simply repeats the steps defined in $\mathcal{F}_{\text{SIG}}$. Specifically, the wallet registers itself with $\mathcal{F}_{\text{SIG}}$, which creates fresh keys for the account upon request, e.g., during address generation, and signs messages, e.g., transactions. $\mathcal{F}_{\text{SIG}}$ is also accessed by the validation predicate of $\mathcal{G}_{\text{LEDGER}}$, in order to verify a transaction's signature during the validation stage.

**Real World.** The operations defined by $\mathcal{F}_{\text{w}}$ are executed in the real world by a set of communicating parties: the *hardware*, the *client*, and the *user*. Thus the protocols of the hardware $\pi_{hw}$, the client $\pi_{client}$, and the human $\pi_{human}$ define the actions of the corresponding parties. The hardware protocol $\pi_{hw}$, uses a signature scheme $\Sigma \equiv \langle \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{Sign} \rangle$, a cryptographic hash function H and a pseudorandom key generation function $\mathsf{HierarchicalKeyGen}(\mathsf{msk}, i)$, in order to derive children keys from the master key. A basic assumption of this setting is that $\pi_{client}$ runs in an untrusted environment, i.e., we do not assume the software to be secure. Thus, in our model, connection to a malicious client is equivalent to corruption of the client by the adversary. Finally, the user communicates with the hardware and the client via a secure channel, i.e., can interact directly with the device via buttons and/or a display.

**Validation Predicate.** In both settings, $\mathcal{F}_{\text{w}}$ and the real-world parties have access to a global Bitcoin ledger functionality $\mathcal{G}_{\text{LEDGER}}$, as defined in [BMTZ17]. $\mathcal{G}_{\text{LEDGER}}$ is parameterized with the validation predicate Validate, which identifies whether a transaction can be added to its buffer, i.e., if it is valid for publishing on the ledger. The predicate takes as input the candidate transaction, the buffer, and the current state. The transaction consists of the signed data $\tau_\sigma = (\tau, \mathsf{vk}, \sigma)$ and ledger parameters, e.g., the transaction id, which is a unique identifier of the transaction, and the timestamp, i.e., the time defined by a global clock. Intuitively, the Bitcoin state is the blockchain, while the buffer is the mempool, which contains the transactions that have not yet been included in the blockchain. In our model, the validation predicate performs the signature verification of a transaction. This is formalized with a wrapper ValidateWrapper, which wraps all instantiations of the validation predicate. In the ideal world, the wrapper accesses the signature functionality $\mathcal{F}_{\text{SIG}}$ to verify the transaction's signature; if the signature is not valid, then it directly outputs $0$, otherwise it performs all additional checks, such as verifying the consumed funds and checking the amounts' validity. The ideal wrapper IdealValidateWrapper is described in Algorithm 1. The real-world wrapper RealValidateWrapper uses a signature scheme instead of $\mathcal{F}_{\text{SIG}}$ and behaves similarly to Algorithm 1, i.e., it first parses BTX and then performs the same branch checks on $\text{Verify}(\tau, \mathsf{vk}, \sigma)$ and returns the proper boolean value.

**Algorithm 1** The validation predicate wrapper, parameterized by Validate and $\mathcal{F}_{\text{SIG}}$. The input is a transaction BTX, the buffer buffer and the state state.

**function** IdealValidateWrapper(BTX, buffer, state)
  $(\tau, \mathsf{vk}, \sigma, \mathsf{txid}, \tau_L, p_i) = \text{parse}(\text{BTX})$
  Send $(\text{Verify}, \mathsf{sid}, \tau, \sigma, \mathsf{vk})$ to $\mathcal{F}_{\text{SIG}}$ and receive $(\text{Verified}, \mathsf{sid}, \tau, f)$
  **return** $f$
**end function**

Finally, Figure 3.2 illustrates the ideal and the real world settings. In both worlds, the environment $\mathcal{Z}$ interacts with the adversary; Specifically, in the ideal world it interacts with the simulator $\mathcal{S}$ and in the real world with the adversary $\mathcal{A}$. In the ideal world, the wallet consists of the ideal wallet functionality $\mathcal{F}_{\text{w}}$ and the signature functionality $\mathcal{F}_{\text{SIG}}$. In the real world, it consists of the user, client, and hardware parties, who execute the respective protocols $\pi_{human}, \pi_{client}, \pi_{hw}$. Communication between the human, client, and hardware is achieved over a *UC-secure channel protocol*, as presented by Canetti [CK02]. Specifically, the adversary is able to observe the encrypted commu-

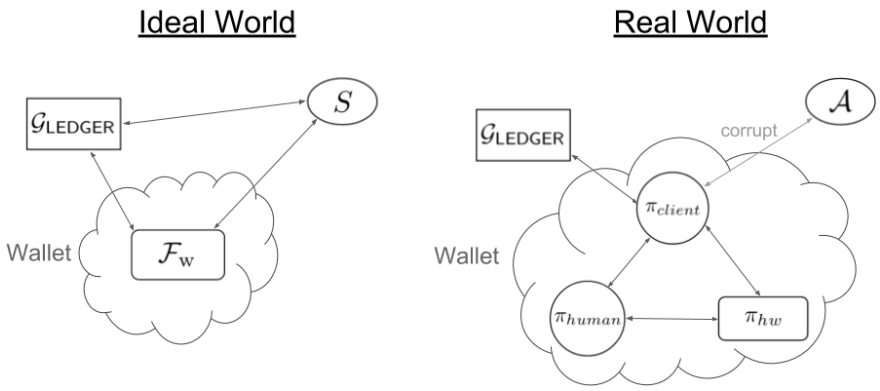


Figure 3.2: A high-level comparison of the ideal and the real worlds.

nication between the honest parties, but can only retrieve the length of the exchanged messages and not tamper with the plaintext. In practice, this can be achieved by establishing a secure channel between the client and the hardware module using standard key exchange techniques, while the human-hardware channel is assumed to be secure by default. We note that, in the absence of a secure channel, the adversary may tamper with the communication, so, in our model, an insecure channel is equivalent to the client being corrupted.

In the chapter's upcoming sections we use a simplified transaction model, for ease of notation. Specifically, each transaction defines a single input and output. We note that adapting it for multiple inputs and outputs, in order to properly model real-world ledgers, is rather straightforward. The wallet's key pairs, and consequently its addresses, are generated using the master key pair $(\mathsf{mvk}, \mathsf{msk})$, which is randomly selected from the key domain $\mathbb{K}$ upon the wallet's setup. Following, the addresses' keys are derived from a master key pair as $\mathsf{sk}_i = \mathsf{msk} + \mathsf{H}(i, \mathsf{mvk}) (\text{ mod } n)$ and $\mathsf{vk}_i = \mathsf{mvk} + \mathsf{H}(i, \mathsf{mvk}) \times N$, where $i$ is the index of an address and $n, N$ are public parameters of the Elliptic Curve. A hardware wallet transaction is a tuple $\tau = (\alpha_{\mathsf{s}}, \alpha_{\mathsf{r}}, \theta_{\mathsf{pay}}, \alpha_{\mathsf{c}}, \theta_{\mathsf{change}})$. Specifically, $\alpha_{\mathsf{s}}$ denotes the sender's address, $\alpha_{\mathsf{r}}$ the receiver's address, and $\alpha_{\mathsf{c}}$ the change address. Additionally, $\theta_{\mathsf{pay}}$, $\theta_{\mathsf{change}}$ $\theta_{\mathsf{fee}}$ are the payment, change and fee funds respectively; notably, $\theta_{\mathsf{change}}$ equals to the account's balance minus the payment and the fee amounts, i.e., $\theta_{\mathsf{change}} = \mathsf{balanceOf}(\alpha_{\mathsf{s}}) - \theta_{\mathsf{pay}} - \theta_{\mathsf{fee}}$. A signed transaction is the tuple $(\tau, \mathsf{vk}, \sigma)$, where $\sigma$ is the signature of $\tau$ under the public key vk. The parties that execute an operation are the user $\mathcal{U}$, i.e., the owner of the wallet, the client $\mathcal{D}$, i.e., the device to which the hardware connects, and the hardware $\mathcal{H}$. Each message is associated with a session id $\mathsf{sid}' = \mathcal{UDH}$, which defines the parties with which it is related.

## 3.2 The Ideal Functionality

$\mathcal{F}_{\text{w}}$ (Figures 3.3 and 3.4) incorporates $\mathcal{F}_{\text{SIG}}$ and runs in the $\mathcal{G}_{\text{LEDGER}}$-setting, interacting with the adversary $\mathcal{A}$, a set of parties $\mathbb{P}$, and the environment $\mathcal{Z}$. It keeps the following, initially empty, items: i) $A_{[]}$: a list of lists, each containing addresses and their corresponding public keys, $(\alpha, \mathsf{vk})$; ii) $B_{[]}$: a list of lists, each containing addresses and their corresponding balance, $(\alpha, \theta)$; iii) $K_{[]}$: a list of master keys $(\mathsf{mvk}, \mathsf{msk})$. $\mathcal{F}_{\text{w}}$ realizes the following operations, with all messages containing a session id of the form $\mathsf{sid} = (\mathbb{P}, \mathsf{sid}')$:

- *Wallet setup*: Upon a setup request, it initializes the list of addresses, generates the account's master key, registers to $\mathcal{G}_{\text{LEDGER}}$, and returns the master private key to the user.
- *Client Corruption*: When $\mathcal{A}$ corrupts a client $\mathcal{D}$, $\mathcal{F}_{\text{w}}$ leaks the public keys and addresses to which $\mathcal{D}$ has access.
- *Client session initialization*: To start a new session, $\mathcal{F}_{\text{w}}$ computes and sends to $\mathcal{D}$, defined in $\mathsf{sid}'$, an assigned pass phrase; in the real world, the pass phrase acts as the authentication mechanism between the parties.
- *Address generation*: $\mathcal{F}_{\text{w}}$ requests a new public key from $\mathcal{F}_{\text{SIG}}$ and picks at random an address, with which to associate the key; it then stores the new address in the corresponding list and returns it to $\mathcal{Z}$. If the client is corrupted, the functionality leaks the address and public key to $\mathcal{A}$.
- *Balance calculation*: If $\mathcal{D}$ is honest, it queries the ledger to retrieve the blockchain; if the connected client is corrupted, it requests a chain from $\mathcal{A}$. Then, it calculates the wallet's available assets, based on the provided chain, and returns it to $\mathcal{Z}$.
- *Transaction issuing*: Upon receiving a transaction request, if $\mathcal{D}$ is corrupted, $\mathcal{F}_{\text{w}}$ leaks the transaction information to the adversary, which responds with a new transaction object. If $\mathcal{U}$ is also corrupted, $\mathcal{F}_{\text{w}}$ discards the original request and keeps the adversarial transaction, otherwise it ignores the adversary's response. Finally, $\mathcal{F}_{\text{w}}$ requests a signature from $\mathcal{F}_{\text{SIG}}$ for the accepted transaction, which it then sends to $\mathcal{G}_{\text{LEDGER}}$.

Functionality $\mathcal{F}_{\mathsf{W}}$

*Setup:* Upon receiving $(\mathsf{Setup}, \mathsf{sid})$ from some party $\mathcal{U} \in \mathbb{P}$, forward it to $\mathcal{A}$. Then add the empty list $A_{\mathcal{U}}$ to $A_{[]}$, register with $\mathcal{G}_{\mathsf{LEDGER}}$, pick the master key pair $(\mathsf{msk}_{\mathcal{U}}, \mathsf{mvk}_{\mathcal{U}}) \xleftarrow{\$} \mathbb{K}$ and add it to $K_{[]}$ and return $(\mathsf{SetupOK}, \mathsf{sid})$ to $\mathcal{U}$.

*Client Corruption:* When $\mathcal{A}$ corrupts $\mathcal{D}$, send to $\mathcal{A}$ $(\mathsf{AddressList}, \mathsf{sid}, A_{\mathcal{U}})$ and $(\mathsf{MasterPubKey}, \mathsf{sid}, \mathsf{mvk}_{\mathcal{U}})$, for every $\mathcal{U}$ such that a *Setup* session with $\mathcal{D}$ has been completed.

*Initialize Client Session:* Upon receiving $(\mathsf{InitSession}, \mathsf{sid})$ from $\mathcal{U}$, pick $\mathsf{pass}_{\mathcal{D}} \xleftarrow{\$} \{0,1\}^{\kappa}$ and send $(\mathsf{InitSession}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}})$ to $\mathcal{D}$. If $\mathcal{D}$ is corrupted, send $(\mathsf{InitSession}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}})$ to $\mathcal{A}$ and wait for a response $(\mathsf{InitSessionOK}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}})$. Finally, send $(\mathsf{Session}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}})$ to $\mathcal{U}$.

*Generate Address:* Upon receiving $(\mathsf{GenAddr}, \mathsf{sid})$ from $\mathcal{U}$, send $(\mathsf{KeyGen}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{SIG}}$ and wait for a response $(\mathsf{Verification\ Key}, \mathsf{sid}, \mathsf{vk})$. Then pick $\alpha \xleftarrow{\$} \{0,1\}^{\ell}$ and add $(\alpha, \mathsf{vk})$ to $A_{\mathcal{U}}$. If $\mathcal{D}$ is corrupted, send $(\mathsf{Address}, \mathsf{sid}, (\alpha, \mathsf{vk}))$ to $\mathcal{A}$ and wait for a response $(\mathsf{AddressOK}, \mathsf{sid}, \alpha')$. If $\mathcal{U}$ is corrupted, set $a = \alpha'$, else set $a = \alpha$; finally, return $(\mathsf{Address}, \mathsf{sid}, a)$ to $\mathcal{U}$.

*Calculate Balance:* Upon receiving $(\mathsf{GetBalance}, \mathsf{sid})$ from $\mathcal{U}$, send $(\mathsf{Read}, \mathsf{sid})$ to $\mathcal{G}_{\mathsf{LEDGER}}$ and wait for a response $(\mathsf{Read}, \mathsf{sid}, \mathcal{C})$. If $\mathcal{D}$ is corrupted, send $(\mathsf{Read}, \mathsf{sid})$ to $\mathcal{A}$ and, upon receiving the response $(\mathsf{Read}, \mathsf{sid}, \mathcal{C}')$, set $\mathcal{C} = \mathcal{C}'$. Then initialize the list $B_{\mathcal{U}} \in B_{[]}$ with $(a, 0)$ for every address $(a, \cdot)$ in $A_{\mathcal{U}}$, and $\forall \tau \in \mathcal{C}$, i.e., the ordered transactions in the ledger s.t. $\tau = (\alpha_s, \alpha_r, \theta_{\mathsf{pay}}, \alpha_c, \theta_{\mathsf{change}})$, do:

- If $\exists(\alpha_s, \cdot) \in A_{\mathcal{U}}$, update the entry $(\alpha_s, \theta_{\mathsf{past}}) \in B_{\mathcal{U}}$ to $(\alpha_s, 0)$;
- If $\exists(\alpha_r, \cdot) \in A_{\mathcal{U}}$, update the entry $(\alpha_r, \theta_{\mathsf{past}}) \in B_{\mathcal{U}}$ to $(\alpha_r, \theta_{\mathsf{past}} + \theta_{\mathsf{pay}})$;
- If $\exists(\alpha_c, \cdot) \in A_{\mathcal{U}}$, update the entry $(\alpha_c, \theta_{\mathsf{past}}) \in B_{\mathcal{U}}$ to $(\alpha_c, \theta_{\mathsf{past}} + \theta_{\mathsf{change}})$;

Finally, compute $b = \sum_{(\cdot,\theta) \in B_{\mathcal{U}}} \theta$ and send $(\mathsf{Balance}, \mathsf{sid}, b)$ to $\mathcal{U}$.

Figure 3.3: The ideal hardware wallet functionality. (Part 1)

Functionality $\mathcal{F}_{\mathsf{w}}$

*Issue Transaction:* Upon receiving $(\mathsf{IssueTX}, \mathsf{sid}, (\alpha_r, \theta_{\mathsf{pay}}, \theta_{\mathsf{fee}}))$ from $\mathcal{U}$, if $\mathcal{D}$ is corrupted, forward the message to $\mathcal{A}$ and wait for a response $(\mathsf{IssueTx}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}}, (\alpha'_r, \theta'_{\mathsf{pay}}, \theta'_{\mathsf{fee}}))$. If $\mathcal{U}$ is corrupted, set $(\alpha_r, \theta_{\mathsf{pay}}, \theta_{\mathsf{fee}}) = (\alpha'_r, \theta'_{\mathsf{pay}}, \theta'_{\mathsf{fee}})$. Then, find $(\alpha_{in}, \theta_{in}) \in B_{\mathcal{U}} : \theta_{in} \geq \theta_{\mathsf{pay}} + \theta_{\mathsf{fee}}$. If such entry exists, do the following: i) compute an address $\alpha_c$ and its public key $\mathsf{vk}_c$, as per the *Generate Address* interface; ii) set $\theta_{\mathsf{change}} = \theta_{in} - \theta_{\mathsf{pay}} - \theta_{\mathsf{fee}}$ and $\tau = (\alpha_{in}, \alpha_{out}, \theta_{\mathsf{pay}}, \alpha_c, \theta_{\mathsf{change}})$; iii) send $(\mathsf{Sign}, \mathsf{sid}, \tau)$ to $\mathcal{F}_{\mathsf{SIG}}$ and wait for $(\mathsf{Signature}, \mathsf{sid}, \tau, \sigma)$; iv) find $(\alpha_{in}, \mathsf{vk}) \in A_{\mathcal{U}}$ and set $\tau_\sigma = (\tau, \mathsf{vk}, \sigma)$. Then, if $\mathcal{D}$ is corrupted, send $(\mathsf{Address}, \mathsf{sid}, \alpha_c, \mathsf{vk}_c)$ and $(\mathsf{Submit}, \mathsf{sid}, \tau_\sigma)$ to $\mathcal{A}$ and wait for the response $(\mathsf{SubmitOK}, \mathsf{sid})$. Finally, send $(\mathsf{Submit}, \mathsf{sid}, \tau_\sigma)$ to $\mathcal{G}_{\mathsf{LEDGER}}$.

Figure 3.4: The ideal hardware wallet functionality. (Part 2)

## 3.3 The Real-World Hybrid Setting

The hybrid setting, which realizes the ideal functionality, consists of the human $\pi_{human}$, client $\pi_{client}$, and hardware $\pi_{hw}$ protocols, each defining the set of operations run by the corresponding parties.

**Human Protocol.** The user $\mathcal{U}$ interacts with $\mathcal{D}$, $\mathcal{H}$, and $\mathcal{Z}$, and runs the protocol $\pi_{human}$ (Figure 3.5), which defines the following, initially empty, items: i) $T$: a list of transactions $\tau = (\alpha_r, \theta_{\mathsf{pay}}, \theta_{\mathsf{fee}})$; ii) $S$: a list of client sessions sid. Under the model, a session is initialized when $\mathcal{U}$ connects the hardware module to the client. $\mathcal{U}$ assigns a random pass phrase $\mathsf{pass}_{\mathcal{D}}$ to each client, which is chosen upon session initialization. $\mathcal{U}$ keeps track of the initiated sessions and pending transactions. However, $\mathcal{U}$ does not perform complex computations, such as verifying a signature, or maintain a large state, like the entire list of generated addresses. Instead, $\mathcal{U}$ has a memory $T$, which contains only the pending transactions, and also is capable of performing simple computations (like addition/subtraction) and equality checks between strings.

**Client Protocol.** The client $\mathcal{D}$ interacts with $\mathcal{U}$, the hardware wallet $\mathcal{H}$, and the environment $\mathcal{Z}$. The protocol $\pi_{client}$ (Figure 3.6) defines the following items: i) mvk: the master public key of the wallet; ii) $i$: the key derivation index; iii) $\mathsf{pass}_{\mathcal{D}}$: the pass phrase assigned by $\mathcal{U}$; iv) $A_{client}$: a list of the account's addresses; v) $T_{utxo}$: a list of

Protocol $\pi_{human}$

*Setup:* Upon receiving $(\mathsf{Setup}, \mathsf{sid})$ from $\mathcal{Z}$, forward it to $\mathcal{H}$, and initialize $T$ to empty. Then, upon receiving $(\mathsf{SetupOK}, \mathsf{sid})$ from $\mathcal{H}$, forward it to $\mathcal{Z}$.

*Initialize Client Session:* Upon receiving $(\mathsf{InitSession}, \mathsf{sid})$ from $\mathcal{Z}$, pick $\mathsf{pass}_{\mathcal{D}} \xleftarrow{\$} \{0,1\}^{\kappa}$ and send $(\mathsf{InitSession}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}})$ to $\mathcal{D}$. Upon receiving $(\mathsf{InitSession}, \mathsf{sid}, \mathsf{pass}'_{\mathcal{D}})$ from $\mathcal{H}$, if $\mathsf{pass}'_{\mathcal{D}} = \mathsf{pass}_{\mathcal{D}}$, add $\mathsf{pass}_{\mathcal{D}}$ to $S$ and send $(\mathsf{Session}, \mathsf{sid}, \mathsf{pass}'_{\mathcal{D}})$ to $\mathcal{H}$ and $\mathcal{Z}$.

*Generate Address:* Upon receiving $(\mathsf{GenAddr}, \mathsf{sid})$ from $\mathcal{Z}$, forward it to $\mathcal{D}$ and wait for two messages, $(\mathsf{Address}, \mathsf{sid}, \alpha_{client})$ from $\mathcal{D}$ and $(\mathsf{Address}, \mathsf{sid}, \alpha_{hw})$ from $\mathcal{H}$. If $\alpha_{client} = \alpha_{hw}$, send $(\mathsf{Address}, \mathsf{sid}, \alpha_{hw})$ to $\mathcal{Z}$.

*Calculate Balance:* Upon receiving $(\mathsf{GetBalance}, \mathsf{sid})$ from $\mathcal{Z}$, forward it to $\mathcal{D}$. Then, upon receiving $(\mathsf{Balance}, \mathsf{sid}, b)$ from $\mathcal{D}$, forward it to $\mathcal{Z}$.

*Issue Transaction:* Upon receiving $(\mathsf{IssueTX}, \mathsf{sid}, \tau)$ from $\mathcal{Z}$, such that $\tau = (\alpha_r, \theta_{\mathsf{pay}}, \theta_{\mathsf{fee}})$, add $\tau$ to $T$ and forward the message to $\mathcal{D}$. Upon receiving $(\mathsf{CheckTx}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}}, \tau', b')$ from $\mathcal{H}$, if $\mathsf{pass}_{\mathcal{D}} \in S$, $\tau' \in T$ and $b' = b - \theta_{\mathsf{pay}} - \theta_{\mathsf{fee}}$, remove $\tau'$ from $T$ and send $(\mathsf{IssueTx}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}}, \tau)$ to $\mathcal{H}$.

Figure 3.5: The "human" protocol run by $\mathcal{U}$.

address balances. $\mathcal{D}$ acts a proxy between $\mathcal{U}$ and $\mathcal{H}$, providing connectivity to the ledger and executing blockchain-related operations, such as computing the account's balance. During the address generation, $\mathcal{D}$ retrieves the public key from $\mathcal{H}$; in practice this is optional, as $\mathcal{D}$ can generate an address independently, via the derivation process of the hierarchical deterministic wallets.

Protocol $\pi_{client}$

*Initialize Client Session:* Upon receiving $(\mathsf{InitSession}, \mathsf{sid}, p)$ from $\mathcal{U}$, forward it to $\mathcal{H}$; upon receiving $(\mathsf{MasterPubKey}, \mathsf{sid}, k)$ from $\mathcal{H}$, set $\mathsf{pass}_{\mathcal{D}} = p$, $\mathsf{mvk} = k$ and $i = 1$.

*Generate Address:* Upon receiving $(\mathsf{GenAddr}, \mathsf{sid})$ from $\mathcal{U}$, forward it to $\mathcal{H}$. Then, upon receiving $(\mathsf{PubKey}, \mathsf{sid}, \mathsf{vk}_i)$ from $\mathcal{H}$, compute $\alpha_i = \mathsf{H}(\mathsf{vk}_i)$, set $i = i + 1$, and add $\alpha_i$ to $A_{client}$. Finally, send $(\mathsf{Address}, \mathsf{sid}, \alpha_i)$ to $\mathcal{U}$.

*Calculate Balance:* Upon receiving $(\mathsf{GetBalance}, \mathsf{sid})$ from $\mathcal{U}$, send $(\mathsf{Read}, \mathsf{sid})$ to $\mathcal{G}_{\mathsf{LEDGER}}$. Upon receiving $(\mathsf{Read}, \mathsf{sid}, \mathcal{C})$ from $\mathcal{G}_{\mathsf{LEDGER}}$, set $T_{utxo}$ to the empty list and $\forall \tau \in \mathcal{C}$, i.e., the ordered transactions in the ledger s.t. $\tau = (\alpha_s, \alpha_r, \theta_{\mathsf{pay}}, \alpha_c, \theta_{\mathsf{change}})$, do:

- If $\alpha_s \in A_{client}$, update the entry $(\alpha_s, \theta_{\mathsf{past}}) \in T_{utxo}$ to $(\alpha_s, 0)$;
- If $\alpha_r \in A_{client}$, update $(\alpha_r, \theta_{\mathsf{past}}) \in T_{utxo}$ to $(\alpha_r, \theta_{\mathsf{past}} + \theta_{\mathsf{pay}})$;
- If $\alpha_c \in A_{client}$, update $(\alpha_c, \theta_{\mathsf{past}}) \in T_{utxo}$ to $(\alpha_c, \theta_{\mathsf{past}} + \theta_{\mathsf{change}})$.

Finally, compute $b = \sum_{(\cdot, \theta) \in T_{utxo}} \theta$ and send $(\mathsf{Balance}, \mathsf{sid}, b)$ to $\mathcal{U}$.

*Issue Transaction:* Upon receiving $(\mathsf{IssueTX}, \mathsf{sid}, \tau)$ from $\mathcal{U}$, such that $\tau = (\alpha_r, \theta_{\mathsf{pay}}, \theta_{\mathsf{fee}})$, send $(\mathsf{SignTx}, \mathsf{sid}, \mathsf{pass}_{\mathcal{D}}, \tau, T_{utxo})$ to $\mathcal{H}$. Upon receiving $(\mathsf{ChangeIndex}, \mathsf{sid}, idx)$, set $i = idx$ and compute and store the change address and its public key, as in the *Generate Address* interface. Then, upon receiving $(\mathsf{SignTx}, \mathsf{sid}, \tau_\sigma)$ from $\mathcal{H}$, send $(\mathsf{Submit}, \mathsf{sid}, \tau_\sigma)$ to $\mathcal{G}_{\mathsf{LEDGER}}$.

Figure 3.6: The client protocol run by $\mathcal{D}$.

**Hardware Wallet Protocol.** The hardware $\mathcal{H}$ interacts with $\mathcal{D}$ and $\mathcal{U}$ and runs the protocol $\pi_{hw}$ (Figure 3.7), which defines the following items: i) $i$: the key derivation index, ii) $S$: a list of the active client sessions, iii) $(\mathsf{mvk}, \mathsf{msk})$: the master key pair of the

wallet, and iv) $A$: a list that contains tuples like $(i, \alpha_i, \mathsf{sk}_i, \mathsf{vk}_i)$, where $i$ is a key derivation index and $\alpha_i, (\mathsf{sk}_i, \mathsf{vk}_i)$ a generated address and its corresponding key. $\mathcal{H}$ can perform some specific complex computations, such as hashing or signature issuing, has limited memory, and completely lacks network connectivity.

Protocol $\pi_{hw}$

*Setup:* Upon receiving $(\mathsf{Setup}, \mathsf{sid})$ from $\mathcal{U}$, initialize $S$ and $A$ to empty lists. Then compute $(\mathsf{mvk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$, set $i = 1$, and return $(\mathsf{Setup}, \mathsf{sid}, \mathsf{msk})$ to $\mathcal{U}$.

*Initialize Client Session:* Upon receiving $(\mathsf{InitSession}, \mathsf{sid}, \mathsf{pass}_\mathcal{D})$ from $\mathcal{D}$, forward it to $\mathcal{U}$. Upon receiving $(\mathsf{Session}, \mathsf{sid}, \mathsf{pass}'_\mathcal{D})$ from $\mathcal{U}$, add $\mathsf{pass}'_\mathcal{D}$ to $S$ and send $(\mathsf{MasterPubKey}, \mathsf{sid}, \mathsf{mvk})$ to $\mathcal{D}$.

*Generate Address:* Upon receiving $(\mathsf{GenAddr}, \mathsf{sid})$ from $\mathcal{D}$, compute $(\mathsf{sk}_i, \mathsf{vk}_i) = \mathsf{HierarchicalKeyGen}(\mathsf{msk}, i)$ and $\alpha_i = \mathsf{H}(\mathsf{vk}_i)$. Then, store $(i, \alpha_i, \mathsf{sk}_i, \mathsf{vk}_i)$ to $A$, set $i = i + 1$, and return $(\mathsf{Address}, \mathsf{sid}, \alpha_i)$ to $\mathcal{U}$ and $(\mathsf{PubKey}, \mathsf{sid}, \mathsf{vk}_i)$ to $\mathcal{D}$.

*Issue Transaction:* Upon receiving $(\mathsf{SignTx}, \mathsf{sid}, \mathsf{pass}_\mathcal{D}, \tau, T_{utxo})$ from $\mathcal{D}$, where $\tau = (\alpha_r, \theta_{\mathsf{pay}}, \theta_{\mathsf{fee}})$, find an entry $(\alpha_{in}, \theta_{\mathsf{in}}) \in T_{utxo} : \theta_{\mathsf{in}} \geq \theta_{\mathsf{pay}} + \theta_{\mathsf{fee}}$. If such entry exists, then: i) find $(\cdot, \alpha_{in}, \mathsf{sk}_{in}, \mathsf{vk}_{in}) \in A$; ii) compute $\theta_{\mathsf{change}} = \theta_{\mathsf{in}} - \theta_{\mathsf{pay}} - \theta_{\mathsf{fee}}$; iii) create an address $\alpha_c$, as in the *Generate Address* interface; iv) compute $b = \sum_{(\cdot,\theta) \in T_{utxo}} \theta$ and set $b' = b - \theta_{\mathsf{pay}} - \theta_{\mathsf{fee}}$. Then, send $(\mathsf{ChangeIndex}, \mathsf{sid}, i)$ to $\mathcal{D}$ and $(\mathsf{CheckTx}, \mathsf{sid}, \mathsf{pass}_\mathcal{D}, \tau', \mathsf{balance}')$ to $\mathcal{U}$, where $\tau' = (\alpha_r, \theta_{\mathsf{pay}}, \theta'_{\mathsf{fee}})$. Upon receiving $(\mathsf{IssueTx}, \mathsf{sid}, \mathsf{pass}_\mathcal{D}, \tau)$ from $\mathcal{U}$, set $\tau = (\alpha_{in}, \alpha_r, \theta_{\mathsf{pay}}, \alpha_c, \theta_{\mathsf{change}})$, compute $\tau_\sigma = (\tau, \mathsf{vk}_{in}, \mathsf{Sign}(\tau, \mathsf{sk}_{in}))$ and send $(\mathsf{SignTx}, \mathsf{sid}, \tau_\sigma)$ to $\mathcal{D}$.

Figure 3.7: The hardware protocol run by $\mathcal{H}$.

## 3.4 Security Analysis

We now assess the security of the proposed model, to prove the security of the hybrid setting w.r.t. $\mathcal{F}_\mathsf{w}$. Our model, as defined in $\mathcal{F}_\mathsf{w}$, incorporates the following problematic scenarios:

1) *Privacy loss*: when $\mathcal{A}$ corrupts a client, he accesses the account's public keys, addresses, and balance;

2) *Payment attack*: during transaction issuing, $\mathcal{A}$ may tamper with the inputs to alter the payable amounts and/or the receiving address; this attack is successful if the client is corrupted *and* the user deviates from their expected behavior, i.e., does not reject the malicious data;

3) *Address generation attack*: $\mathcal{A}$ may tamper with address generation on the client's side, providing an adversarial address to the user; again, this attack is successful if the client is corrupted *and* the user deviates from their expected behavior, i.e., does not cross-check the addresses provided by the client and the hardware;

4) *Chain attack*: $\mathcal{A}$ may tamper with the balance calculation by providing a malicious chain to the wallet; this family of attacks is successful if only the client is corrupted, i.e., regardless if the user follows the protocol.

Although the first three scenarios have been previously identified by empirical studies [doc18, GAK17], the *chain attack* is more nuanced. Under our model, the client is the only party that connects to the network. Therefore, a corrupted client can mount an eclipse attack [HKZG15], like the chain attack showcased by the following example.

Let $\mathcal{C}_{\top}$ be the honest chain, i.e., the longest chain available on the network. Also, let $\tau$ be a transaction which transfers $\theta_{\mathsf{in}}$ funds to an address $\alpha$; $\tau$ is published in the $j$-th block $\mathcal{B}_j$ of $\mathcal{C}_{\top}$. Let $\mathcal{C} \prec \mathcal{C}_{\top}$ be a prefix of $\mathcal{C}_{\top}$, which does not include $\mathcal{B}_j$. Also assume that $\mathcal{C}$ includes a number of transactions, which transfer an aggregate amount $\theta_{\mathsf{past}}$ to $\alpha$. When the hardware requests a chain, the adversary $\mathcal{A}$ supplies $\mathcal{C}$ instead of $\mathcal{C}_{\top}$. Hence, during balance calculation, the wallet assumes that $\alpha$ owns only $\theta_{\mathsf{past}}$ funds, instead of the correct amount $\theta_{\mathsf{past}} + \theta_{\mathsf{in}}$. Following, when the user attempts to spend the assets owned by $\alpha$, the wallet does not consume the latest UTxO (of $\theta_{\mathsf{in}}$ assets), as it is not aware of it. This may result in various hazards, from displaying a wrong balance to the user to failing to recover these funds, e.g., if the wallet deletes old (i.e., consumed) addresses.

To prove the security of our model, we show that the hybrid setting of Section 3.3 (denoted by $\pi_{hybrid}$) securely realizes the wallet ideal functionality $\mathcal{F}_{\mathsf{w}}$ of Section 3.2. In the ideal execution, $\mathcal{G}_{\mathsf{LEDGER}}$ uses the wrapper IdealValidateWrapper of Section 3.1, whereas in the real world it utilizes RealValidateWrapper. Theorem 1 is restricted to environments that do not corrupt the hardware party $\mathcal{H}$; in effect, our analysis does not consider attacks mounted by the hardware's manufacturer or physical attacks.

**Theorem 1** (Hybrid Wallet)**.** *Let the hybrid setting $\pi_{hybrid}$, which is parameterized by a signature scheme $\Sigma$ and a hash function* H*, and interacts with $\mathcal{G}_{\mathsf{LEDGER}}$, parameterized by*

RealValidateWrapper. $\pi_{hybrid}$ *securely realizes the ideal functionality* $\mathcal{F}_{\text{W}}$*, which interacts with* $\mathcal{G}_{\text{LEDGER}}$ *parameterized by* IdealValidateWrapper*, if and only if* $\Sigma$ *is EUF-CMA and* H *is an instantiation of the random oracle.*

*Proof. The "if" part.* For this part of the theorem we assume that the environment $\mathcal{Z}$ can distinguish between the ideal and the real execution with non-negligible probability. We then describe a "generic" simulator $\mathcal{S}$ for each adversary $\mathcal{A}$, which emulates the interfaces defined by the functionality. $\mathcal{S}$ also runs an internal copy of $\mathcal{A}$ and forwards the outputs of its computations to $\mathcal{A}$. We then construct a forger $G$ that runs an internal simulation of the environment $\mathcal{Z}$. Thus, for each property assumption, we show that there exists a "bad" event $E$ such that, as long as $E$ does not occur, the two executions are statistically close. However, when $E$ occurs, the environment $\mathcal{Z}$ distinguishes between the executions. At this point, $G$ uses $\mathcal{Z}$ and outputs the values that break the property under question. Therefore, since, by assumption, $E$ occurs with non-negligible probability, we show that $G$ is also successful with non-negligible probability.

*The simulator.* Let us now construct the generic simulator $\mathcal{S}$. For every interface defined by the ideal functionality, $\mathcal{S}$ completes the operations in the manner defined by the protocols in the hybrid setting. It internally runs a copy of the adversary $\mathcal{A}$ and forwards the necessary messages to it as defined in the hybrid setting. So, the view of the $\mathcal{A}$ when it interacts with $\mathcal{S}$ is the same as in the case it operates in the real world setting. $\mathcal{S}$ performs as follows:

- Any inputs received from the environment $\mathcal{Z}$, forward them to the internal copy of $\mathcal{A}$. Moreover, forward any output from $\mathcal{A}$ to $\mathcal{Z}$;
- *Party Setup:* For every party $\mathcal{P}$ for which $\mathcal{F}_{\text{W}}$ sends messages, spawn an internal simulation of $\mathcal{U}$, $\mathcal{D}$, and $\mathcal{H}$, which also interact with $\mathcal{A}$ as needed and run the protocols $\pi_{human}$, $\pi_{client}$ and $\pi_{hw}$ respectively;
- *Party Corruption:* Whenever the adversary $\mathcal{A}$ corrupts a party, $\mathcal{S}$ corrupts it in the ideal process and hands to $\mathcal{A}$ its internal state;
- *(Setup, Initialize Session, Generate Address, Issue Transaction):* For any message for these interfaces, follow the protocols $\pi_{human}$, $\pi_{client}$, and $\pi_{hw}$.

To prove the theorem regarding the properties of the signature scheme we follow the reasoning of Canetti [Can03]. We will show the proof for the *unforgeability* property of the signature scheme, as the proofs for the other properties follow similarly.

*Unforgeability.* Assume that *consistency* and *completeness* hold for $\Sigma$ and H instantiates the random oracle. In this case, the *Setup, Initialize Session* and *Generate Address* interfaces are the same in the both settings from $\mathcal{A}$'s point of view. Since, by assumption, $\mathcal{Z}$ distinguishes between the two, this occurs during the *Issue Transaction* phase, i.e., by observing a valid signature of a transaction which has not been issued by the hardware wallet.

We now construct a forger $G$ that runs a simulated copy of $\mathcal{Z}$. $G$ follows the generic simulator as above, except for the transaction issuing interface. Upon receiving $(\mathsf{Submit}, \mathsf{sid}, \tau_\sigma)$, where $\tau_\sigma = (\tau, \mathsf{vk}, \sigma)$, it checks if $\mathsf{Verify}(\tau, \mathsf{vk}, \sigma) = \mathsf{True}$. If so, it accesses the internal state of the hardware $\mathcal{H}$ and checks whether it has issued $\tau_\sigma$. If so, then it continues the simulation. Else $G$ outputs $\tau_\sigma$ as a forgery. Since, as long as this does not occur, the two executions are statistically close and, by assumption, $\mathcal{Z}$ is successful with non-negligible probability, then the probability that $G$ is also successful is non-negligible.

*The "only if" direction.* We show that if one property does not hold, the probability that the "bad" event $E$ (as above) occurs is non-negligible, so that the environment $\mathcal{Z}$ can distinguish between the real and ideal executions. Again we prove the theorem for the *unforgeability* property, as the proofs for the other properties of $\Sigma$ are constructed similarly. Also, we conclude the proof with the *address randomness* property, which accompanies the assumption that H instantiates a random oracle.

*Unforgeability.* Assume that *unforgeability* does not hold for $\Sigma$, i.e., there exists a forger $G$ for $\Sigma$. When $G$ wishes to obtain a signature for some message $m$, the environment sends the message $(\mathsf{IssueTx}, \mathsf{sid}, m)$ and forwards the response to $G$. When $G$ outputs a forgery $\tau_\sigma = (\tau, \mathsf{vk}, \sigma)$, if $\tau$ has been previously signed, $\mathcal{Z}$ halts. Else it sends $\tau_\sigma$ to $\mathcal{G}_{\mathsf{LEDGER}}$ and observes the ledger's updates. In the ideal setting, the transaction will be rejected by the validation predicate and will never be included in the ledger. However, in the real world, the probability that the transaction is accepted and eventually published in the ledger is non-negligible, as the forgery output by $G$ is successful with non-negligible probability.

*Address randomness.* Assume that all properties for $\Sigma$ hold. Now, the *Setup, Initialize Session* and *Issue Transaction* interfaces are similar in both settings. So, if the two worlds are distinguishable, then this is due to address generation. Specifically, $\mathcal{Z}$ should observe addresses which are not uniformly distributed over the space of possible addresses. This is impossible in the ideal world, by construction. However, if this was true for the real world, then H would not instantiate the random oracle. Therefore, by assumption, it is

impossible for $\mathcal{Z}$ to distinguish between the two worlds. □

Our model, and the accompanying Theorem 1, can be used to prove the security of any hardware wallet that realizes the hybrid setting. Specifically, to evaluate a wallet implementation, we first need to identified whether it faithfully realizes the three protocols. Under this premise, the security assumptions of its building components should be evaluated. More precisely, the signature algorithm that the wallet uses must be EUF-CMA, the hash function must simulate the random oracle, and the communication channels between the parties must be secure. Typical examples of such components are the *ECDSA* [JMV01] signature algorithm and a *SHA-2* [PvW08] hash function. If these assumptions hold, then the wallet is secure under our model.

Finally, one core assumption that deems further investigation is the honesty of the human user of the wallet. In Section 3.3, we presented a well-defined protocol that the user should follow. As shown in Section 3.4, as long as the parties follow the defined protocols faithfully, and the cryptographic primitives used are strong enough, the hardware wallet is secure. The integrity of transaction issuing and address generation is based, however, on the assumption that the user follows $\pi_{human}$ correctly.

$\pi_{human}$ requires from the user to identify malicious data, by comparing the data shown by the client with the data shown by the hardware. In practice, the presented data is long hexadecimals (i.e., Bitcoin addresses). However, even though such comparison is trivial for software, people are prone to errors. Comparison of long hexadecimal strings has been proved a challenging procedure, with research [HLS+09, TBB+17, UKA07] suggesting that it is unrealistic to expect a perfect comparison of cryptographic hashes, as humans find this process difficult. In real world scenarios, the user typically acts hastily, which often causes deviations from the "honest" behavior. Additionally, expecting the user to manually copy a Bitcoin address from the hardware's screen defeats the usability purposes of the wallets. Thus, it is highly possible that the user simply copies the address directly from the client. Hence, such usability difficulties of the compare-and-confirm process open an attack vector for the payment and address generation attacks.

We model the probability of a user diverging from $\pi_{human}$ as a random variable $R_h$. The distribution of $R_h$ may vary, depending both on the vigilance of the user and usability parameters. For example, a user allowing all requests to be completed without checks, e.g., because the process takes too long and the data is difficult to read, demonstrates $R_h$ close to $1$. A user who carefully checks the data, e.g., because the hardware presents

it in such way that captures the user's attention, demonstrates $R_h$ closer to $0$. Finally, another factor that affects $R_h$ is address length; the longer the address, the more difficult to read and compare.

## 3.5 Product evaluation

The evaluation of commercial hardware wallet products was performed on September 2018. At that time, the hardware wallets suggested by bitcoin.org were Digital Bitbox, KeepKey, Ledger, and Trezor. The latter $3$ had an embedded screen to present information to the user, so we focus on them. We manually inspected these wallets, identified their protocols, and mapped them to our model.

These implementations bare significant similarities to each other and our model. Although the wallets did have different low-level implementations, the protocols that they execute are captured by the hybrid setting of Section 3.3. This similarity, between our model and the actual implementations, reaffirmed the correctness of previous empirical studies, which suggest that the Ledger wallets are prone to the payment [GAK17] and address generation [doc18] attacks. The wallets were subject to these attacks when the client is dishonest and secure only if the cryptographic primitives are secure *and* the user does not deviate from the defined protocol, i.e., successfully identifies tampered data. Moreover, the instantiation of our model to the three implementations suggests that the wallets were prone to chain attacks, as discussed in Section 3.4.

For each implementation we focused on the two core wallet operations: *address generation* and *transaction issuing*. Since all implementations were susceptible to chain attacks, we focused on the viability of payment and address attacks in each case. We showed that Trezor and KeepKey were secure against payment and address attacks, as long as the user follows the protocol and verifies the data, whereas Ledger wallets were prone to address attacks, due to divergence from our model. We expect this type of evaluation to become an industry standard for hardware wallets, so that vendors can improve the security and performance of their products by employing formal verification methods, instead of empirical techniques.

**Trezor and KeepKey.** We investigated the implementation of the Trezor Model T and KeepKey hardware wallets. Both products are implemented similarly, so we focused only on Trezor. Trezor provides a touch screen for both displaying information and receiving input from the user. Based on the developer's guide [Tre18b], we next

describe an abstraction of Trezor's behavior under our model. During address generation, Trezor requires that the user connects the token to the client and unlocks it, i.e., the user *initiates a session* similar to our model definition. The client then retrieves the address from the hardware token and displays it to the user. The hardware also displays the address, as long as the "Show on Trezor" option is enabled. If this option is disabled, then the user cannot verify the client's address and is prone to an address attack, i.e., the client might display a malicious address which the user cannot cross-check with the hardware wallet. However, the user manual does urge the user to always check the two addresses [Tre18c] to avoid such attack scenarios. During transaction issuing, the user again connects the device to the client and unlocks it. Then they initiate a transaction by giving to the client the recipient's address, and the payment and fee amounts, similarly to our hybrid model setting. The client initiates the transaction signing process with the hardware by providing this data, which the token then displays to the user for verification. After the user has verified the transaction, the hardware communicates with the client and signs the needed data. Again, given our high level investigation, this process matches the communication steps that our model describes.

**Ledger.** We investigate the implementation of Ledger Nano S according to the user manual [Led18] and our own analysis. Like Trezor, before performing any operation the user is required to initiate a session by connecting the hardware to the client and unlocking it. The hardware provides a small screen for displaying information and a pair of two buttons for receiving commands from the user. During address generation, the client displays the newly generated address to the user. However, there is no option for the hardware wallet to also display the address,[1] so that the user can cross-check and verify the two. This is a clear divergence from our model and allows for address attacks, e.g., by a corrupted client that displays a malicious address to the user. Transaction issuing is also similar to Trezor and captured by our model. The user inputs the transaction's data to the client, i.e., the recipient's address and the payment and fee amounts. The client forwards this data to the hardware, which displays it to the user for verification. After receiving the user's confirmation, the hardware interacts with the client to sign and publish the transaction on the blockchain.

[1] Ledger issued a firmware update to address this issue and allow both the client and the hardware to generate and display the address; however, the update process is manual and often neglected by users.

# Chapter 4

# Account Management in Proof-of-Stake Ledgers

Blockchain protocols based on the Proof-of-Stake (PoS) paradigm depend — by nature — on the active participation of stakeholders. Specifically, in PoS, the set of potentially eligible parties is comprised of "stakeholders", i.e., parties who own some amount of the digital assets which are maintained by the ledger. Importantly, this set is open, i.e., parties can arbitrarily join and leave, while also remaining pseudonymous. Therefore, the eligible party, dubbed the "minter", is selected proportionally to its stake, i.e., the amount of assets that it owns; the details of this selection process have resulted in a number of PoS flavors and protocols.

However, the inherent *duality* of PoS digital assets, which can both be transferred between entities and used in the protocol's execution, diverges from the Proof-of-Work (PoW) setting and might result in users abstaining from engaging in the PoS mechanism. The stakeholders are expected to consistently follow the protocol's execution, by checking whether they are eligible to participate, and, in those cases and within a specific time frame, engage in transaction processing per the PoS protocol's rules [KRDO17, CGMV18, GHM+17b, Vas14]. This feature is in sharp contrast with PoW protocols, which offer a natural decoupling between the consensus layer participants who generate blocks, i.e., the *miners*, and the users of the system who issue transactions. Naturally, the set of users subsumes the miners, since e.g., the miners collect fees and may also transact using them, and a substantial number of users do not participate in the consensus protocol.

This dual nature of PoS assets raises two major considerations. First, some secret, key-related information is frequently used on behalf of an asset, thus potentially reveal-

ing critical information that weakens the asset's security or increase the attack surface against a user's wallet. For instance, in the UTxO model (implemented by Bitcoin), the public key which controls the assets is published only upon spending them; however, PoS systems cannot adapt this model directly, since the public key is revealed when participating in the protocol, i.e., (typically) before spending the funds. Therefore, using the same key for all operations both increases the attack surface against it and introduces quantum attacks, given that most implementations employ non-post-quantum secure signature schemes. Second, a computational and availability requirement is imposed on the stakeholders; therefore, in an environment where the majority of users are often offline and abstain from the protocol's execution, the security guarantees of the ledger are weakened.

The above issues are well-known and have already been informally considered by the blockchain community. For instance, some schemes propose a separation between a staking and a payment key to address the first consideration[1]. The second consideration, although seemingly unavoidable given the nature of PoS protocols, can be countermeasured by enabling the delegation of the PoS protocol participation rights, i.e., block generation and transaction validation. This mechanism would allow users to organize in "stake pools", i.e., consortiums managed by a single user, the pool "leader", who participates in the PoS protocol on behalf of all members. Stake pools bring efficiency advantages, since the set of stake pool leaders is (typically) smaller than the entire stakeholders' set. Given that some PoS protocols [KRDO17] are based on multi-party computation, introducing a *committee* of pool leaders substantially reduces the computational and communication complexity overhead.

Interestingly, PoS implementations don't typically address these concerns comprehensively. On the one hand, "delegated PoS", e.g., in Steem [Ste18] and EOS [Com18], attempts to address the second consideration by enabling the voting of delegates. However, both systems use a single key for payment and voting, thus failing to address the first consideration. On the other hand, systems like NEO [NEO18], as of September 2020, enabled only 7 consensus nodes, 5 of which are controlled by a single entity, whereas the only way of participating in the consensus mechanism is via central approval. Finally, Decred [dec19] uses a ticketing system, i.e., stakeholders buy a ticket to participate in the protocol, which is akin to using a separate key; however, it requires the locking of funds while staking, i.e., it does not allow concurrent payments and staking, a major blow to the system's usability.

[1] One such discussion is available at: `https://www.reddit.com/r/ethereum/comments/6idf2c`

Evidently, even though practical solutions do exist, a comprehensive and formal treatment of a PoS system's account management is less well-researched. In fact, due to the very little systematization of the PoS wallets' security and the lack of concise guidelines, developers often resolve to ad hoc solutions. Given that PoS systems are increasingly gaining momentum, it is imperative that this problem receives a thorough and formal treatment. The results of this chapter fill this gap by acting as a guideline for system architects and developers, aiming to better wallet implementations in terms of safety, as well as enable robust designs with improved performance. A further important motivation behind our work is the current low level of decentralization in PoS systems. As illustrated above, some projects are yet to allow stakeholders to practice staking, opting instead for a closed set of block producing nodes. Even in cases where stakeholders are arguably in control of their stake, they may choose their stake representatives from a very narrow set of accounts; for instance, the EOS block producing nodes are only $21$ at any given time. Our work aims at alleviating such centralization tendencies by enabling every user to either participate on their own or assign their stake to any delegate of their choice. Consequently, the formalization of the PoS wallet is a stepping stone for future research, given that the wallet is the gateway through which users interact with the system, and a core element of the consensus protocol itself. As a result, the composable nature of our framework allows future research to employ it without composability concerns about its underlying implementation.

**Related Work.** Despite the large number of available PoS cryptocurrencies, formal wallet research has been rather sparse and limited so far. A widely implemented wallet standard is the HD Wallet Standard BIP32 [Wui18], based on the idea of deterministic wallets [M+14]. Gutoski and Stebila [GS15] studied security in the presence of partial key leakage, focusing on BIP32-compliant wallets, whereas Courtois *et al.* [CEV14] investigated the state of wallets and key management for Bitcoin [Nak08a]. We note that these works focus on PoW-based systems and thus do not consider issues such as stake delegation and protocol participation. In the PoS setting, a related work is the Bitshares' delegation PoS mechanism [SL17], which, however, does not provide any formal model or proof of security. Furthermore, a formal specification of Cardano's wallet is also available [DC18], although it focuses on the wallet's implementation, rather than the security properties and cryptographic model.

**Contributions.** This chapter provides a formal framework of account management and stake pool formation in PoS systems. In more detail, our contributions are as follows. First, we present the relevant desiderata of a PoS wallet, i.e., a set of requirements general and flexible enough to be compatible with a number of different types of actors in the system. Based on the desiderata, we provide a formal treatment of address and asset management in the form of the ideal functionality $\mathcal{F}_{\text{CoreWallet}}$, which epitomizes a PoS wallet's core. Our analysis follows in the tradition of the UC Framework [Can01] and is inspired by Canetti [Can03], which allows us to define our setting in a composable way. While formalizing the addresses and their attributes, a nuanced and overlooked, but equally important, notion emerges, namely *address malleability*. This property is also of independent interest for any setting where an object, such as an identification tag, depends on multiple attributes, such as keys or metadata. We describe various threat models and protection levels, which range from fully malleable schemes to entirely non-malleable. Given our theoretical foundations, we next define the wallet protocol, which securely realizes $\mathcal{F}_{\text{CoreWallet}}$ under standard cryptographic assumptions. The protocol is highly parametric, thus offering flexibility in the wallet's implementation. Following, we describe an address construction mechanism, which both ensures a strong level of security and covers the majority of our desiderata. Finally, we combine the core wallet funcitonality with a generic PoS ledger to form a complete PoS wallet. Specifically, we detail how the core wallet can participate in the PoS protocol and conduct payments, stake pool registration, and stake delegation, and conclude with proving the security of the stake-pooled variant of any PoS protocol, as well as describe various modes of operation, which allow for enhanced privacy and security.

## 4.1 General Desiderata

Before presenting our framework, we first identify the properties that the wallet in a PoS setting should offer. This investigation is an important step in understanding the restrictions in designing such systems, as well as evaluating the choices that a PoS protocol's designer should make. We organize the desiderata in three basic categories, based on the related system component.

**Addresses and Attributes.** In a PoS system, each account manages a set of addresses. These addresses own a non-negative amount of cryptocurrency assets and may contain various account metadata. Any PoS system should offer at minimum two

basic operations for each user's account: i) *paying* and ii) *staking*. Addresses, simply put, are strings which have cryptocurrency balances associated with them. They may also contain metadata, in the form of arbitrary attributes, which are useful for various system operations. We identify the following desiderata for addresses in a PoS setting:

- *Address Non-malleability*: assuming access to an address and the ledger, an adversary should not be able to construct a different address that shares *only some* of the address's attributes;
- *Address Uniqueness*: any two addresses with different attributes should be distinct;
- *Short Addresses*: addresses should be relatively short (in order to be usable and storage efficient);
- *Multiple Types of Addresses*: construction of more than one type of addresses should be allowed, with each type supporting a different subset of basic operations (e.g., to ban some addresses from participating in consensus);
- *Multiple Device Support*: an account should be able to exist on multiple devices that share *no joint internal state*;
- *Address Recovery*: an account should be able to identify its addresses, assuming access to the ledger and the payment keys which it controls;
- *Privacy and unlinkability:* addresses should be indistinguishable from one another and not publicly-linkable to their account.

**Basic Account Operations.** The two basic types of operations, i.e., *payment* and *staking*, can be performed independently by two separate pieces of information, denoted by $\mathcal{I}_p$ and $\mathcal{I}_s$. The main advantage of this approach is that $\mathcal{I}_p$ is reserved only for transferring funds, while staking operations require access only to $\mathcal{I}_s$. Another desirable result is the ability to recover all addresses given a master key, e.g., as implemented by HD Wallets [DFL19, GS15, Wui18]; this feature is particularly important in case the equipment which hosts the wallet is lost. We summarize the above, with some additions, as follows:

- *Account Master Key:* there should exist a "master" piece of information, e.g., a master seed, that can be used to generate all of the account's management information, i.e., its keys;

- *Staking and Payment Separation:* compromising the staking operations should not affect the payment operations (and vice-versa);
- *Payment Key Information Safety:* apart from a cryptographic hash, no other information about the payment key $\mathcal{I}_p$ should be public prior to issuing a payment;
- *Key Exposure Mitigation:* ownership of the account's assets and staking ability should be recoverable, in case the staking information $\mathcal{I}_s$ is compromised.

**Delegation Mechanism and Stake Pools.** Delegation depends on the ability of a user to give the rights over her stake to another user. This action should be distinguishable from other actions, like payment, in order to protect the users and facilitate automatic reward schemes. Therefore, the delegation desiderata are as follows:

- *Cost Effectiveness:* stake delegation, re-delegation, or pool formation should be cost effective;
- *Chain Delegation Restriction:* a limit to the number of re-delegations of the same stake unit should be enforceable;
- *Delegation Verification:* participants in the system should be able to verify the status of active delegations.

## 4.2 Address Malleability

We first introduce the notion of address malleability. We describe the adversarial threat models that might exploit it, distill the components that define a malleability attack, and organize this family of attacks based on threat levels. Next, we formalize malleability via the *malleability predicate* and present a formal addressing of these attacks. Before proceeding with the definition though, we first define the objects that underpin our constructions and explore the malleability of addresses.

An *account* comprises of multiple addresses. An address is associated with a number $g$ of *attributes*, each identifying a property of the address, which are organized as follows: i) *public* attributes are part of the address, so they are public and available without any interaction with the account's owner; ii) *semi-public* attributes are not public by default, but become public when a transaction is issued, which spends some assets that the address owns; iii) *private* attributes never become public. An *attribute list* $l$ is a vector of $g$ attributes. In our work, the fundamental list of attributes is the *address generation*

*list* $l_{\alpha,Gen}$, i.e., the list of all attributes used during the generation of an address $\alpha$. In this list, the sub-list of public attributes $d = (\delta_1, \ldots, \delta_{p-1})$ contains the attributes which are available by every party with access to the address. Of the remaining attributes, the sub-list $(\delta_p, \ldots, \delta_{i-1})$ contains the semi-public attributes, while $(\delta_i, \ldots, \delta_g)$ contains the private attributes. The parameters $p$ and $i$ depend on the inner workings of the protocol and the address generation scheme. For example, in Bitcoin, a typical address attribute is the payment key pair $(\mathsf{vkp}, \mathsf{skp})$: vkp is used to verify signatures and is *semi-public*, its hash is a *public* attribute, whereas skp, which signs transactions, is *private*.

Before exploring address malleability, a brief motivation is necessary. Address malleability is similar to the preexisting malleability notions [DDN03] and is intrinsically tied to address generation. Here, we treat it as a novel family of attack vectors due to the complex structure of PoS addresses. Namely, our framework's addresses encode information that extend beyond the simple transfer of assets and pertain to additional functionalities, like staking. Therefore, to ensure the PoS protocol's security, it is crucial to prevent an adversary $\mathcal{A}$ from constructing addresses on behalf of honest users, or manipulating honestly generated addresses.

The goal of $\mathcal{A}$ is to inflate their stake in the system by exploiting address malleability. We remind that an address is associated with the payment information $\mathcal{I}_p$, which is used to issue payments (and practically controls the assets) and the staking information $\mathcal{I}_s$, which is used to participate in the PoS consensus protocol. $\mathcal{A}$ attempts to forge an address, for which it controls the staking information $\mathcal{I}_s$ without controlling the payment information $\mathcal{I}_p$. This results in a stake "shift", via which $\mathcal{A}$ gains illegitimate control of the staking rights of assets it does not own. Importantly, this forgery should take place in such a way that the address's honest owner does not easily recognize this stake shift, unless actively looking for it.

Numerous types of adversaries would try to exploit this hazard. One such type is a malicious stake pool leader. Such attacker would operate under the — natural — assumption that the reward amount depends on the stake percentage, the staking rights of which the pool controls. This assumption is amplified given incentives mechanisms [BKKS18] which mandate that larger stake pools receive greater rewards. By deploying a generalized malleability attack, $\mathcal{A}$ would artificially increase the amount of its rewards (Figure 4.1). To deploy this attack, $\mathcal{A}$ could intercept honestly-generated addresses, which are exchanged between users, and replace them with forged addresses which are created on-the-fly. As long as $\mathcal{A}$ remains unnoticed, it may be very profitable to mount such attack on a broad network level.

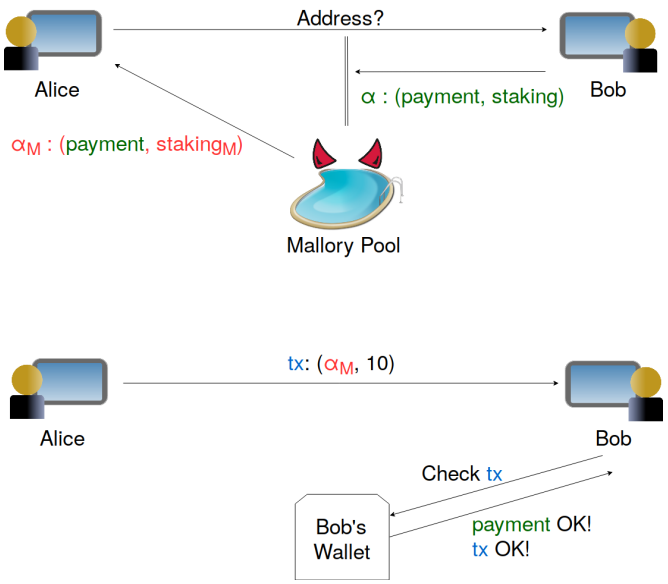


Figure 4.1: An example of a network-level, Man-in-the-Middle malleability attack. A malicious pool changes on-the-fly the staking object of one of Bob's addresses, such that it points to Mallory's stake pool. If Bob's wallet does not identify the attack, Mallory has successfully (but artificially) inflated its delegated stake. If the wallet does identify the attack, it has to transfer the funds to a correct address and impose extra fees on Bob.

Following we identify multiple types of address malleability attacks. In all cases, we assume that the adversary does not control the signing payment key skp, as this would trivialize all attacks. An address created by $\mathcal{A}$ without access to its payment key is called *forgery*. A forgery is successful if funds that are sent to it can be spent and the wallet accepts the forged address as its own, whereas it fails if funds that are sent to it can never be spent and are, therefore, "burnt".

Our analysis assumes two types of adversaries, depending on the level of information which they access. First is the *network adversary*, who has access to the ledger, observes the network, and can intercept addresses exchanged between parties. Second is the *targeting adversary*, with access to the same information as the first, but also knows the set of past transactions issued by the "victim", i.e., the honest user for which it attempts to produce a forgery. We stress that the assumptions for the targeting adversary are much stronger compared to the network adversary.

The malleability levels presented next range from *full malleability*, i.e., when no inherent protection against malleability attacks exists, to *non malleability*, i.e., where an adversary cannot create a successful forgery without access to the address's private attributes.

Defining the intermediate levels allows us to construct various address schemes, which are suitable for different needs of real world projects. For example, a fully malleable address is typically short and suitable for performance-oriented applications. In contrast, a security-oriented project would aim for the higher levels of malleability protection. The malleability levels are organized around the following three properties: i) the type of adversary, i.e., $\mathcal{A}$ is at the network level or has targeted access to the user's wallet; ii) self-verification, i.e., whether a wallet can recognize a forgery for one of its own payment keys; iii) cross-verification, i.e., whether a wallet can identify a forgery for an address which it does not own. Below we describe the malleability levels and summarize them in Table 4.1:

- *Level 1, Full Malleability:* This level enables successful forgeries from both network and targeting adversaries. A wallet checks only the payment information, so it accepts the forgery without recognizing it as such, while also both spending from and sending funds to forgeries.

- *Level 2, Full Verifiable Malleability:* Again, both types of adversaries, i.e., network and targeting, can create forgeries. However, during recovery, the wallet identifies forgeries and rejects them; we note that a "hacked" wallet can spend the assets owned by such forgery. Also all wallets, including honest ones, may send funds to forgeries.

- *Level 3, A Posteriori Malleability:* This level prohibits network adversaries. Specifically, if such adversary produces a forgery, any transaction which spends from the forgery is rejected. However, targeting adversaries can create successful forgeries as before.

- *Level 4, A Posteriori Verifiable Malleability:* This level is similar to *Level 3*, with the addition that an honest wallet can identify and reject forgeries of its own addresses, as in *Level 2*.

- *Level 5, "Sink" Malleability:* Both network and targeting adversaries are prohibited. Thus, all forgeries are rejected by the wallet and its funds are burnt. Intuitively, considering transactions as a graph, where the graph's nodes are the addresses and its edges are transactions, forgeries are "sinks" which trap all funds sent to them. However, a wallet still cannot identify a forgery of an address that it does not own.

| Level of Malleability | Network protection | Targeting protection | Self-verification | Cross-verification |
|---|---|---|---|---|
| 1 - Full | ✗ | ✗ | ✗ | ✗ |
| 2 - Verifiable | ✗ | ✗ | ✓ | ✗ |
| 3 - A posteriori | ✓ | ✗ | ✗ | ✗ |
| 4 - A posteriori verifiable | ✓ | ✗ | ✓ | ✗ |
| 5 - Sink | ✓ | ✓ | ✓ | ✗ |
| 6 - None | ✓ | ✓ | ✓ | ✓ |

Table 4.1: Comparison of the malleability levels.

- *Level 6, Non Malleability:* The highest level of protection against malleability, where a forgery is rejected by every wallet, i.e., given only an address anybody can identify whether it is honestly-generated or a forgery. Therefore, all transactions that interact in any way with a forgery are rejected.

We now formalize address malleability with the predicate $M$. The predicate returns $1$ or $0$ to denote whether an address is valid or not. Here, a valid address is either honestly-generated or a successful forgery, i.e., it is an address that the wallet accepts. The first parameter of the predicate is the set $L_{\mathcal{P}}$ of all created addresses and their attributes for a party $\mathcal{P}$; this parameter is necessary, so that the predicate compares the given address with the honestly generated ones. The second parameter is the auxiliary information $\text{aux}_M$, which takes the values recover issue or verify this information is used to instantiate different modes of the predicate and allows it to adjust its actions, thus making it more versatile. The third parameter is the address under question.

**Full Malleability.** This predicate does not allow the ideal functionality to perform any malleability checks when issuing a transaction. During recovery and verification, it first identifies the list of public attributes $d$, then outputs $1$ if these attributes have been

used in at least one address that the wallet controls. Intuitively, a forgery can be constructed by using public attributes from other addresses that the wallet has created. In the real-world, the wallet accepts an address as long as its payment key is controlled by the wallet. In a *fully malleable* construction, the predicate is instantiated with $M_{\mathsf{FM}}$ described in Algorithm 2; the function parsePubAttrs denotes the public attribute parsing of an address.

**Algorithm 2** The *fully malleable* predicate. The inputs are: i) a list of tuples of previously-generated addresses and their attributes, ii) auxiliary information on the wallet's operation, iii) the address under question.

**function** $M_{\mathsf{FM}}(L_P, \mathsf{aux}_M, \alpha)$
  **switch** $\mathsf{aux}_M$ **do**
    **case** issue
      **return** 1
    **case** verify OR recover
      $d = \mathsf{parsePubAttrs}(\alpha)$
      **for** $\delta \in d$ **do**
        **if** $\forall \alpha' \in L_P, d' = \mathsf{parsePubAttrs}(\alpha')$: $\delta \notin d'$ **then**
          **return** 0 $\triangleright$ $\delta$ not registered detected
        **end if**
      **end for**
  **return** 1
**end function**

**A Posteriori Malleability.** In an *a posteriori malleable* construction, the predicate first identifies the list $d$ of public attributes of the address $\alpha$. Then, for each such attribute $\delta$, it checks: i) if there exists an issued transaction $\tau = (\Theta, \alpha_s, \alpha_r, m)$, such that the public attributes of $\alpha_s$ include $\delta$; ii) if there exists an address that the wallet has created, such that $\delta$ is part of its public attributes. If both checks fail the predicate returns $0$, otherwise $1$. Intuitively, this construction enables malleability only for addresses whose payment key has been previously used and for which all public attributes have been used in other addresses. Therefore, as long as the payment key of the address has not been used, the scheme provides non-malleability. The *a posteriori malleable* construction, instantiated with the predicate $M_{\mathsf{PM}}^{\mathbb{T}}$ which is parameterized by the list of all transactions $\mathbb{T}$, is described in Algorithm 3.

**Algorithm 3** The *a posteriori* malleability predicate. The inputs are: i) a list of tuples of previously-generated addresses and their attributes, ii) auxiliary information on the wallet's operation, iii) the address under question.

**function** $M^{\mathbb{T}}_{\mathsf{PM}}(L_P, \mathsf{aux}_M, \alpha)$
  **switch** $\mathsf{aux}_M$ **do**
    **case** issue
      **return** 1
    **case** verify OR recover
      $d = \mathsf{parsePubAttrs}(\alpha)$
      **for** $\delta \in d$ **do**
        **if** $\forall \alpha' \in L_P, d' = \mathsf{parsePubAttrs}(\alpha')$: $\delta \notin d'$ **then**
          **return** 0
        **end if**
        **if** $\forall (\Theta, \alpha_s, \alpha_r, m) \in \mathbb{T}, d_s = \mathsf{parsePubAttrs}(\alpha_s)$: $\delta \notin d_s$ **then**
          **return** 0
        **end if**
      **end for**
      **return** 1
**end function**

**Sink Malleability.** Intuitively, a *sink malleable* address generation algorithm requires that only the owner of the honest wallet can create addresses for payment keys of the wallet. This is expressed by differentiating the behavior depending on the auxiliary information. If $\mathsf{aux}_M$ pertains to the issuing of transactions, the predicate returns $1$, i.e., accepts all addresses to which the wallet tries to send funds. For all other cases, it requires that the address is honestly generated. The sink malleable construction, with the predicate $M_{\mathsf{SM}}$, is described in Algorithm 4.

**Algorithm 4** The *sink malleable* predicate. The inputs are: i) a list of tuples of previously-generated addresses and their attributes, ii) auxiliary information on the wallet's operation, iii) the address under question.

```
function M_SM(L_P, aux_M, α)
    switch aux_M do
        case issue
            return 1
        case verify OR recover
            if ∃ l_α : (α, l_α) ∈ L_P then
                return 1                          ▷ α is registered
            end if
    return 0                                      ▷ No α is registered
end function
```

**Non Malleability.** The fully non malleable predicate behaves similarly to the sink malleable case, although it also checks transaction issuance. Specifically, when $\mathsf{aux}_M$ is issue it verifies that the recipient's address has been generated by some party via the honest process. Therefore, upon issuing a transaction, the malleability predicate checks the address of the receiver, to identify whether it is legitimate, so if the transaction is acceptable. Similarly, when verifying a transaction, the wallet identifies whether the sender's address has been properly constructed. The *fully non-malleable* construction, instantiated with the predicate $M_{\mathsf{NM}}$, is described in Algorithm 5.

## 4.3 The Core-Wallet Functionality

The major contribution of this chapter is the definition of the ideal functionality of the core wallet. The goal of this definition is to distill, in a concise way, a formal model of

**Algorithm 5** The *fully non-malleable* predicate. The inputs are: i) a list of tuples of previously-generated addresses and their attributes, ii) auxiliary information on the wallet's operation, iii) the address under question.

**function** $M_{\mathsf{NM}}(L_P, \mathsf{aux}_M, \alpha)$
  **switch** $\mathsf{aux}_M$ **do**
    **case** issue
      **if** $\exists P'$ such that $\exists l_\alpha : (\alpha, l_\alpha) \in L_{P'}$ **then**
        **return** 1
      **end if**
    **case** verify OR recover
      **if** $\exists l_\alpha : (\alpha, l_\alpha) \in L_P$ **then**
        **return** 1
      **end if**
  **return** 0
**end function**

the properties of a PoS wallet.

Our ideal functionality $\mathcal{F}_{\mathsf{CoreWallet}}$ is inspired by Canetti [Can03]. $\mathcal{F}_{\mathsf{CoreWallet}}$ (Figures 4.2 and 4.3) interacts with the ideal adversary $\mathcal{S}$ and a set of parties denoted by $\mathbb{P}$ and is parameterized by the predicate $M(\cdot,\cdot,\cdot) \to \{0,1\}$. It also keeps the, initially empty lists, $S$ of staking actions and $\mathcal{T}$ of transactions. We assume, without loss of generality, that, given a list of attributes $l_{\alpha,Gen} = (\delta_1, \ldots, \delta_g)$, $\delta_1$ is the staking key's information and $\delta_2$ is a recovery tag (which will be further investigated in the upcoming Section 4.6). We remind that access to an address implies access to its public attributes $d = (\delta_1, \ldots, \delta_{p-1})$ and, given the ledger, access to its semi-public attributes $(\delta_p, \ldots, \delta_{i-1})$ as well.

The functionality distinguishes the addresses in three types: base, pointer, and exile. As we will show in Section 4.6, each type has a specific utility; briefly, base addresses help bootstrap a wallet, pointer addresses are shorter and aim at better performance, and exile addresses are withdrawn from the PoS protocol's execution.

**Remark.** *Although $\mathcal{F}_{\mathsf{CoreWallet}}$ offers a suitable security definition, for our requirements, as it relies on the standard security properties of digital signatures and the address generation properties to be described in Section 4.5.1, it does not offer any type of forward security in the sense of Bellare and Miner [BM99]. However, protocols which require stronger security properties for their building blocks do exist. For instance, Ouroboros Praos [DGKR18] relies*

Functionality $\mathcal{F}_{\text{CoreWallet}}^{M}$

*Initialization:* Upon receiving $(\text{Init}, \text{sid})$ from $\mathcal{P} \in \mathbb{P}$, forward it to $\mathcal{S}$ and wait for $(\text{InitOk}, \text{sid})$. Then initialize the empty lists $L_{\mathcal{P}}$ of addresses and attribute lists and $K_{\mathcal{P}}$ of staking keys, and send $(\text{InitOk}, \text{sid})$ to $\mathcal{P}$.

*Address Generation:* Upon receiving $(\text{GenerateAddress}, \text{sid}, \text{aux})$ from $\mathcal{P} \in \mathbb{P}$, forward it to the $\mathcal{S}$. Upon receiving $(\text{Address}, \text{sid}, \alpha, l_\alpha)$ from $\mathcal{S}$, parse $l_\alpha$ as $(\delta_1, \ldots, \delta_g)$ and $\forall \mathcal{P}' \in \mathbb{P}$ check if $\forall (\alpha', (\delta'_1, \ldots, \delta'_g)) \in L_{\mathcal{P}'}$ it holds that $\alpha \neq \alpha'$, $\delta'_2 \neq \delta_2$, and $\forall j \in [i, \ldots, g] : \delta'_j \neq \delta_j$, i.e., the address, recovery tag, and private attributes are unique. If so, then:

1. if $\text{aux} = (\text{base})$, check that $\forall (\alpha', (\delta'_1, \ldots, \delta'_g)) \in L_{\mathcal{P}} : \delta'_1 \neq \delta_1$, i.e., the new staking key is unique,
2. else if $\text{aux} = (\text{pointer}, \text{vks})$, check that $\delta_1 = \text{vks}$,
3. else if $\text{aux} = (\text{exile})$, check that $\delta_1 = \perp$.

If the checks hold or $\mathcal{P}$ is corrupted, then insert $(\alpha, l_\alpha)$ to $L_{\mathcal{P}}$ and return $(\text{Address}, \text{sid}, \alpha)$ to $\mathcal{P}$. If $\text{aux} = (\text{base})$ also insert $\delta_1$ to $K_{\mathcal{P}}$ and return $(\text{StakingKey}, \text{sid}, \delta_1)$ to $\mathcal{P}$.

*Wallet Recovery:* Upon receiving $(\text{RecoverWallet}, \text{sid}, i)$ from $\mathcal{P} \in \mathbb{P}$, for the first $i$ elements in $L_{\mathcal{P}}$ return $(\text{Tag}, \text{sid}, \delta_2)$.

*Address Recovery:* Upon receiving a message $(\text{RecoverAddr}, \text{sid}, \alpha, i)$ from a party $\mathcal{P} \in \mathbb{P}$, if $(\alpha, l)$ is one of the first $i$ elements of $L_{\mathcal{P}}$ or $M(L_{\mathcal{P}}, \text{recover}, \alpha) = 1$, return $(\text{RecoveredAddr}, \text{sid}, \alpha)$.

*Issue Transaction:* By receiving from $\mathcal{P} \in \mathbb{P}$ the message $(\text{Pay}, \text{sid}, \Theta, \alpha_s, \alpha_r, m)$, if $\exists l_\alpha : (\alpha_s, l_\alpha) \in L_{\mathcal{P}}$ forward it to $\mathcal{S}$. Upon receiving $(\text{Transaction}, \text{sid}, \tau, \sigma)$ from $\mathcal{S}$, such that $\tau = (\Theta, \alpha_s, \alpha_r, m)$, check if $\forall (\tau', \sigma', b') \in \mathcal{T} : \sigma' \neq \sigma$, if $(\tau, \sigma, 0) \notin \mathcal{T}$, and if $M(L_{\mathcal{P}}, \text{issue}, \alpha_r) = 1$. If all checks hold, then insert $(\tau, \sigma, 1)$ to $\mathcal{T}$ and return $(\text{Transaction}, \text{sid}, \tau, \sigma)$.

Figure 4.2: The PoS core-wallet ideal functionality. (Part 1)

Functionality $\mathcal{F}_{\text{CoreWallet}}^{M}$

*Verify Transaction:* Upon receiving from $\mathcal{P} \in \mathbb{P}$ the message $(\text{VerifyPay}, \text{sid}, \tau, \sigma)$, with $\tau = (\Theta, \alpha_s, \alpha_r, m)$ for a metadata string $m$, forward it to $\mathcal{S}$ and wait for a reply message $(\text{VerifiedPay}, \text{sid}, \tau, \sigma, \phi)$. Then:

- if $M(L_{\mathcal{P}}, \text{verify}, \alpha_s) = 0$, set $f = 0$
- else if $(\tau, \sigma, 1) \in \mathcal{T}$, set $f = 1$
- else, if $\mathcal{P}$ is not corrupted and $(\tau, \sigma, 1) \notin \mathcal{T}$, set $f = 0$ and insert $(\tau, \sigma, 0)$ to $\mathcal{T}$
- else, if $(\Theta, \alpha_s, \alpha_r, m, \sigma, b) \in \mathcal{T}$, set $f = b$
- else, set $f = \phi$.

Finally, send $(\text{VerifiedPay}, \text{sid}, \tau, \sigma, f)$ to $\mathcal{P}$.
*Issue Staking:* Upon receiving $(\text{Stake}, \text{sid}, \tau_s)$ from $\mathcal{P}$, such that $\tau_s = (\text{vks}, m)$ for a metadata string $m$, forward the message to $\mathcal{S}$. Upon receiving $(\text{Staked}, \text{sid}, \tau_s, \sigma)$ from $\mathcal{S}$, if $\forall(\tau_s', \sigma', b') \in S : \sigma' \neq \sigma$, $(\tau_s, \sigma, 0) \notin S$, and $\text{vks} \in K_{\mathcal{P}}$, then add $(\tau_s, \sigma, 1)$ to $S$ and return $(\text{Staked}, \text{sid}, \tau_s, \sigma)$ to $\mathcal{P}$.
*Verify Staking:* Upon receiving from $\mathcal{P} \in \mathbb{P}$ the message $(\text{VerifyStake}, \text{sid}, \tau_s, \sigma)$, with $\tau_s = (\text{vks}, m)$, forward it to $\mathcal{S}$ and wait for $(\text{VerifiedStake}, \text{sid}, \tau_s, \sigma, \phi)$. Then find $\mathcal{P}_s$, such that $\text{vks} \in K_{\mathcal{P}_s}$, and:

- if $(\tau_s, \sigma, 1) \in S$, set $f = 1$
- else if $\mathcal{P}_s$ is not corrupted and $(\tau_s, \sigma, 1) \notin S$, set $f = 0$ and insert $(\tau_s, \sigma, 0)$ to $S$
- else if exists an entry $(\tau_s, \sigma, f') \in S$, set $f = f'$
- else set $f = \phi$ and insert $(\tau_s, \sigma, \phi)$ to $S$.

Finally, return $(\text{VerifiedStake}, \text{sid}, \tau_s, \sigma, f)$ to $\mathcal{P}$.

Figure 4.3: The PoS core-wallet ideal functionality. (Part 2)

*on a forward secure digital signature scheme, among other primitives, to provide security guarantees against fully-adaptive corruption in a semi-synchronous setting. Additionally, protocols like Cryptonote [VS13] allow an arbitrary number of parties to operate the address generation interface, instead of restricting it to the wallet's owner.*

## 4.4 The Generic Core-Wallet Protocol

In this section we describe the protocol $\pi_{\mathsf{CoreWallet}}$ (Figures 4.4 and 4.5), which realizes the core-wallet ideal functionality. The protocol interacts with the party $\mathcal{P}_o$, i.e., the wallet's owner, and maintains the, initially empty, lists $PK$ of payment keys and addresses and $SK$ of staking keys. Additionally, it uses a number of functions for different processes. parsePubAttrs returns the list of public attributes $[\mathsf{vks}, \mathsf{wrt}, \mathsf{aux}]$ given an address. The HKeyGen and RTagGen functions take effect during address generation and are analyzed next in Section 4.5.1. Finally, we assume the existence of a signature scheme $\Sigma$.

For this definition, we ease notation by dropping the generic attribute notation $\delta_i$ and instead using names representative of each attribute. Therefore, the staking and the payment information are the staking key $(\mathsf{vks}, \mathsf{sks})$ and the payment key $(\mathsf{vkp}, \mathsf{skp})$ respectively. The list of public attributes $d = [\mathsf{vks}, \mathsf{wrt}, \mathsf{aux}]$ comprises of the public staking key, the recovery tag, and the address's auxiliary information, which identifies its type. The semi-public attribute is the public payment key vkp, whereas the private attributes are the private keys sks and skp.

## 4.5 Security Analysis

The security of $\pi_{\mathsf{CoreWallet}}$ is given w.r.t. $\mathcal{F}^{M}_{\mathsf{CoreWallet}}$, the signature scheme's Existential Unforgeability under Adaptive Chosen Message Attacks (EUF-CMA) property, and a number of properties of our custom algorithms. Therefore, before presenting the analysis for $\mathcal{F}^{M}_{\mathsf{CoreWallet}}$ parameterized with the predicate $M_{\mathsf{SM}}$, we introduce the properties of the address and metadata generation algorithms.

### 4.5.1 Properties of the Generation Algorithms

Briefly, GenAddr is the address generation algorithm which, given an attribute list, returns an address. This algorithm is implemented in Section 4.6, although $\pi_{\mathsf{CoreWallet}}$ can

Protocol $\pi_{\mathsf{CoreWallet}}$

*Initialization:* Upon receiving the message $(\mathsf{Init}, \mathsf{sid})$ from $\mathcal{P}_o$, set $\mathsf{msk} \xleftarrow{\$} \{0,1\}^\kappa$ and return $(\mathsf{InitOk}, \mathsf{sid})$ to $\mathcal{P}_o$.
*Address Generation:* Upon receiving $(\mathsf{GenerateAddress}, \mathsf{sid}, \mathsf{aux})$ from $\mathcal{P}_o$, compute the index and child attributes as follows: i) pick an $i$ from the set $\mathbb{I}$; ii) compute the key pair $(\mathsf{vkp}_c, \mathsf{skp}_c) =$
$\mathsf{HKeyGen}(\langle \mathsf{msk}, \mathsf{payment}, i \rangle)$; iii) compute the tag $\mathsf{wrt} = \mathsf{RTagGen}(\mathsf{vkp}_c)$. Also:

- if $\mathsf{aux} = (\mathsf{base})$, compute $(\mathsf{vks}_c, \mathsf{sks}_c) = \mathsf{HKeyGen}(\langle \mathsf{msk}, \mathsf{staking}, i \rangle)$;
- else if $\mathsf{aux} = (\mathsf{pointer}, \mathsf{vks})$, find $(\mathsf{vks}_c, \mathsf{sks}_c) \in K : \mathsf{vks} = \mathsf{vks}_c$;
- else if $\mathsf{aux} = (\mathsf{exile})$, set $(\mathsf{vks}_c, \mathsf{sks}_c) = (\perp, \perp)$.

Then insert the list $l_\alpha = \langle \mathsf{vks}_c, \mathsf{wrt}, \mathsf{aux}, \mathsf{vkp}_c, \mathsf{skp}_c, \mathsf{sks}_c \rangle$ to $L$, create the address $\alpha = \mathsf{GenAddr}(\langle \mathsf{aux}, \mathsf{vks}_c, \mathsf{vkp}_c, \mathsf{wrt} \rangle)$, and insert the tuple $\langle \alpha, (\mathsf{vkp}_c, \mathsf{skp}_c) \rangle$ to $PK$. Then return $(\mathsf{Address}, \mathsf{sid}, \alpha)$ to $\mathcal{P}_o$. If $\mathsf{aux} = \mathsf{base}$ also insert $(\mathsf{vks}_c, \mathsf{sks}_c)$ to $SK$ and send the message $(\mathsf{StakingKey}, \mathsf{sid}, \mathsf{vks}_c)$ to $\mathcal{P}o$.
*Wallet Recovery:* Upon receiving $(\mathsf{RecoverWallet}, \mathsf{sid}, i_{max})$ from $\mathcal{P}_o$, $\forall i \in \mathbb{I} : i < i_{max}$ set $(\mathsf{vkp}_i, \mathsf{skp}_i) = \mathsf{HKeyGen}(\langle \mathsf{msk}, \mathsf{payment}, i \rangle)$ and return $(\mathsf{Tag}, \mathsf{sid}, \mathsf{RTagGen}(\mathsf{vkp}_i))$.
*Address Recovery:* Upon receiving from $\mathcal{P}_o$ the message $(\mathsf{RecoverAddr}, \mathsf{sid}, \alpha, i_{max})$, parse the address's attributes $(\mathsf{vks}, \mathsf{wrt}, \mathsf{aux}) = parsePubAttrs(\alpha)$. If exists $i \in \mathbb{I} : i < i_{max}$, where $(\mathsf{vkp}_i, \mathsf{skp}_i) = \mathsf{HKeyGen}(\langle \mathsf{msk}, \mathsf{payment}, i \rangle)$ and $\mathsf{RTagGen}(\mathsf{vkp}_i) = \mathsf{wrt}$, return $(\mathsf{RecoveredAddr}, \mathsf{sid}, \alpha)$.
*Issue Transaction:* Upon receiving from $\mathcal{P}_o$ the message $(\mathsf{Pay}, \mathsf{sid}, \Theta, \alpha_s, \alpha_r, m)$, find $\langle \alpha_s, (\mathsf{vkp}, \mathsf{skp}) \rangle \in PK$ and send the message $(\mathsf{Transaction}, \mathsf{sid}, \tau, \mathsf{Sign}(\mathsf{skp}, \tau))$ to $\mathcal{P}_o$, such that $\tau = (\Theta, \alpha_s, \alpha_r, m)$.

Figure 4.4: The PoS core-wallet protocol. (Part 1)

Protocol $\pi_{\mathsf{CoreWallet}}$

*Verify Transaction:* Upon receiving the message $(\mathsf{VerifyPay}, \mathsf{sid}, \tau, \sigma)$ from $\mathcal{P}_o$, where $\tau = (\Theta, \alpha_s, \alpha_r, m)$ for some metadata string $m$, find an entry $\langle \alpha_s, (\mathsf{vkp}, \mathsf{skp}) \rangle$ in $PK$ and return the message $(\mathsf{VerifiedPay}, \mathsf{sid}, \tau, \sigma, \mathsf{Verify}(\tau, \sigma, \mathsf{vkp}))$ to $\mathcal{P}_o$.

*Issue Staking:* Upon receiving $(\mathsf{Stake}, \mathsf{sid}, \tau_s)$ from $\mathcal{P}_o$ such that $\tau_s = (\mathsf{vks}, m)$ for metadata $m$, find an entry $(\mathsf{vks}, \mathsf{sks}) \in SK$ and return $(\mathsf{Staked}, \mathsf{sid}, \tau_s, \mathsf{Sign}(\mathsf{sks}, \tau_s))$.

*Verify Staking:* Upon receiving the message $(\mathsf{VerifyStake}, \mathsf{sid}, \tau_s, \sigma)$ from the party $\mathcal{P}_o$, where $\tau_s = (\mathsf{vks}, m)$ for metadata $m$, find $(\mathsf{vks}, \mathsf{sks}) \in SK$ and return the message $(\mathsf{VerifiedStake}, \mathsf{sid}, \tau_s, \sigma, \mathsf{Verify}(\tau_s, \sigma, \mathsf{sks}))$.

Figure 4.5: The PoS core-wallet protocol. (Part 2)

be parameterized with any address generation scheme which offers the necessary properties.

Specifically, address generation should be collision resistant, i.e., analogously to hash functions, it should be infeasible for an adversary to produce the same address for different attribute lists, cf. Definition 9. Also address generation should be attribute non-malleable, i.e., it should be infeasible for an adversary to generate valid addresses for a payment key without access to the entire attribute list (including the private attributes), cf. Definition 10. In addition, we assume that $\mathsf{HKeyGen}(\cdot)$ is hierarchical, i.e., the distribution of produced keys is indistinguishable from that of KeyGen, cf. Definition 11, and $\mathsf{RTagGen}(\cdot)$ is collision resistant, thus ensuring that the recovery tags are both unique (with overwhelming probability) and deterministically computable.

**Address and Attribute Generation Properties.** In order to prove the security of our protocol, the address generation algorithm $\mathsf{GenAddr} : \Delta_1 \times \ldots \times \Delta_g \to \mathbb{A}$ should offer the following properties.

**Definition 9** (Address collision resistance)**.** *Analogously to hash functions (cf. Definition 1),* GenAddr *is collision resistant when it is infeasible to produce two different attribute lists* $l_i = (\delta_1^i, \ldots, \delta_g^i)$ *for* $i \in \{1, 2\}$*, i.e., they differ in at least one attribute like* $\exists j \in [1, g]$*:* $\delta_j^1 \neq \delta_j^2$*, such that* $\mathsf{GenAddr}(l_1) = \mathsf{GenAddr}(l_2)$*, after running* $\mathsf{GenAddr}(\cdot)$ *a polynomial number of times.*

Similarly, we assume the existence of an algorithm GenMeta : $\Delta_1 \times ... \times \Delta_g \rightarrow \{0,1\}^{p(\kappa)}$ for the generation of the metadata $m$ associated with the address. Before describing extra properties for GenAddr, we introduce the following property for the GenMeta function.

**Metadata extraction resistance.** For a challenge address $\alpha$ with attribute list $l$, it is intractable for the adversary that is given $\alpha$ and the public attributes to generate the metadata information $m$, such that $\mathsf{GenMeta}(l) = m$, even with polynomial number of address generation and metadata queries, upon arbitrary choices of public attributes $(\delta'_i, ..., \delta'_g)$. We say that GenMeta algorithm has metadata extraction resistance with respect to private attributes $(\delta_1, ..., \delta_{i-1})$ from an attribute list $l = (\delta_1, ..., \delta_g)$, and address generation algorithm GenAddr, when for every attribute list $l \in \Delta_1 \times ... \times \Delta_g$, and a fixed index $i$, such that the private list of attributes is $d = (\delta_1, ..., \delta_{i-1})$, it holds that:

$$\mathsf{Pr}\left[\begin{array}{c} l = (\delta_1, ..., \delta_g), \\ \alpha \leftarrow \mathsf{GenAddr}(l), \\ m' \leftarrow \mathcal{A}^{\mathsf{GenAddr}^d(\cdot)}(\delta_i, ..., \delta_g) \end{array} : \mathsf{GenMeta}(l) = m'\right] \leq \mathsf{negl}(\kappa)$$

where the probabilities are computed on the used randomness, the values of $l$ and every PPT adversary $\mathcal{A}$.

**Definition 10** (Non-malleable attribute address generation)**.** *Let $\mathcal{L}$ be a distribution of attribute lists, such that* $\mathsf{dom}(\mathcal{L}) = \Delta_1 \times ... \times \Delta_g$*, and that $\delta_1, ..., \delta_i$, i.e., the first attributes of the attribute list $l \leftarrow \mathsf{dom}(\mathcal{L})$, relate to a property over which we define non-malleability. Even with access to the metadata of the address, i.e., the semi-public attributes, it is infeasible for the adversary to generate valid addresses, i.e., acceptable addresses with the same verification payment key, from $\alpha$, without accessing the attribute list, including the private attributes. Concretely, given* Addrs*, i.e., the list of addresses queried by $\mathcal{A}$ to the oracle* $\mathsf{GenAddr}^d(\cdot)$*, it holds that:*

$$\mathsf{Pr}\left[\begin{array}{c} l = (\delta_1, ..., \delta_g), \\ \alpha \leftarrow \mathsf{GenAddr}(l), \\ (\alpha', \delta'_i, ..., \delta'_g) \leftarrow \mathcal{A}^{\mathsf{GenAddr}^d(\cdot), \mathsf{GenMeta}^d(\cdot)}(\delta_i, ..., \delta_g) \end{array} : \begin{array}{c} (\mathsf{GenAddr}^d(\delta'_i, ..., \delta'_g) = \alpha') \wedge \\ (\alpha' \neq \alpha) \wedge \\ (\alpha' \notin \mathsf{Addrs}) \end{array}\right] \leq \mathsf{negl}(\kappa)$$

*for probabilities computed over the randomness in* GenAddr *algorithm, every PPT adversary $\mathcal{A}$ and the values of $l$.*

**Hierarchical Key Generation Properties.** We now define the hierarchical property of the key generation function HKeyGen$(\cdot)$, which is used by the core wallet protocol $\pi_{\mathsf{CoreWallet}}$. Intuitively, this allows us to use HKeyGen in order to securely and deterministically produce keys for indexes, instead of using the KeyGen function of $\Sigma$.

**Definition 11** (Hierarchical Key Generation)**.** *Assume a signature scheme* $\Sigma$*. A key generation function* HKeyGen$(\cdot)$ *is hierarchical for* $\Sigma$ *if, for all parameters* $i$*, the distribution of keys produced by* HKeyGen$(i)$ *is computationally indistinguishable from the distribution of keys produced by* KeyGen*.*

### 4.5.2 Security in the Sink Malleable Setting

We now describe the core theorem of this chapter. Here, the ideal functionality, parameterized with the sink malleability predicate $M_{\mathsf{SM}}$, is realized by the protocol that employs the sink malleable address generation function. It also uses tag and hierarchical key construction functions which present the above necessary properties.

**Theorem 2.** *Let the generic protocol* $\pi_{\mathsf{CoreWallet}}$ *be parameterized by a signature scheme* $\Sigma$ *and the* RTagGen*,* HKeyGen*, and* GenAddr *functions. Then* $\pi_{\mathsf{CoreWallet}}$ *securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{SM}}}$ *if and only if* $\Sigma$ *is EUF-CMA,* GenAddr *is collision resistant and attribute non-malleable (cf. Definitions 9 and 10),* RTagGen *is collision resistant (cf. Definition 1), and* HKeyGen *is hierarchical for* $\Sigma$ *(cf. Definition 11).*

*Proof.* The proof is constructed in the UC Framework, therefore it is a simulation based proof. As such, we will show that the environment $\mathcal{Z}$ cannot efficiently distinguish between two executions, the ideal and the real. Here, the simulator $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{SM}}}$ in the ideal execution, whereas $\mathcal{A}$ interacts with $\pi_{\mathsf{CoreWallet}}$ in the real execution. We divide the proof in the "if" and "only if" parts. First, the "if" part shows that if $\pi_{\mathsf{CoreWallet}}$ does securely realize the ideal functionality $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{SM}}}$, when instantiated with a EUF-CMA signature scheme $\Sigma$, a collision resistant and non-malleable address generation scheme GenAddr, and suitable RTagGen, HKeyGen functions at least one of the conditions is violated. The "only if" part shows that, if either of the functions' properties does not hold, e.g., if $\Sigma$ is not EUF-CMA or GenAddr is either not collision resistant or non-malleable, then $\pi_{\mathsf{CoreWallet}}$ does not securely realize the functionality $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{SM}}}$, i.e., the environment is able to distinguish between the two executions.

Let us now provide the construction for the simulator $\mathcal{S}$, which will be useful in the "if" part of the proof.

*The simulator $\mathcal{S}$.* The simulator $\mathcal{S}$ runs internally a copy of the adversary $\mathcal{A}$, and keeps a table TABLE of tuples $(\cdot,\cdot,\cdot,\cdot)$ of respectively addresses, attributes, and staking key pairs. Also it performs as follows:

- Any inputs received from the environment $\mathcal{Z}$, forward them to the internal copy of $\mathcal{A}$. Moreover, forward any output from $\mathcal{A}$ to $\mathcal{Z}$;
- *Initialization:* Upon receiving $(\mathsf{Initialise}, \mathsf{sid})$ from the functionality $\mathcal{F}_{\mathsf{CoreWallet}}$, compute a dummy master key $\mathsf{msk} \xleftarrow{\$} \{0,1\}^{\kappa}$ and return $(\mathsf{InitialiseOk}, \mathsf{sid})$;
- *Address Generation:* Upon receiving the message $(\mathsf{GenerateAddress}, \mathsf{sid}, \mathsf{aux})$ from $\mathcal{F}_{\mathsf{CoreWallet}}$, do similarly to protocol $\pi_{\mathsf{CoreWallet}}$, that is:
  - set $i \leftarrow \mathbb{I}$,
  - set the key pair $(\mathsf{vkp}_c, \mathsf{skp}_c) = \mathsf{HKeyGen}(\langle \mathsf{msk}, \mathsf{payment}, i\rangle)$,
  - set the tag $\mathsf{wrt} = \mathsf{RTagGen}(\langle \mathsf{msk}, i\rangle)$,

  and do the following:

  - if $\mathsf{aux} = (\mathsf{base})$ compute $(\mathsf{vks}_c, \mathsf{sks}_c) = \mathsf{HKeyGen}(\langle \mathsf{msk}, \mathsf{staking}, i\rangle)$ and set $\beta = \mathsf{vks}_c$;
  - if $\mathsf{aux} = (\mathsf{pointer}, \mathsf{vks})$, set $\beta = \mathsf{vks}$;
  - if $\mathsf{aux} = (\mathsf{exile})$, set $\beta = \perp$.

  Then compute $\alpha = \mathsf{GenAddr}(\langle \mathsf{aux}, \beta, \mathsf{vkp}_c, \mathsf{wrt}\rangle)$ and set its attribute list $l_\alpha = \langle \mathsf{vks}_c, \mathsf{wrt}, \mathsf{aux}, \mathsf{vkp}_c, \mathsf{skp}_c, \mathsf{sks}_c\rangle$. Then record the tuple $(\alpha, l_\alpha, \beta, \mathsf{skp}_c)$ to TABLE. Finally, hand to $\mathcal{F}_{\mathsf{CoreWallet}}$ the message $(\mathsf{Address}, \mathsf{sid}, \alpha, l_\alpha)$;
- *Issue Transaction:* Upon receiving $(\mathsf{Pay}, \mathsf{sid}, \Theta, \alpha_s, \alpha_r, m)$ find a record $l_\alpha \in$ TABLE that contains the sender's address $\alpha_s$ as the first item. Then generate the signature $\sigma$ for the transaction $\tau$, such that $\tau = (\Theta, \alpha_s, \alpha_r, m)$, using Sign and the payment key of $\alpha_s$, and hand the message $(\mathsf{Transaction}, \mathsf{sid}, \tau, \sigma)$ to the functionality $\mathcal{F}_{\mathsf{CoreWallet}}$. Note that with the attribute list $l_\alpha$, $\mathcal{S}$ can properly generate $\sigma$. Moreover such record is expected to be in TABLE, since the functionality allows the issuing of transactions by properly generated addresses by checking on the list $L_P$ before sending to $\mathcal{S}$;

- *Verify Transaction:* Upon receiving the message $(\text{VerifyPayment}, \text{sid}, \tau, \sigma)$ from $\mathcal{F}_{\text{CoreWallet}}$, find the recorded verification key $\text{vkp}_c$ for the sender's address $\alpha_s$ in $\tau = (\Theta, \alpha_s, \alpha_r, m)$ by looking up $l_\alpha$ for $\alpha_s$ in TABLE, and use Verify to retrieve the verification bit $\phi$. Then return $(\text{VerifiedPayment}, \text{sid}, \tau, \sigma, \phi)$ to $\mathcal{F}_{\text{CoreWallet}}$;

- *Issue Staking:* Similarly to issuing a payment, upon receiving $(\text{Stake}, \text{sid}, \tau_s)$, such that $\tau_s = (\text{vkp}, m)$, find the correspondent staking key skp and use Sign to generate the signature $\sigma$, then hand $(\text{Staked}, \text{sid}, \tau_s, \sigma)$ to $\mathcal{F}_{\text{CoreWallet}}$;

- *Verify Staking:* As before, upon receiving the message $(\text{VerifyStaking}, \text{sid}, \tau_s, \sigma)$, find the staking key that pertains to $\tau_s$, use Verify to retrieve the verification bit $\phi$, and send the message $(\text{VerifiedPayment}, \text{sid}, \tau_s, \sigma, \phi)$ to $\mathcal{F}_{\text{CoreWallet}}$. Similarly to *Issue Transaction* interface, note that $\mathcal{S}$ knows $\text{skp}_c$ via TABLE;

- *Party Corruption:* Whenever the adversary $\mathcal{A}$ corrupts a party $P$, $\mathcal{S}$ corrupts it in the ideal process and hands to $\mathcal{A}$ the corresponding entries in TABLE.

*The "if" part.* Assume, for the sake of the argument, that the environment $\mathcal{Z}$ can distinguish between the ideal and the real execution with non-negligible probability for any simulator construction, including the earlier $\mathcal{S}$ construction. In that case, it suffices to show that, if $\pi_{\text{CoreWallet}}$ does not securely realize $\mathcal{F}_{\text{CoreWallet}}$, and given the $\mathcal{S}$ construction, then either of the following holds when a "bad" event $E$ occurs: the signature scheme is not EUF-CMA, GenAddr is not collision resistant or attribute non-malleable, the function RTagGen is not collision resistant, or the hierarchical property of HKeyGen does not hold.

*Unforgeability:* We assume that *collision* resistance and non-malleability properties of the algorithm GenAddr hold, likewise the collision resistance and hierarchical properties for RTagGen and HKeyGen respectively, along with the signature scheme properties *completeness* and *non-repudiation*, but unforgeability does not hold (otherwise the theorem completes). Note that, by hypothesis, $\mathcal{Z}$ distinguishes between the two worlds for any construction of $\mathcal{S}$, including the earlier $\mathcal{S}$ construction.

Now we construct a forger $G$ for the unforgeability game per the EUF-CMA definition, which initially receives $(\text{vkp}, \text{skp})$ as a challenge (we focus on the payment keys and interfaces, but we note that the case for staking is analogous). $G$ simulates the earlier described $\mathcal{S}$ in the interaction with $\mathcal{Z}$. It then issues signature queries to its game, when requested by its simulation of $\mathcal{S}$ and $\mathcal{F}_{\text{CoreWallet}}$ for signatures on vkp.

In addition, when receiving verification messages of the form $(\mathsf{VerifyPayment}, \mathsf{sid}, \tau, \sigma)$ for other addresses (possibly from other verification keys), $G$ uses its internal simulation, specifically the *Transaction Verification* interface, i.e., its internally generated keys, to properly simulate the execution to $\mathcal{Z}$. In particular it checks if $(\tau, \sigma)$ has been queried in the security game. If it is not listed as queried and $\mathsf{Verify}(\tau, \sigma, \mathsf{vkp}) = 1$, then it outputs $(\tau, \sigma)$ and wins the game. Otherwise, it continues with the simulation.

Given that the environment $\mathcal{Z}$ distinguishes between the two executions by hypothesis and all other properties of the functions hold, we are guaranteed that if $\pi_{\mathsf{CoreWallet}}$ does not securely realize $\mathcal{F}_{\mathsf{CoreWallet}}$, then the unforgeability property does not hold.

Note that the earlier unforgeability reasoning is valid for $\Sigma$ with key generation relying on KeyGen, however $\pi_{\mathsf{CoreWallet}}$ relies on HKeyGen. Therefore, consider the following argument.

HKeyGen *is hierarchical for $\Sigma$ and every index $i$ is used only once:* Assume the two following protocols: i) a protocol which is similar to $\pi_{\mathsf{CoreWallet}}$, except the key generation function KeyGen of $\Sigma$ is used instead of HKeyGen, and ii) the protocol $\pi_{\mathsf{CoreWallet}}$. Now, it is evident that the first protocol securely realizes the ideal functionality (since all other properties needed as per the Theorem hold). Therefore, the execution of the first protocol is indistinguishable from the execution of the ideal functionality, as proved in the earlier reasoning.

Next, consider a PPT algorithm $D$ who tries to distinguish between the executions of the two protocols above. Specifically, assume that there exists a "special" index $i$, for which the usage of HKeyGen in the signature scheme is insecure, i.e., a forgery can be computed by the adversary. For the sake of argument, consider the probability that the scheme breaks for this index $i$ by $p$. It is clear that, for this index $i$, $D$ is successful, by observing the violation of the properties of the signature scheme with HKeyGen. Note also that the number of produced keys, i.e., the number of used indexes in both executions, is bounded by a polynomial $P(\kappa)$. Therefore, the overall probability that $D$ is successful is equal to $\frac{p}{P(\kappa)}$.

However, by definition, HKeyGen is hierarchical for $\Sigma$. Thus, the executions of the two protocols are indistinguishable and, as a result, the probability that $D$ is successful, i.e., $\frac{p}{P(\kappa)}$, is negligible. Consequently, the probability $p$, i.e., that the signature scheme which uses HKeyGen breaks, is also negligible. Therefore, the execution of $\pi_{\mathsf{CoreWallet}}$ is indistinguishable from the execution of the ideal functionality as well, thereby $\pi_{\mathsf{CoreWallet}}$ securely realizes the ideal functionality.

*The "only if" part.* Here we show that, if a single property does not hold, then the

environment $\mathcal{Z}$ can distinguish between the real and ideal executions with non-negligible probability. In other words, there is no simulator construction that prevents $\mathcal{Z}$ from distinguishing both executions.

*Success probability of $\mathcal{Z}$ under weakened assumptions.* We now assume that some property of the functions used by the protocol is broken. We will then show that $\pi_{\mathsf{CoreWallet}}$ does not securely realize $\mathcal{F}_{\mathsf{CoreWallet}}$. Specifically, we can create an environment $\mathcal{Z}$ and an adversary $\mathcal{A}$ such that, for *any* simulator $\mathcal{S}$, $\mathcal{Z}$ distinguishes between the real execution of $\mathcal{A}$ with $\pi_{\mathsf{CoreWallet}}$ and the ideal execution of $\mathcal{S}$ with $\mathcal{F}_{\mathsf{CoreWallet}}$.

Initially, the environment sends $(\mathsf{Initialise}, \mathsf{sid})$ for some party $\mathcal{P}$. For each property required by the theorem, we show that the environment can distinguish between the two executions as follows:

- *Completeness:* We assume that $\Sigma$ is not complete. $\mathcal{Z}$ initializes the wallet for a second party $\mathcal{P}'$ and sends two messages $(\mathsf{GenerateAddress}, \mathsf{sid}, \mathsf{aux})$, for some arbitrary auxiliary information aux, and obtains two addresses $\alpha_s$ and $\alpha_r$ for the parties $\mathcal{P}$ and $\mathcal{P}'$ respectively. Next, it creates a transaction object $\tau = (\Theta, \alpha_s, \alpha_r, m)$, for arbitrary values of assets $\Theta$ and metadata $m$, and obtains a signature $\sigma$ for the transaction $\tau$ by sending the message $(\mathsf{Pay}, \mathsf{sid}, \tau)$. Finally, it calls the verification interface by sending the message $(\mathsf{VerifyPayment}, \mathsf{sid}, \tau, \sigma)$. In the ideal execution the output is always $(\mathsf{VerifiedPayment}, \mathsf{sid}, \tau, \sigma, 1)$, whereas in the real execution the probability that the output is $(\mathsf{VerifiedPayment}, \mathsf{sid}, \tau, \sigma, 0)$ is non-negligible. The environment could also succeed in distinguishing the executions by accessing the *Staking* and *Staking Verification* interfaces, issuing staking acts and checking the verification bit similarly as with payment transactions.

- *Non-repudiation:* We assume that $\Sigma$ does not offer non-repudiation. The environment now acts like in the case of *completeness*, obtaining a signed transaction $(\tau, \sigma)$. However, it now calls the verification interface twice. In the ideal execution, the verification bit of the response will both times be equal to $1$, whereas in the real execution the probability that the verification bit is $0$ is non-negligible. Again the environment could access the staking issuing and verification interfaces similarly.

- *Unforgeability:* We assume that $\Sigma$ is forgeable, so there exists a forger $G$ for $\Sigma$. The environment now runs an internal copy of $G$. When $G$ wishes to obtain a signature from its oracle for some transaction $\tau$, $\mathcal{Z}$ accesses the *Issue Transaction* interface by sending the message $(\mathsf{Pay}, \mathsf{sid}, \tau)$ and obtains a signature, which

it forwards to $G$. When $G$ outputs a signed transaction $(\tau, \sigma)$, $\mathcal{Z}$ proceeds as follows. If $\tau$ has been previously signed, i.e., if $\sigma$ has been creating by accessing the *Issue Transaction* before, then it halts. Otherwise, it accesses the verification interface by sending the message $(\mathsf{VerifyPayment}, \mathsf{sid}, \tau, \sigma)$. Now, in the ideal execution the verification bit in the response from the verification interface is always $0$, whereas in the real world it is $1$ with non-negligible probability.

- *Collision resistance:* We assume that GenAddr is not collision resistant. The environment obtains two addresses by calling the address generation interface twice, i.e., sending two messages $(\mathsf{GenerateAddress}, \mathsf{sid}, \mathsf{aux})$ for the same auxiliary information aux. In the ideal execution the attribute lists in the address responses will always be different, whereas in the real execution the probability that two equal addresses for different attribute lists are returned is non-negligible.

- *Attribute non-malleable:* We assume that GenAddr is not attribute non-malleable. Then the environment $\mathcal{Z}$, which may retrieve correctly generated addresses by accessing the *Address Generation* interface, can generate a forged address $\alpha^*$. Assume, without loss of generality, that $\alpha^*$ has been the receiving address for some past transaction, therefore $\alpha^*$ owns some assets. Assume also that $\mathcal{Z}$ issues a transaction from $\alpha^*$ to a legitimate address $\alpha_r$, i.e., created via the address generation interface of $\mathcal{F}_{\mathsf{CoreWallet}}$. On submitting $(\mathsf{VerifyPayment}, \mathsf{sid}, \Theta, \alpha^*, \alpha_r, m, \sigma)$, for some assets $\Theta$ and metadata $m$, the environment $\mathcal{Z}$ will receive a message $(\mathsf{VerifiedPayment}, \mathsf{sid}, \Theta, \alpha^*, \alpha_r, m, \sigma, 0)$, since the check of the predicate $M_{\mathsf{SM}}$ within $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{SM}}}$ outputs $0$. On the other hand, in the real world interaction with $\pi_{\mathsf{CoreWallet}}$, $\mathcal{Z}$ receives $(\mathsf{VerifiedPayment}, \mathsf{sid}, \Theta, \alpha^*, \alpha_r, m, \sigma, 1)$. Therefore, $\mathcal{Z}$ is able to distinguish between the executions.

- RTagGen *collision resistance and every index $i$ is used only once:* Let us now assume that either RTagGen is not collision resistant or that an index $i$ is used more than once. Then $\mathcal{Z}$ can generate several address requests $(\mathsf{GenerateAdddress}, \mathsf{sid}, \mathsf{aux})$ and observe the generated tags wrt within each address $\alpha$. If RTagGen is not collision resistant, then $\mathcal{Z}$ will observe a bias in the distribution of the output of RTagGen in the real world, i.e., it will observe the same recovery tag for two different addresses.

- HKeyGen *hierarchical property and every index $i$ is used only once:* Assume now that HKeyGen is not hierarchical for $\Sigma$. Then, following the reasoning above,

the execution of the protocol which uses KeyGen is indistinguishable from the ideal execution. However, by assumption the execution of $\pi_{\mathsf{CoreWallet}}$ is not indistinguishable from the execution of that protocol anymore. Therefore, it is not indistinguishable from the execution of the ideal functionality as well.

Note that, in all cases, there is no mention of the simulator $\mathcal{S}$, therefore the reasoning applies for any construction of $\mathcal{S}$.

In conclusion, we have shown that if either of the properties is broken, then the environment can distinguish between the two executions, thus a protocol that uses a scheme that does not provide one of the properties does not securely realize the ideal functionality $\mathcal{F}_{\mathsf{CoreWallet}}$. □

### 4.5.3 Security in the Fully Malleable Setting

We also realize the functionality parameterized with the fully malleable predicate $M_{\mathsf{FM}}$; the predicate description is given in Algorithm 2.

**Theorem 3.** *Let the generic protocol $\pi_{\mathsf{CoreWallet}}$ be parameterized by a signature scheme $\Sigma$ and the* RTagGen*,* HKeyGen*, and* GenAddr *functions. Then $\pi_{\mathsf{CoreWallet}}$ securely realizes the ideal functionality $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{FM}}}$ if and only if $\Sigma$ is EUF-CMA,* RTagGen *is a collision resistant (cf. Definition 1),* HKeyGen *is hierarchical for $\Sigma$ as (cf. Definition 11), and* GenAddr *is collision resistant (cf. Definition 9).*

*Proof.* The proof follows similarly to the proof of Theorem 2, while not considering the non-malleability property of the address generation scheme. □

### 4.5.4 Attacking the Malleable Protocol in the Non-Malleable Setting

In this section, before presenting the attack, we describe an additional property of GenAddr. Next we show that with the extra property, in a more restricted setting which is captured by the non-malleable predicate $M_{\mathsf{NM}}$ of Algorithm 5, the protocol cannot be proven secure.

**Metadata extraction for a posteriori malleability.** Given $\alpha$, it is infeasible for the adversary to generate valid addresses, i.e., for the same verification payment key, without access to at least the metadata (for completeness, or the attribute list of the original address). More concretely:

$$\Pr\left[\begin{array}{cc} l=(\delta_1,\ldots,\delta_g), & (\mathsf{GenAddr}^d(\delta'_i,\ldots,\delta'_g)=\alpha')\wedge \\ \alpha\leftarrow\mathsf{GenAddr}(l), & : \quad (\alpha'\neq\alpha)\wedge \\ (\alpha',\delta'_i,\ldots,\delta'_g)\leftarrow\mathcal{A}^{\mathsf{GenAddr}^d(\cdot)}(\delta_i,\ldots,\delta_g) & (\alpha'\notin\mathsf{Addrs}) \end{array}\right] \leq \mathsf{negl}(\kappa)$$

where the probabilities are computed on the randomness used in GenAddr algorithm, every PPT adversary $\mathcal{A}$ and the values of $l$. Furthermore, Addrs is the list of all the addresses received from the $\mathsf{GenAddr}(\cdot)$ oracle.

**Remark.** *The a posteriori malleability property implicitly requires that* GenMeta *is metadata extraction resistant, otherwise the metadata are easily accessible by the adversary.*

We now describe the attack on the protocol $\pi_{\mathsf{CoreWallet}}$ instantiated with an a posteriori malleable GenAddr algorithm.

**Theorem 4.** *Let the protocol $\pi_{\mathsf{CoreWallet}}$ be instantiated with a collision resistant and a posteriori malleable* GenAddr*, a metadata extraction resistant* GenMeta *function as defined in the a posteriori malleability setting of Section 4.5.1, an EUF-CMA signature scheme $\Sigma$, a collision resistant* RTagGen*, and a hierarchical* HKeyGen *for $\Sigma$ as per Definition 11. Then, $\pi_{\mathsf{CoreWallet}}$ does not securely realize the ideal functionality $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{NM}}}$.*

*Proof.* We construct an environment $\mathcal{Z}$ which distinguishes efficiently between the ideal execution of $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{NM}}}$ and $\mathcal{S}$, and the real one given by $\pi_{\mathsf{CoreWallet}}$ and $\mathcal{A}$. The attack exploits the malleable feature of the address construction regarding the exposure of the metadata while spending the funds.

The environment issues an address via party $\mathcal{P}_1$, say $\alpha_1$, now assume without loss of generality that $\alpha_1$ receives some funds during the execution. Now $\mathcal{Z}$ issues an address generation request for party $\mathcal{P}_2$, therefore it would receive $\alpha_2$. The next step is to generate a transaction from $\alpha_1$ to $\alpha_2$, which $\mathcal{Z}$ can do by issuing message $(\mathsf{Pay},\mathsf{sid},\Theta,\alpha_1,\alpha_2)$ from $\mathcal{P}_1$. $\mathcal{Z}$ receives $(\mathsf{Transaction},\mathsf{sid},\tau,\mathsf{Sign}(\mathsf{skp},\tau))$ such that $\tau=(\Theta,\alpha_1,\alpha_2,m)$ for metadata $m$.

At this point, the property metadata extraction resistance of the GenMeta nor GenAddr help any longer, since $m$ is exposed to $\mathcal{Z}$. It is fair to assume that $\mathcal{Z}$ uses $m$ to generate locally, i.e., without using the *Address Generation* interface, the address $\alpha_{local}$.

Next, for the sake of argument, assume the existence of a simulator $\mathcal{S}$ which perfectly simulates the ideal execution, therefore we can also assume that eventually $\alpha_{local}$

has assets $\Theta'$. At this point, $\mathcal{Z}$ submits the message $(\mathsf{Pay}, \mathsf{sid}, \Theta', \alpha_{local}, \alpha)$ for any party $\mathcal{P}$ and some (correctly generated) address $\alpha$, then the environment $\mathcal{Z}$ receives $(\mathsf{Transaction}, \mathsf{sid}, \tau, \sigma)$ such that $\tau = (\Theta', \alpha_{local}, \alpha, m')$ for metadata $m'$.

Finally, in order to distinguish the executions, $\mathcal{Z}$ submits $(\mathsf{VerifyPayment}, \mathsf{sid}, \tau, \sigma)$ to any party, for which it receives $(\mathsf{VerifiedPayment}, \mathsf{sid}, \tau, \sigma, b)$. Note that $b = 1$ if it is a real execution by $\pi_{\mathsf{CoreWallet}}$ and $\mathcal{A}$, because $b = \mathsf{Verify}(\tau, \sigma, \mathsf{vkp})$. On the other hand, due to the fully non-malleable predicate $M_{\mathsf{NM}}$, which would output $0$, $b = 0$.

$\mathcal{Z}$ efficiently distinguishes between the executions, thereby $\pi_{\mathsf{CoreWallet}}$, in the a posteriori malleability setting, does not securely realizes the ideal functionality $\mathcal{F}_{\mathsf{CoreWallet}}^{M_{\mathsf{NM}}}$. □

## 4.6 PoS Addresses: Construction and Recovery

The final step in fully realizing the core wallet is to implement the functions used by the protocol $\pi_{\mathsf{CoreWallet}}$, more importantly the address generation function. In the following paragraphs, we outline the three types of addresses in our framework, i.e., *base*, *pointer*, and *exile*. We concretely describe an address's attributes and how an index is used to generate a "child" address. Then, we present multiple address schemes, each offering different levels of protection against malleability attacks, resulting in different address lengths, but all easily implementable with standard cryptographic primitives.

### 4.6.1 Address Types and their Attributes

As discussed above, at least two attributes are required per address, i.e., the staking $(\mathsf{vks}, \mathsf{sks})$ and the payment $(\mathsf{vkp}, \mathsf{skp})$ key pairs. As shown in Section 4.3, the signing keys sks and skp are *private*, while the verification keys vkp and vks are *semi-public* and *public* attributes respectively. We remind that $(\mathsf{vkp}, \mathsf{skp})$ is used in proving ownership of the assets and issuing payments, whereas $(\mathsf{vks}, \mathsf{sks})$ is used to perform staking actions on behalf of the assets.

The first step in computing a "child" address and its attributes is the choice of an index $i$ from the set $\mathbb{I}$. As with hardware wallets (Chapter 3), an index is an identifier that is used to generate a "child" key. We define a list of domains $[\mathbb{I}_1, \mathbb{I}_2, \ldots,]$, where each $\mathbb{I}_i$ has a finite, relatively small, cardinality. During address generation, the wallet initially picks indexes from $\mathbb{I}_1$. After all indexes in $\mathbb{I}_1$ have been used, it uses $\mathbb{I}_2$ and so on. It is required that at least one address for an index in $\mathbb{I}_j$ is published on the ledger, i.e., is on the sending or receiving end of a transaction, before indexes from $\mathbb{I}_{j+1}$ are

used. During recovery, the wallet sets $j = 1$ and generates all indexes in $\mathbb{I}_j$. It then constructs the recovery tag for each index and compares it with each address in the blockchain. If at least one index has been used in a published address, then the wallet sets $j = j + 1$ and repeats for $\mathbb{I}_{j+1}$. When, for some $j$ no index corresponds to any published address, recovery is complete.

We now discuss the complexity of the recovery procedure. Given that the cardinality of $\mathbb{I}$ is small, the probability that the same index is chosen twice, even for a random choice, is not negligible. Therefore, two devices that maintain the same wallet core would need to share the state of the indexes that have been used by each. The number of indexes and addresses that the wallet can generate is not restricted, since the set of domains is infinite. In terms of complexity, this naive scheme is linear to the number of addresses in the ledger. We can improve this with an index of published addresses, based on their recovery tag. Using such index, the recovery complexity becomes linear to the cardinality of $\bigcup_j \mathbb{I}_j$, i.e., the number of addresses that the wallet owns and has published. Since the recovery tags are public, such index can be constructed and circulated on the network by everybody.

A hierarchical key, which is derived from the wallet's master key msk and is linked to a child address, is created by HKeyGen. This function takes the master key, a label $\texttt{lbl} \in \{\text{payment}, \text{staking}\}$ and an index $i$ and passes them to a pseudorandom function, which outputs a pseudorandom value passes it to a pseudorandom number generator that outputs $p(\kappa)$ bits $\rho$, for some suitable polynomial $p$. These bits are then passed as random coins to the key generation function $\text{KeyGen}(1^\kappa; \rho)$. Therefore, to generate a child payment key, the protocol runs $\text{HKeyGen}(\langle \text{msk}, \text{payment}, i \rangle)$, while similarly $\texttt{lbl} = \text{staking}$ is used to issue a new staking key.

The wallet produces three types of addresses, differentiated by the staking object $\beta$. To produce a *base* address, the wallet computes a staking key vks and sets $\beta$ as the hash of it. For a *pointer* address, the address's staking key is set indirectly. Specifically, the staking object $\beta$ is a delegation pointer $ptr$. The pointer is a string that identifies a published certificate, i.e., the representation of a staking action on the ledger, which is described in detail in Section 4.7. If $ptr$ points to a valid delegation certificate $\tau_{del}$, then the address's staking key is the delegate's key in $\tau_{del}$, whereas, if $ptr$ points to a registration certificate, then the delegate's key is the key of the stake pool defined in that certificate. Finally, for an *exile* address, the staking object is a fixed value $\epsilon = \perp$. Since $\epsilon$ does not identify a staking key, the owner of an exile address cannot perform staking actions or delegate the address's staking rights, so all assets owned by such addresses

are effectively removed from the PoS protocol.

Each address also contains the recovery tag wrt, i.e., a public parameter which allows the identification of addresses. The tag is created by the function RTagGen and links the address to the attribute list, without revealing the semi-public attributes. Recovery is a process that relies only on the master key of the wallet, so the wallet should be able to recover an address by *only knowing its payment key*. In the simplest setting, during recovery the wallet computes the keys for its indexes and hashes them to compute the recovery tags. Therefore, RTagGen is the hash function H and, by definition, is collision resistant as required.

**Searchable recovery.** This design builds on the premise of searchable encryption [BBO07]. Specifically, access to the wallet's master key enables searching all addresses in the ledger and recognize the ones that belong to the wallet. Assuming an instance of a semantically secure symmetric encryption scheme $\langle \mathsf{Enc}, \mathsf{Dec} \rangle$, the hierarchical tag generation function SearchableTagGen computes the output of Enc, using msk as the encryption key and the index $i$ as the plaintext:

$$\mathsf{sht} = \mathsf{Enc}(\mathsf{msk}, i) \tag{4.1}$$

During recovery, the wallet parses all addresses in the blockchain and decrypts the tag $\mathsf{sht}'$ in each as $i' = \mathsf{Dec}(\mathsf{msk}, \mathsf{sht}')$. If the output is a well-formed index $i' \in \mathcal{I}$, the address belongs in the wallet and its hierarchical index is $i'$. Since the output of the encryption function Enc is by definition random in the domain of Enc, the recovery tag generation is a PRF as needed.

**Malleability verification tags.** These tags are used to verify whether a part of the address has been altered during transit. Specifically, the motivation is for the address owner to identify whether a malleability attack has occurred, even in case the transaction is accepted by the system. The tag is computed as follows:

$$\mathsf{mht} = \mathsf{H}(\mathsf{msk}||i||\beta) \tag{4.2}$$

where $\beta$ is the staking attribute, for which the wallet needs to identify possible malleability attacks. Clearly the tag generation is again a pseudorandom function. Also, without the master key msk, an attacker cannot forge such tag. However, only the wallet owner can verify that such tag is properly constructed for a given address, i.e., it is not sufficient to construct non-malleable address schemes.

### 4.6.2 Malleable Addresses

As discussed in Section 4.5.1, The two basic properties of an address generation function are *collision resistance* and *non-malleability*. On the one hand, collision resistance is a feature that has been extensively investigated in the literature. On the other hand, address non-malleability, in the manner presented in this chapter, is a novel notion. Therefore, let us first present a malleable scheme and the problems that stem from it.

Suppose two users $\mathcal{P}_A$ and $\mathcal{P}_B$. $\mathcal{P}_B$ wishes to receive some assets from $\mathcal{P}_A$, so she generates an address $\alpha$ and gives it to $\mathcal{P}_A$; the payment key of $\alpha$ is vkp and the staking object is $\beta$. If the address generation function is malleable, $\mathcal{P}_A$ can create an address $\alpha'$, where the payment key is again vkp but the staking object is $\beta'$. Notice that $\mathcal{P}_A$ only knows the address $\alpha$, i.e., neither the payment key vkp nor any other information, e.g., other addresses that $\mathcal{P}_B$ owns. Now, $\mathcal{P}_B$ can spend from $\alpha'$, so $\mathcal{P}_A$ can claim that the payment is valid and complete, even though $\mathcal{P}_A$ has effectively chosen the key that controls that stake. To make matters worse, the malleability attack may go unnoticed by $\mathcal{P}_B$, unless she keeps a state of her generated addresses and compares with the forged one.

A malleable address is constructed by concatenating the hash of the verification payment key vkp and the staking object $\beta$, so $\alpha = \mathsf{H}(\mathsf{vkp})||\beta$. The value $\mathsf{H}(\mathsf{vkp})$ acts as both the association of the address with the payment key and the wallet recovery tag wrt. The staking object $\beta$ takes the following forms, depending on the type of address: i) *base:* $\beta = \mathsf{H}(\mathsf{vks})$; ii) *pointer:* $\beta = \mathsf{getPointer}(\mathsf{vks})$; iii) *exile:* $\beta = \epsilon$. The malleable address generation function is defined in Algorithm 6, while Lemma 1 shows that MGenAddr is collision resistant.

**Lemma 1.** MGenAddr *is collision resistant if* H *is collision resistant.*

*Proof.* The proof is by contradiction. Suppose an adversary produces two attribute lists $l_1 = (\mathsf{aux}_1, \delta_1, \mathsf{vkp}_1)$, $l_2 = (\mathsf{aux}_2, \delta_2, \mathsf{vkp}_2)$, such that $l_1 \neq l_2$, which correspond to the addresses $\alpha_1 = \mathsf{MGenAddr}(l_1) = \mathsf{H}(\mathsf{vkp}_1)||\beta_1$ and $\alpha_2 = \mathsf{MGenAddr}(l_2) = \mathsf{H}(\mathsf{vkp}_2)||\beta_2$.

If the adversary finds a collision of addresses, then $\alpha_1 = \alpha_2$ and it also holds that $\mathsf{H}(\mathsf{vkp}_1) = \mathsf{H}(\mathsf{vkp}_2)$ and $\beta_1 = \beta_2$. However, by assumption it holds that $l_1 \neq l_2$, so it also holds that $\mathsf{vkp}_1 \neq \mathsf{vkp}_2$ and the adversary has found a collision for H. □

MGenAddr is malleable since, for every address $\alpha = \mathsf{H}(\mathsf{vkp})||\beta$ that $\mathcal{P}_B$ gives to $\mathcal{P}_A$, $\mathcal{P}_A$ can create a new address as $\alpha' = \mathsf{H}(\mathsf{vkp})||\beta'$ for some staking object $\beta' \neq \beta$. Also, $\mathcal{P}_B$ cannot identify a forgery without keeping track of the addresses that she has

**Algorithm 6** The malleable address generation function, parameterized by $\mathsf{H}(\cdot)$. The input is a tuple $l_{\alpha,Gen}$, consisting of the auxiliary information aux and the attributes.

**function** MGenAddr($l_{\alpha,Gen}$)
  $(\mathsf{aux}, st, \mathsf{vkp}) = \mathsf{parse}(l_{\alpha,Gen})$
  **switch** aux **do**
    **case** base
      $\beta = \mathsf{H}(\mathsf{vks})$
    **case** pointer
      $\beta = \mathsf{getPointer}(\mathsf{vks})$
    **case** exile
      $\beta = \epsilon$
  $\alpha = \mathsf{H}(\mathsf{vkp})||\beta$
  **return** $\alpha$
**end function**

honestly generated. However, this is not always feasible, e.g., wallet recovery, when the wallet owner knows the payment keys but does not necessarily keep track of the staking object for each address in the wallet.

**Verifiably malleable addresses.** This scheme is similar to the above, with the addition that the wallet can identify a forgery. The address $\alpha$ is now constructed by concatenating the string generated by MGenAddr, as before, and the malleability verification tag, which is constructed as in Section 4.6.1:

$$\mathsf{mht} = \mathsf{H}(\mathsf{msk}||i||\beta) \tag{4.3}$$

$$\alpha = \mathsf{H}(\mathsf{vkp})||\beta||\mathsf{mht} \tag{4.4}$$

This scheme is also susceptible to the malleability attack described above, since, given $\alpha = \mathsf{H}(\mathsf{vkp})||\beta||\mathsf{mht}$, $\mathcal{A}$ can create a forgery like $\alpha' = \mathsf{H}(\mathsf{vkp})||\beta'||\mathsf{mht}$. However, now the owner of the wallet can compare the staking object $\beta'$ with the tag mht and identify the attack, while also using $\mathsf{H}(\mathsf{vkp})$ as the recovery tag.

### 4.6.3 A Posteriori Malleable Addresses

In this section we describe various schemes of a posteriori malleable address generation. First, Algorithm 7 defines an a posteriori verifiable malleable address generation function PNMGenAddr.

**Algorithm 7** A posteriori malleable address generation function, parameterized by $\mathsf{H}(\cdot)$. The input is a tuple $l_{\alpha,Gen}$, consisting of the auxiliary information aux and the attributes.

**function** PNMGenAddr($l_{\alpha,Gen}$)
  $(\mathsf{aux}, \mathsf{vks}, \mathsf{vkp}, \mathsf{ht}) = \mathsf{parse}(l_{\alpha,Gen})$
  **switch** aux **do**
    **case** base
      $\beta = \mathsf{H}(\mathsf{vks})$
    **case** pointer
      $\beta = \mathsf{getPointer}(\mathsf{vks})$
    **case** exile
      $\beta = \epsilon$
  $\mathsf{root} = \mathsf{H}(\mathsf{vkp}||\mathsf{ht}||\beta)$
  $\alpha = \mathsf{root}||\mathsf{ht}||\beta$
  **return** $\alpha$
**end function**

In this case, $\mathcal{A}$ cannot act as above. Specifically, if $\mathcal{A}$ changes $\beta$ to some $\beta'$, without changing root, then, upon spending from the address, root will be invalid. Since $\mathcal{A}$ does not know vkp, it cannot construct a valid root for both vkp and $\beta'$, unless explicitly asking $\mathcal{P}_B$ for such address. Alternatively, if she only changes root and keeps $\beta$, it can be easily found that root $\neq \mathsf{H}(\mathsf{vkp}||\mathsf{ht}||\beta)$, so the payment is rejected and the assets are effectively burnt in the malformed address. We note that tag ht is necessary, since root includes the staking object, so a wallet that knows only the payment key could not recreate it during recovery.

However, this scheme allows for a posteriori malleability, as defined in Section 4.2. Specifically, $\mathcal{A}$ can perform the attack if she knows a past key of the wallet, for which she wishes to generate the forged address. In this case, both the wallet recognizes the forged address as its own and the owner is able to spend from it.

Finally, as with the malleable addresses, this scheme can be transformed into a verifiable one via a verification tag. The tag would be $\mathsf{ht} = (\mathsf{ht}_1, \mathsf{mht})$, where $\mathsf{ht}_1$ is the recovery tag, for either a predefined index iht or a searchable one sht, and mht is the malleability verification tag.

### 4.6.4 Sink Malleable Addresses

Our final address scheme is a sink malleable construction. The core idea is to certify the staking object with the payment key; thus, to produce a forgery, the adversary needs to forge a payment key's signature. A sink malleable address is as follows:

$$\alpha = \mathsf{H}(\mathsf{vkp})||\beta||\mathsf{Sign}(\mathsf{skp}, \beta) \tag{4.5}$$

The value $\mathsf{H}(\mathsf{vkp})$ associates the address with the payment key, while also serving as the recovery tag wrt. The staking object $\beta$ for the three address types is: i) *base:* $\beta = \mathsf{H}(\mathsf{vks})$; ii) *pointer:* $\beta = \mathsf{getPointer}(\mathsf{vks})$; iii) *exile:* $\beta = \epsilon$.

In practice, upon issuing a transaction and revealing the payment key vkp, everybody can check the signature to identify forgeries. Algorithm 8 formally defines the sink malleable construction, while Lemmas 2 and 3 prove that our scheme is both collision resistant and non-malleable.

**Algorithm 8** The sink malleable address generation function, parameterized by a hash $\mathsf{H}(\cdot)$ and a signature scheme $\Sigma$. The input is a tuple $l_{\alpha,Gen}$, consisting of the auxiliary information aux and the attributes.

**function** SinkGenAddr($l_{\alpha,Gen}$)
  $(\mathsf{aux}, st, \mathsf{vkp}, \mathsf{skp}) = \mathsf{parse}(l_{\alpha,Gen})$
  **switch** aux **do**
    **case** base
      $\beta = \mathsf{H}(st)$
    **case** pointer
      $\beta = \mathsf{getPointer}(st)$
    **case** exile
      $\beta = st$
  $\alpha = \mathsf{H}(\mathsf{vkp})||\beta||\mathsf{Sign}(\mathsf{skp}, \beta)$
  **return** $\alpha$
**end function**

**Lemma 2.** SinkGenAddr *is collision resistant if* H *is collision resistant.*

*Proof.* Let $l_1 = (\mathsf{aux}_1, st_1, \mathsf{vkp}_1, \mathsf{skp}_1)$, $l_2 = (\mathsf{aux}_2, st_2, \mathsf{vkp}_2, \mathsf{skp}_2)$ be two attribute lists, $l_1 \neq l_2$, corresponding to addresses $\alpha_1 = \mathsf{MGenAddr}(l_1) = \mathsf{H}(\mathsf{vkp}_1)||\beta_1||\mathsf{Sign}(\mathsf{skp}_1, \beta_1)$ and $\alpha_2 = \mathsf{MGenAddr}(l_2) = \mathsf{H}(\mathsf{vkp}_2)||\beta_2||\mathsf{Sign}(\mathsf{skp}_2, \beta_2)$.

If $\alpha_1 = \alpha_2$ then it also holds that $\mathsf{H}(\mathsf{vkp}_1) = \mathsf{H}(\mathsf{vkp}_2)$, $\beta_1 = \beta_2$, and $\mathsf{Sign}(\mathsf{skp}_1, \beta_1) = \mathsf{Sign}(\mathsf{skp}_2, \beta_2)$. However, by assumption we have $l_1 \neq l_2$, so it also holds that $\mathsf{vkp}_1 \neq \mathsf{vkp}_2$ and thus the adversary has found a collision for H on the two values of the payment verification keys and signatures on the staking object. □

**Lemma 3.** SinkGenAddr*, when parameterized with a signature scheme* $\Sigma$*, is attribute non-malleable if* $\Sigma$ *is EUF-CMA.*

*Proof.* Assume the key pair $(\mathsf{vkp}, \mathsf{skp})$ and the address $\alpha = \mathsf{H}(\mathsf{vkp})||\beta||\mathsf{Sign}(\mathsf{skp}, \beta)$, for staking object $\beta$. Also assume the existence of an adversary $\mathcal{A}$ who breaks the attribute non-malleability property of SinkGenAddr. We will construct a forger $F$ for the signature scheme, which simulates the security game for $\mathcal{A}$. The forger works as follows. $F$ receives a key vkp and has access to the signing oracle. It sets the attribute list $l = (\mathsf{vkp}, \mathsf{aux}, \beta, \mathsf{wrt})$ and initializes $\mathcal{A}$ with public attributes $(\mathsf{aux}, \beta, \mathsf{wrt})$. Note that $F$ answers generation address queries by using its own signature oracle. That is, upon receiving $(\mathsf{aux}_i, \beta_i, \mathsf{wrt}_i)$, it issues a signature query on $\beta_i$ and generates a new address $\alpha_i$. Moreover, $F$ may receive a metadata query on issued addresses $\alpha_i$ and answer by revealing vkp. By hypothesis $\mathcal{A}$ outputs a list $(\alpha^*, \mathsf{aux}^*, \beta^*, \mathsf{wrt}^*)$, as per the attribute non-malleability game, for which it holds that $\mathsf{SinkGenAddr}(\mathsf{skp}, \mathsf{vkp}, \mathsf{aux}^*, \beta^*, \mathsf{wrt}^*) \rightarrow \alpha^*$, where $\alpha^*$ was not queried during the game and $\alpha^* \neq \alpha$, where $\alpha$ is the original address provided during the challenge: $\mathsf{SinkGenAddr}(\mathsf{skp}, \mathsf{vkp}, \mathsf{aux}, \beta, \mathsf{wrt}) \rightarrow \alpha$. Since $\mathcal{A}$ is successful, the signature holds for both $\alpha$ and $\alpha^*$, so $F$ uses $\alpha^* = (\mathsf{H}(\mathsf{vkp}||\sigma^*), \beta^*)$ to output $(\alpha^*, \sigma^*)$ as its pair of forged message and signature. □

**Remark.** *In an attempt to shorten the address, one may entertain the idea of including the signature in the hashed data. However, in such case, the hash could not act as the recovery tag anymore, since both the staking object* $\beta$ *and the signature cannot be predicted, given only the address's payment key. Therefore, either the wallet's recovery becomes necessarily linear to the number of addresses in the ledger, which is a significant performance overhead compared to the recovery mechanism of Section 4.6.1, or an additional component is introduced, i.e., a dedicated recovery tag, which effectively counters the address shortening effort. Alternatively, the address could contain the signature's hash, rather than the signature itself, as the signature is only checked after a payment is issued, i.e., when* vkp *is revealed. However, although such design could reduce the address's length, it would also increase the ledger's overall size, since both the signature and its hash would be published.*

## 4.7 The Proof-of-Stake Wallet

So far, we have covered the core wallet protocol and functionality and the respective security analysis. Now, we focus on the wallet's interaction with a PoS ledger. Specifically, we consider a PoS wallet with access to a core (as defined by $\mathcal{F}_{\text{CoreWallet}}$) and a PoS ledger, which enables a user to issue payments, participate in the PoS protocol, or delegate stake in the ledger. Specifically, in this section, we describe how to construct such PoS wallet and describe the issuing of payments, delegation, stake pool registration, as well as the generic PoS protocol participation's rules. Finally, we formalize the security of the "stake-pooled" variant of any PoS protocol and describe various modes of execution of the PoS wallet, which offer enhanced safety and privacy.

Our system depends on the following functions and properties, parameterized by a chain $\mathcal{C}$:

1. $F_\theta(\cdot)$: given an address $\alpha$, the function returns the assets that $\alpha$ owns;
2. $F_{\Phi,tx}(\cdot)$: given a transaction $\tau$, the function outputs the fees $\Phi$ of publishing $\tau$ on the ledger;
3. $F_{otx}(\cdot)$: given an address $\alpha$, the function outputs the number of $\alpha$'s outgoing transactions;
4. $\Phi_{reg}$: the (protocol-specific) cost of stake pool registration;
5. $\alpha_{reg}$: a special address that pertains to pool registration;
6. $F_{PoS,player}()$: given a chain, a function that outputs the next participant in the PoS protocol.

### 4.7.1 Payment

Payment is the transfer of $\Theta$ assets from a sender's address $\alpha_s$ to the receiver's address $\alpha_r$. To make a payment, the wallet first creates an object $\tau = (\Theta, \alpha_s, \alpha_r, m)$, with some metadata $m$. For a UTxO blockchain, the metadata contains a change address $\alpha_c$. For an account-based ledger, the wallet retrieves the number of output transactions $otx = F_{otx}(\alpha_s)$ and adds it to the metadata, as replay attack protection (see below Section 4.7.5). All addresses are generated via the *Address Generation* interface of $\mathcal{F}_{\text{CoreWallet}}$. Also the wallet calls $F_\theta(\alpha_s)$ to retrieve the balance $\Theta'$ of $\alpha_s$ and calculates the fees $\Phi = F_{\phi,tx}(tx)$.

After ensuring that the address has enough balance, i.e., that $(\Theta \cup \Phi) \subseteq \Theta'$, the wallet sends $(\mathsf{Pay}, \tau)$ to $\mathcal{F}_{\mathsf{CoreWallet}}$. When $(\mathsf{Transaction}, \tau, \sigma)$ is returned, the wallet publishes it on the network. The signed transaction can be easily verified by sending $(\mathsf{VerifyPay}, \tau, \sigma)$ to $\mathcal{F}_{\mathsf{CoreWallet}}$ and waiting for the response $\mathsf{VerifiedPay}(\tau, \sigma, 1)$.

### 4.7.2 Stake Pool Registration

A stake pool is identified by a registered staking key. Prior to registration, the pool's wallet first uses the address generation interface of $\mathcal{F}_{\mathsf{CoreWallet}}$ to compute a staking key $(\mathsf{vks}, \mathsf{sks})$. It then creates a registration certificate $r = (\mathsf{vks}, m)$, where $m$ is the pool's metadata, e.g., the name of the pool's leader. To register, the wallet sends $(\mathsf{Stake}, r)$ to $\mathcal{F}_{\mathsf{CoreWallet}}$. Upon receiving $(\mathsf{Staked}, r, \sigma)$, it publishes $\Sigma = (r, \sigma)$ on the ledger by creating a special payment transaction. Specifically, the wallet retrieves the registration fees $\Phi_{reg}$ and the special address $\alpha_{reg}$. It then uses an address $\alpha_s$, which it owns, to create $\tau = (\Phi_{reg}, \alpha_s, \alpha_{reg}, \Sigma)$, i.e., a transaction including the registration certificate in its metadata. This transaction is signed and published as above. To validate a registration certificate $\Sigma = (r, \sigma)$, a party sends $(\mathsf{VerifyStake}, r, \sigma)$ to $\mathcal{F}_{\mathsf{CoreWallet}}$ and waits for $(\mathsf{VerifiedStake}, r, \sigma, 1)$.

### 4.7.3 Delegation

Stake delegation is achieved with certificates via a process similar to the staking pool registration. A delegation certificate is a tuple $d = (\mathsf{vks}_s, \langle \mathsf{vks}_d, m \rangle)$. The first element is the staking key $\mathsf{vks}_s$, on which the certificate applies. The second is the staking key $\mathsf{vks}_d$ of the delegate. The third element is the certificate's metadata. To sign the delegation certificate, the wallet sends $(\mathsf{Stake}, d)$ to $\mathcal{F}_{\mathsf{CoreWallet}}$. Upon receiving $(\mathsf{Staked}, d, \sigma)$ it publishes $\Sigma = (d, \sigma)$ on the ledger, following the same method as pool registration, i.e., via a payment transaction. As before, a party validates $\Sigma$ via the staking verification interface of $\mathcal{F}_{\mathsf{CoreWallet}}$.

An address with staking object $\beta$ is associated with a staking key $\mathsf{vks}_s$ in two ways: i) *base address*: $\beta = \mathsf{H}(\mathsf{vks}_s)$; ii) *pointer address*: $\beta$ needs to point to either a delegation certificate $(\mathsf{vks}_s, \cdot, \cdot, \cdot)$ or a registration certificate $(\mathsf{vks}_s, \cdot, \cdot)$, i.e., the pointer address is associated with the staking key defined in the certificate to which it points.

We stress that, when a staking key issues a delegation certificate, *all* associated addresses are re-delegated accordingly. Specifically, the certificate to which a pointer address points is used *only* to identify the staking key with which it is associated, and does

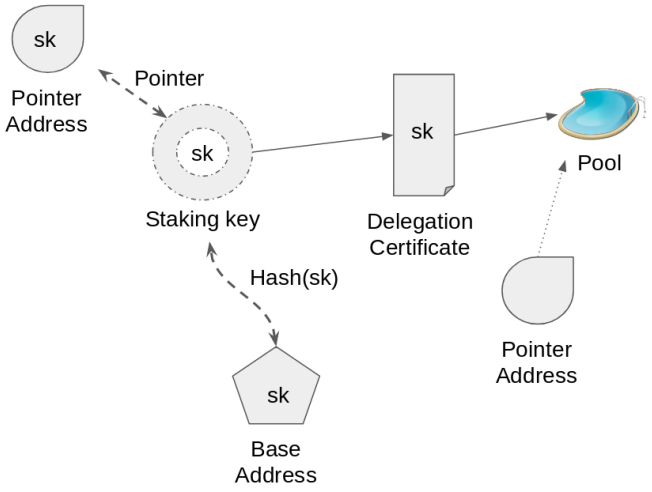


Figure 4.6: A representation of the delegation mechanism. Base pointers are linked to a key via a hash, whereas pointers may point to either a staking key or a pool directly.

not necessarily define its delegation profile. For example, let $\alpha_p$ be a pointer address that points to delegation certificate $(\mathsf{vks}_s, \mathsf{vks}_d, m, \sigma)$, thus is initially associated with $\mathsf{vks}_s$. Now, $\mathsf{vks}_s$ publishes a newer certificate $(\mathsf{vks}_s, \mathsf{vks}'_d, m', \sigma')$. All addresses associated with $\mathsf{vks}_s$, including $\alpha_p$, are now delegated to $\mathsf{vks}'_d$. Figure 4.6 illustrates delegation for various types of addresses.

Delegation certificates come in two forms, depending on how they are published. *Heavyweight* certificates are published on the ledger and are identified by an *index*, which represents the position of the certificate in the ledger. The index is the tuple $ptr = (b, x, c)$, where $b$ relates to a block in the ledger, $x$ represents a transaction in $b$, and $c$ identifies a certificate in the metadata of $x$. Therefore, every pointer address contains the index of the certificate to which it points. If an address $\alpha$ points to an invalid certificate, then $\alpha$'s stake is not delegated. *Lightweight* certificates are not published by default, but instead become public when their staking key participates in the PoS protocol. Conflicts are resolved based on seniority. For instance, let $(\mathsf{vks}, \mathsf{sks})$ be a staking key which issues two certificates $\Sigma_1, \Sigma_2$. If they are published on the ledger in that order, $\Sigma_2$ takes effect for all addresses associated with $(\mathsf{vks}, \mathsf{sks})$. The metadata of a lightweight certificate also contain a "counter", i.e., an integer that breaks ties between lightweight certificates; if two conflicting lightweight certificates are presented, then whichever has the higher counter is accepted.

**Certificate length.** Let the curve of the ECDSA scheme be *secp256r1* [GBWM+04] and the hash function be *SHA256*. The staking pool registration certificate is the tuple $\Sigma = (\mathsf{vks}, m, \sigma)$. The public key size for vks (in the compressed form) is $33$ bytes and the signature $\sigma$ is $132$ bytes. The metadata value $m$ depends on the implementation; for simplicity, we can assume that it consists of only a hash, so its length is $32$ bytes. Therefore, the staking pool certificate is $197$ bytes. The delegation certificate is the tuple $\Sigma = (\mathsf{vks}_s, \mathsf{vks}_{delegate}, m, \sigma)$. As before, the public keys $\mathsf{vks}_s, \mathsf{vks}_{delegate}$ are $33$ bytes each, whereas the signature $\sigma$ is $132$ bytes. The metadata $m$ contains the counter for the lightweight certificates; setting it to $1$ byte enables up to $256$ conflicting lightweight certificates. Therefore, heavyweight and lightweight delegation certificates are $198$ and $199$ bytes respectively.

### 4.7.4 Protocol Participation

Participation in the PoS protocol consists of publishing specially-crafted, protocol-related, signed messages. The address which is responsible for participation at any given time is decided by the function $F_{PoS,player}$ and its staking key is used to sign these messages. Here, we consider the typical example of PoS participation, *block generation*, i.e., the act of extending an existing chain with a newly-created block. For simplicity, a block is a tuple $(\mathsf{vks}, m)$, where $(\mathsf{vks}, \mathsf{sks})$ is the staking key which issues the block and $m$ is the block's contents, i.e., the headers, the tree of transactions, etc. As with delegation and stake pool registration, the wallet obtains a signature for a new block via the staking interface of $\mathcal{F}_{\mathsf{CoreWallet}}$, while a verifier can similarly check it. However, whether a block is valid for a given ledger, i.e., whether a block can extend a chain, depends on the protocol's chain validity rules. Notably, delegation affects these rules. Next, we describe the validity rules in the presence of delegation, as well as the rules that pertain to chain delegation, i.e., the re-delegation of delegated stake.

**Block Validity.** For a chain $\mathcal{C}$, the chosen address $\alpha_{PoS}$ is associated with a staking key $\mathsf{vks}_{PoS}$ and a candidate block $\mathcal{B}$ is signed by a different staking key $(\mathsf{vks}, \mathsf{sks})$. The rules for deciding if $\mathcal{B}$ is valid for $\mathcal{C}$ are as follows:

- if a delegation certificate for vks is published in $\mathcal{C}$, $\mathcal{B}$ is invalid;
- else if a certificate that delegates from $\mathsf{vks}_{PoS}$ to vks is published in $\mathcal{C}$, $\mathcal{B}$ is valid;
- else if either no delegation for $\mathsf{vks}_{PoS}$ or a certificate that delegates from $\mathsf{vks}_{PoS}$

to $\text{vks}_h$ is published in $\mathcal{C}$ and $\mathcal{B}$ contains a lightweight certificate delegating from $\text{vks}_h$ to vks, $\mathcal{B}$ is valid;

- else $\mathcal{B}$ is invalid.

**Chain Validity.** The empty chain $\mathcal{C} = \epsilon$ is valid. Given a candidate chain $\mathcal{C} = \mathcal{C}'||\mathcal{B}$, if $\mathcal{C}'$ is valid and the block $\mathcal{B}$ is valid for $\mathcal{B}$, then $\mathcal{C}$ is valid.

**Chain Decision.** Assume $\mathcal{C}$ is the chain stored locally by the wallet and $\mathbb{C}$ is the set of valid chains available on the network. The longest valid chain from $\mathbb{C} \cup \{\mathcal{C}\}$ is chosen. In case of tie between two valid chains $\mathcal{C}_1$ and $\mathcal{C}_2$, where head$(\mathcal{C}_1)$ and head$(\mathcal{C}_2)$ are signed by $(\text{vks}_1, \text{sks}_1)$ and $(\text{vks}_2, \text{sks}_2)$ respectively, the following rules apply:

- if $\text{vks}_1$ and $\text{vks}_2$ are delegated via the heavyweight certificates $\Sigma_1$ and $\Sigma_2$, with indexes $idx_1$ and $idx_2$ respectively, $\mathcal{C}_1$ is chosen if $idx_1 > idx_2$, otherwise $\mathcal{C}_2$ is chosen;
- else if $\text{vks}_1$ is delegated via the heavyweight certificate $\Sigma_1$ and $\text{vks}_2$ is delegated via the *lightweight* certificate $\Sigma_2$, $\mathcal{C}_1$ is chosen;
- else if $\text{vks}_1$ and $\text{vks}_2$ are delegated via a combination of heavyweight and lightweight certificates $(\Sigma_{1,1}, \Sigma_{1,2})$ and $(\Sigma_{2,1}, \Sigma_{2,2})$ respectively, then:
  - if $\Sigma_{1,1}$ and $\Sigma_{2,1}$ have different indexes, then choose the one with the higher index;
  - else if $\Sigma_{1,2}$ and $\Sigma_{2,2}$ have different counters, then choose the one with the higher counter;
  - else choose the first observed on the network.

**Delegation Chains.** Chain delegation is the ability of a staking key to re-delegate stake that has been delegated to it. As described in Section 4.7.3, the metadata section of a delegation certificate defines the rules that pertain to the certificate. One such rule relates to chain delegation. The metadata entry for this property is a boolean value, "allowChain", which identifies whether chain delegation is allowed for the applicable stake of the certificate. For example, let two certificates $\Sigma_0$ and $\Sigma_1$, where $\Sigma_0$ delegates from a key $\text{vks}_0$ to $\text{vks}_2$ and sets "allowChain" $= true$, whereas $\Sigma_1$ delegates from $\text{vks}_1$ to (the same key as before) $\text{vks}_2$ and sets "allowChain" $= false$. Suppose now that a third

certificate $\Sigma_3$ is published, delegating from $\mathsf{vks}_2$ to $\mathsf{vks}_3$. Although $\mathsf{vks}_3$ is eligible to participate on behalf all addresses associated with $\mathsf{vks}_0$, it cannot participate for those associated with $\mathsf{vks}_1$, since the corresponding key is $\mathsf{vks}_2$ (as chain delegation is not permitted for this key). More concretely, a delegation chain is a list of certificates $[\Sigma_1, \ldots, \Sigma_i]$ such that, for each certificate $\Sigma_j, 1 < j \leq i$, it holds that $\Sigma_{j-1}[\mathsf{vks}_d] = \Sigma_j[\mathsf{vks}_s]$ We say that vks is delegated via a hain $[\Sigma_1, \ldots, \Sigma]$ if $\Sigma[\mathsf{vks}_d] = \mathsf{vks}$.

### 4.7.5 Security in the Presence of Stake Pools

In this section, first we analyze the security of stake pools w.r.t. the underlying PoS protocol's security assumptions (cf. Corollary 1). Next, we discuss prominent hazards, namely *sybil* and *replay* attacks.

**Stake-pooled Security.** The security analysis of a PoS stake-pooled variant, i.e., a PoS protocol with stake pools, is based on the protocol's honest stake threshold assumption $\tau$. This parameter identifies the minimum percentage of honest stake needed for the protocol to be secure. It is typically set to $\frac{1}{2} + \epsilon$ or $\frac{2}{3} + \epsilon$, for some $\epsilon > 0$. For simplicity, we assume that all stake is delegated to a total number of $P$ pools. Each pool possibly controls a different amount of stake. Let $P_h$ be the honest pools among $P$, which control an aggregate $\rho_h$ percentage of the total stake. When a player delegates their stake, they effectively relinquish their staking rights. Therefore, as Corollary 1 shows, the focus should be put on the adversarial power over pools, rather than stake itself. For example, if the adversary delegates some stake to an honest pool, then this stake becomes honest in the stake-pooled setting, as it is controlled by an honest leader. Intuitively, the adversary compromises the stake-pooled variant's security by corrupting an appropriate amount of pools, such that honestly-controlled stake percentage is less than $\tau$. We stress that this does not imply that the adversary is required to corrupt a large number of pools. For instance, if a single pool controls $\rho_a \geq 1 - \tau$ of the total stake, then the adversary can compromise security by only corrupting this single, albeit large, pool.

**Corollary 1.** *The stake-pooled variant of a PoS protocol $\pi$ is secure if $\rho_h \geq \tau$, where $\rho_h$ is the percentage of the total stake controlled by pools which are managed by honest leaders and $\tau$ is the honest stake threshold assumption of $\pi$.*

**Sybil Attacks.** Using stake pools for the PoS protocol's execution, rather than the stakeholders themselves, introduces the possibility of *sybil attacks* [Dou02]. Specifically, suppose that the adversary creates a large number of stake pools. Honest players cannot immediately identify whether a pool is adversarial, so these pools could appear legitimate and (honest) users might be convinced to delegate to them, thus increasing the adversarial stake ratio. This is an inherent problem to decentralized PoS systems, as no form of external identification exists and an adversary can easily create a large number of staking keys and registration certificates. A potential countermeasure is to have pool leaders commit (some of) their own stake to their pool. In our setting, this method can be facilitated via an extra field in the delegation certificate's metadata, which identifies the leader's addresses and funds, which are committed to the pool. Evidently, as long as these funds are locked in the corresponding addresses, a malicious leader cannot use them for multiple pool commitments. While this does not directly prevent a Sybil attack, it does force the attacker to commit some stake to its pools, hence bounding its identity production capabilities.

**Replay Attacks.** Another important consideration is replay protection. Replay attacks are prominent in account-based ledgers, where an adversary may re-publish past transaction. For instance, suppose Alice sends $x$ assets from her address-account $\alpha$ to Bob. After the payment is published, $\alpha$ controls $y = z - x$ assets, $z$ being the funds that $\alpha$ controlled before Alice made the payment. In a replay scenario, Bob re-publishes this payment, such that a further amount $x$ of funds is sent from Alice's account $\alpha$ to Bob's. The same vulnerability exists against the certificates of our scheme. For instance, an attacker can re-publish a certificate to forcefully change a user's delegation choice. To solve this issue, we employ an address *whitelist*. Specifically, each certificate defines the addresses allowed to publish it. Naturally, this scheme assumes that the wallet knows these addresses a priori. Next, during certificate verification, a party checks whether it is published in a transaction issued by a whitelisted address. To replay the certificate, the adversary would then need to obtain the private payment key of one of the whitelisted addresses. Notably, our solution requires no state to be maintained by the verifiers, as the information needed to counter a replay attack, i.e., the address whitelist, exists in the certificate itself. Therefore, there is no the need to parse the entire ledger or maintain extra local state, as is the case with counter-based replay protection mechanisms [Eth18a].

### 4.7.6 Modes of Execution

Our final contribution is a set of modes of operation of a PoS wallet.

**Regular.** This being most straightforward wallet deployment method, a regular wallet is bootstrapped with a *base* address $\alpha_0$ and its stake is managed by a key $(\mathsf{vks}, \mathsf{sks})$. After $\alpha_0$ receives its first assets, the wallet may perform staking actions by using $(\mathsf{vks}, \mathsf{sks})$. To perform staking on its own, the wallet publishes a delegation certificate $\Sigma$ that delegates to its own key $(\mathsf{vks}, \mathsf{sks})$. Subsequent addresses are pointer addresses to $\Sigma$, so eventually all addresses are managed by the same staking key. When the user wishes to delegate to a staking pool, identified by the key $\mathsf{vks}_P$, the wallet simply publishes a certificate $\Sigma_d$ delegating from vks to $\mathsf{vks}_P$.

**Offline with Cold Staking.** This wallet lives in an offline device, e.g., on paper. It is rarely accessed for payments, but regularly performs staking actions. It is bootstrapped similarly to the regular wallet and, since its payment keys are stored offline, the staking keys are managed as follows:

- *basic security*: the staking key $(\mathsf{vks}, \mathsf{sks})$, which manages all addresses, is online; in case sks is compromised, the user accesses the payment keys and sends the funds to new addresses, controlled by a new staking key;
- *enhanced security*: the wallet creates a certificate $\Sigma'$, which delegates from vks to the "hot" key $\mathsf{vks}_h$; after $\Sigma'$ is published, the user stores $(\mathsf{vks}, \mathsf{sks})$ offline and $(\mathsf{vks}_h, \mathsf{sks}_h)$ online; in case $(\mathsf{vks}_h, \mathsf{sks}_h)$ is compromised, $(\mathsf{vks}, \mathsf{sks})$ is used to delegate to a new "hot" key, without requiring access to the wallet's payment keys.

**Enhanced Unlinkability of Addresses.** This wallet aims at a higher level of privacy, with each address managed by a single staking key. Therefore, to change its delegation profile, the wallet creates a certificate for each address. Additionally, it achieves different security guarantees as follows:

- *online:* the wallet is online and creates only pointer addresses, which point to the stake pool's registration certificate; to re-delegate, it creates new pointer addresses and moves the funds from the old to the new addresses;
- *offline:* the payment keys are stored offline, so the wallet creates base addresses, each managed by a unique staking key, and keeps the staking keys online; to re-delegate, it publishes a new certificate for each address.

**The Stake Pool's Wallet.** A stake pool's wallet performs only staking actions, so its key $(\mathsf{vks}_P, \mathsf{sks}_P)$ is managed as follows:

- *basic security*: the key pair $(\mathsf{vks}_P, \mathsf{sks}_P)$ is stored online and is used directly for staking; in case of compromise, the wallet creates a new staking key while, for practical purposes, an alert mechanism should exist to notify the users to re-delegate their stake to the new key;
- *enhanced security*: the wallet creates a lightweight certificate $\Sigma_l$, which delegates to a "hot" key $\mathsf{vks}_{Ph}$, and then stores $(\mathsf{vks}_P, \mathsf{sks}_P)$ offline, while using $(\mathsf{vks}_{Ph}, \mathsf{sks}_{Ph})$ and $\Sigma_l$ for staking; if $(\mathsf{vks}_{Ph}, \mathsf{sks}_{Ph})$ is compromised, the wallet creates a new hot key $(\mathsf{vks}'_{Ph}, \mathsf{sks}'_{Ph})$ and a new lightweight certificate $\Sigma'_l$, which delegates from $(\mathsf{vks}_P, \mathsf{sks}_P)$ to $\mathsf{vks}'_{Ph}$ and includes a higher counter compared to $\Sigma_l$, using them for staking instead.

We note that, in particular, the stake pool's wallet is further explored in the upcoming chapter on collective stake pools.

## 4.8 Discussion

Our framework offers a range of choices for embedding information in PoS addresses, as well as enabling multiple address types. The former allows the addition of *metadata* into the addresses, including keys for staking and spending and *device and account identification tags*. The latter gives the ability to various players, e.g., enterprises such as exchanges, to operate using well-crafted, special-purpose addresses that are fit for special needs. These features embrace a wide range of the desiderata outlined in Section 4.1. *Address Non-Malleability* is addressed during the address generation phase, described in detail in Section 4.2. Moreover, the two types of keys, for payment and staking, cover the need for *Staking and Spending Separation*, whereas *Key Exposure Mitigation* is achieved by providing the flexibility to issue new delegation certificates using the "staking action" interface, in case the staking key of the delegate is compromised. *Address Uniqueness* is addressed by the checks that the functionality performs upon receiving a possible address from the adversary. Additionally, the flexibility on defining attributes allows for *Multiple Devices Support* and *Address Recovery*, by constructing special tags that are embedded in the address. Furthermore, *Delegation Verification* is possible by obtaining the delegation certificates that pertain to its staking key. Another advantage of this design is the *Cost Effectiveness* of the delegation mechanism, assigning and changing a delegate at

the cost of only one transaction. The delegation mechanism also allows a party to prove that it has the right to append the ledger, although possibly restricting *Chain Delegation*. Finally, our framework enables a smooth *bootstrapping* delegation process, by allowing an initial delegation assignment phase which depends on the implementation details of the ledger.

Notably, our desiderata revolve primarily around key management and address generation. Therefore, we don't capture network reliability or availability requirements, but rather assume an abstract ledger model, which satisfies persistence and liveness in a synchronous setting. This allows us to focus on address and signature generation, while treating the other components of the system as black boxes, which presumably satisfy the required properties. This is evident in Section 4.7, which assumes a generic ledger model, abstracted as a set of variables and algorithms. Nonetheless, a more rigorous analysis on the incorporation of our wallet core in a complete wallet is an important next step. Such treatment could formally capture security on all layers, i.e., key management, consensus, network, *etc.* A possible path towards this goal could propose a variant of $\mathcal{F}_{\mathsf{CoreWallet}}$ with key-evolving signatures, which would replace $\mathcal{F}_{\mathsf{KES}}$ in a protocol like Ouroboros Praos [DGKR18], followed by a rigorous proof of security of this consensus protocol variant.

# Chapter 5

# Collective Stake Pools

Both PoW and PoS ledgers are economies of scale, favoring parties with large amounts of participating power. One reason is poorly-designed incentives, resulting in disproportionate power accumulation [KKNZ19, FKO+19]. Another is temporal discounting, i.e., the tendency to disfavor rare or delayed rewards [RL11]. Specifically, in Bitcoin, a party is rewarded for every block it produces, so parties with insignificant amounts of power are rarely rewarded. In contrast, accumulating the power of multiple small parties in "pools" yields a steadier reward.

As a result, these systems often see the formation of collaborative entities of participants. In PoW systems, this takes the form of mining pools.[1] Similarly, PoS systems often opt for stake pools, i.e., collaborative entities comprising of multiple stakeholders, which allow a party to earn rewards more regularly, compared to participating on an individual basis. Particularly in PoS, delegation to stake pools is often preferred over "pure" PoS, where parties act independently, as the ledger's performance and security is often better under fewer participants. For instance, PoS systems require participants to be constantly online, since abstaining is a security hazard; this requirement is more easily guaranteed within a small set of dedicated delegates.

However, a major drawback of existing stake pool designs, including our scheme of Chapter 4, is that they are typically managed by a single party, i.e., the pool operator. This party participates in consensus, claims the rewards offered by the system, and then distributes them among the pool's members (after subtracting a fee). However, the operator is a single point of failure. In this chapter, we extend the results of Chapter 4 by exploring a design which allows players to jointly form a *collective pool*,

[1] $86\%$ of Bitcoin's hashing power and $83\%$ of Ethereum's hashing power are controlled by $5$ entities each. (`https://miningpools.com`; May 2021)

i.e., a conclave. This design assumes no single operator, minimizing excess fees, and trust and security concerns, altogether. Collective stake pools also promote a more fair and decentralized environment. In existing incentive schemes [BKKS20], operators who can pledge large amounts of stake to the pool are preferred. Consequently, the system favors a few major pool operators and, in the long run, its wealth is concentrated around them, resulting in a "rich get richer" situation. Although this problem is inherent in all decentralized financial systems [KKNZ19], a well-designed collective pool may offset the stakeholder imbalance and slightly decelerate this tendency. Especially in PoS systems, a well-designed pool mechanism can prevent attacks observed on PoW [JLG+14, WC14, LJG15].

**Related Work.** In cryptographic literature, pools are mostly treated from an engineering perspective. In PoW systems, SmartPool [LVTS17] is a notable design of a distributed mining pool for Ethereum, which, similar to our work, utilizes smart contracts for reward distribution. On the PoS domain, Ouroboros [KRDO17] offers a brief description of how delegation can be used within the protocol. This idea is expanded in [KKL20], which provides a formal definition of PoS wallets and includes stake pool formation method via certificates. However, the pool's management is again centralized around the operator; our work extends this line of work by enabling the formation of a collective pool. Another work, orthogonal to ours, by Brünjes *et al.* [BKKS20] considers the incentives of distributing rewards among stake pools and aims to incentivize the creation of a (pre-defined) number of pools. However, it assumes that the pool operator commits part of their stake to make the pool more appealing, thus favoring larger pool operators. Our work eases such wealth concentration tendencies by enabling a collective pool to be equally competitive to a centralized one.

**Contributions.** The core contribution of this chapter is the ideal functionality $\mathcal{F}_{pool}$, a simulation-based security definition of collective stake pools, which captures the security properties of our collective pool scheme. We then describe $\pi_{pool}$, a distributed protocol executed by a set of $n$ parties $\mathbb{P}$ which realizes $\mathcal{F}_{pool}$. A major consideration and performance enhancement of our design is load balancing of transaction verification. Each transaction is verified by a (deterministically elected) committee of parties, whose size is a tradeoff between balancing workload, i.e., not requiring each party to verify every transaction, and reducing trust on the chosen validator(s). We thus construct a distributed mempool, i.e., a collectively managed set of unpublished transactions, s.t. if

a majority of the committee's members are honest, transaction verification is secure.

## 5.1 Desiderata

Our design assumes a group of stakeholders who jointly create a stake pool without a single operator. Since large stakeholders typically form pools on their own, our protocol concerns smaller stakeholders, who could otherwise not participate directly. Therefore, our design could e.g., be appealing to a group of friends or colleagues, who aim for a more steady reward ratio without relying on a third party. Importantly, it should operate in a trustless environment as, unfortunately, even in these scenarios, trust is not a given. Notably, our targeted audience is parties who wish to actively participate, i.e., always be online to perform the required consensus actions; parties who wish to remain offline may instead opt for delegation schemes [Com18, KKL20].

In the absence of a central party, the responsibility of running the pool is shared among all pool's members, requiring some level of coordination which may be cumbersome. For instance, if the protocol requires unanimous actions, a single member could halt the pool's operation. To ensure good performance, the pool should allow a subset (of a carefully chosen size) to act on behalf of the whole group. The choice of such subsets depends on each party's "weight", which is in proportion to their stake. In summary, we have the following initial assumptions, which form the basis for outlining our work's desiderata:

- *small number of parties*: a collective pool is operated by a small group of players;
- *small stake disparity*: the profiles of the collective pool's members are similar, i.e., they contribute a similar amount of stake to the pool;
- *stake proportion as "weight"*: each party is assigned a weight for participating in the pool's actions, relative to their part of the pool's total stake.

Next, we provide an exhaustive list of basic requirements of a collective stake pool. We note that an *admissible party set* is a set of parties with enough stake, i.e., above a threshold of the total pool's stake which is agreed upon during the pool's initialization. To the extent that some desiderata are conflicting, our design will aim to satisfy as many requirements as possible:

- *Proportional Rewards*: the claim of each member on the entire pool's protocol rewards should be proportional to their individual contribution.

- *Joint Control of Rewards*: the members of a pool should jointly control the access to its funds.
- *Unilateral Reward Withdrawal*: at any point in time, a stakeholder should be able to claim their reward, accumulated up to that point, without necessarily interacting with other members of the pool.
- *Permissioned Access*: new users can join the pool following agreement by an admissible set of pool members.
- *Robustness against Aborting*: the pool should not fail to participate in consensus, unless an admissible set of members aborts or is corrupted.
- *Public Verifiability*: stake pool formation and operation should be publicly verifiable (s.t. consensus could take into account the aggregate pool's stake).
- *Stake Reallocation*: users should freely change their personal stake allocated to the pool, without interacting with other members of the pool.
- *Parameter Updates*: an admissible set of parties should be able to update the stake pool's parameters.
- *Force Removal*: an admissible set of parties should be able to remove a member from the pool.
- *Pool Closing*: an admissible set of parties should be able to permanently close the stake pool.
- *Prevention of Double Stake Allocation*: a party should not simultaneously commit the same stake to two different stake pools.

## 5.2 Execution Model

In our setting, the ledger's maintainers are the stake pools. An adversary can break the ledger's properties by corrupting a set of stake pools which are allocated a majority of the total stake. Consequently, since we assume that the majority of stake is owned by honest stakeholders, the adversary needs to corrupt at least one pool to which honest players delegate. Especially in the collective setting, the adversary may control a subset of the pool's members, although not a majority of them. Therefore, our work considers adversaries who attempt to violate the security of a collectively-owned stake pool while controlling a minority of its members. Security violations include producing invalid blocks or transactions, as well as abstaining from join the consensus protocol.

In terms of message delivery, we assume a synchronous network. As such, the adversary may delay messages up to an upper bound. This assumption will prove par-

ticularly useful in the implementation of the collective pool protocol, during which participants employ consensus and broadcast algorithms. We note that, although this assumption is realizable in a setting where parties are well-connected, it does not cover possible real-world threats, such as eclipse attacks. Nonetheless, exploring variations of the collective pool functionality and protocol designs over semi-synchronous or asynchronous networks presents an interesting problem, which would offer a more robust implementation.

### 5.2.1 Weighted Threshold Digital Signatures

In a weighted threshold digital signature scheme [MPSV99, Sha79], each party $\mathcal{P}$ is associated with a (integer) weight $\omega[\mathcal{P}] \geq 0$, where $\omega$ is a mapping of players to weights. A signature can be produced by any set of keys, the aggregate weight of which is above the defined threshold. The weighted threshold signature scheme (Definition 13) is constructed by combining a digital signature scheme (Definition 2) with a weighted threshold secret sharing scheme (Definition 12). Additionally, standard threshold signatures is a special case of the weighted variant, with $\omega[\mathcal{P}] = 1$ for every party $\mathcal{P}$.

**Definition 12** (Weighted Threshold Secret Sharing)**.** *A $(T, n, \omega)$-threshold secret sharing of a secret $x$ consists of $n$ shares $x_1, \ldots, x_n$, each associated with a weight $w_1, \ldots, w_n$, such that an efficient algorithm exists, that takes as input a set of shares $B$, with $\sum_{i \in B} w_i > T$, and outputs the secret value $x$. Any set of shares $B$ with $\sum_{i \in B} w_i \leq T$ cannot obtain any information about the secret $x$.*

**Definition 13** (Weighted Threshold Signature)**.** *Given a signature scheme $\Sigma$, a $(T, n, \omega)$-threshold signature scheme $\Sigma_{\mathsf{thresh}} = \langle \mathsf{ThreshKeyGen}, \mathsf{ThreshSign}, \mathsf{ThreshVerify} \rangle$, given $n$ parties $\mathcal{P}_1, \ldots, \mathcal{P}_n \in \mathcal{P}$, is defined as:*

- $\mathsf{ThreshKeyGen}(1^\kappa, \omega) \to (\mathsf{vk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n)$*: given the security parameter $\kappa$, outputs a public key* vk *and a list of private keys $\mathbb{K} = [\mathsf{sk}_1, \ldots, \mathsf{sk}_n]$ which form a $(T, n, \omega)$-threshold secret sharing of* sk*; the pair $(\mathsf{vk}, \mathsf{sk})$ has the same distribution as the keys output by* KeyGen *of Definition 2;*

- $\mathsf{ThreshSign}(m, B) \to \sigma$*: given a message $m$ and a set of private keys $B$, $B \subseteq \mathbb{K}$, outputs a signature $\sigma$;*

- $\mathsf{ThreshVerify}(m, \mathsf{vk}, \sigma) \to \{0, 1\}$*: a deterministic algorithm that, given a message $m$, a public key* vk*, and a signature $\sigma$ outputs $1$ if a signature is valid w.r.t. message $m$ and verification key* vk *(resp. $0$ if the signature is invalid).*

*A $(T, n, \omega)$-threshold signature scheme $\Sigma_{\mathsf{thresh}}$ is EUF-CMA if it presents the properties of Definition 2 and the following:*

**Threshold Completeness:** *For any message $m$, it holds:*

$$\mathsf{Pr}[(\mathsf{vk}, \mathbb{K}) \leftarrow \mathsf{ThreshKeyGen}(1^\kappa, \omega), \sigma \leftarrow \mathsf{ThreshSign}(m, B), \sum_{k \in B} w_k > T : 0 \leftarrow \mathsf{Verify}(m, \sigma, \mathsf{vk})] \leq \mathsf{negl}(\kappa)$$

*and*

$$\mathsf{Pr}[(\mathsf{vk}, \mathbb{K}) \leftarrow \mathsf{ThreshKeyGen}(1^\kappa, \omega), \sigma \leftarrow \mathsf{ThreshSign}(m, B), \sum_{k \in B} w_k \leq T : 1 \leftarrow \mathsf{Verify}(m, \sigma, \mathsf{vk})] \leq \mathsf{negl}(\kappa)$$

*where all the probabilities are computed over the random coins of the key generation and sign algorithms.*

### 5.2.2 Transactions, Blocks, and the Global Ledger

Our protocol utilizes two features of the Kachina [KKK21b] framework: i) the formalization of the ledger and ii) smart contracts.

**The Simple Ledger Functionality.** Our collective pool protocol interacts with a ledger functionality in a hybrid execution. One option is the formalization of Bitcoin's ledger from [BMTZ17]. However, this functionality is local, while we would prefer a global functionality, following the Global UC Framework [CDPW07], hence we will use the $\overline{\mathcal{G}}_{simpleLedger}$ functionality from Kachina [KKK21b]. The functionality is available in Figure 5.1, where $\prec$ defines the prefix operation, i.e., $\Omega \prec \Omega'$ means the state $\Omega$ is included in $\Omega'$, and, for readability and consistency purposes, we rename *transaction* ($\tau$) to *block* ($b$).

The $\overline{\mathcal{G}}_{simpleLedger}$ Functionality is generic enough to abstract transactions and blocks, focusing on the ledger's properties. However, in our setting we need to define these objects, in order to better formulate a real-world blockchain.

In this chapter, a transaction is $\tau = \langle \alpha_s, \alpha_r, v, f \rangle$, where i) $\alpha_s, \alpha_r \in \{0,1\}^*$ are the sender's and receiver's addresses respectively, ii) $v \in \mathbb{R}$ is the value transferred from $\alpha_s$ to $\alpha_r$, and iii) $f \in \mathbb{R}$ is the fees of the transaction. A block consists of an ordered list of transactions. To organize transaction in blocks, we assume a function blockify which, given a set of transactions and a chain, returns a block which can extend the chain, i.e., satisfies the validity requirements of the system.

Global Ledger Functionality $\overline{\mathcal{G}}_{simpleLedger}$

The functionality keeps a state $\Omega$ and a mapping $M$ of parties to states, both initially empty.

- When receiving a message $(\mathsf{SUBMIT}, b)$ from a party $p$, query $\mathcal{A}$ with $(\mathsf{BLOCK}, b)$.
- When receiving a message READ from a party $p$, return $M(p)$; if $p$ is $\mathcal{A}$, it returns $\Omega$.
- When receiving a message $(\mathsf{EXTEND}, \Omega')$ from $\mathcal{A}$, set $\Omega \leftarrow \Omega || \Omega'$.
- When receiving a message $(\mathsf{ADVANCE}, p, \Omega')$ from $\mathcal{A}$, if $M(p) \prec \Omega' \prec \Omega$ then set $M(p) \leftarrow \Omega'$.

Figure 5.1: The Simple Global Ledger Ideal Functionality.

**Reward Management via Smart Contracts.** We also employ the formal model of smart contracts from [KKK21b]. This model considers smart contracts from a privacy-preserving perspective. However, it also provides a UC definition of standard smart contracts, consisting of the universal machine $\mathcal{U}$, which acts as the oracle over the ledger's state, and the *core* contract $\Gamma$, as illustrated by Figure 5.3 adapted to our stake pool design. In our setting, the latter relates directly to the management of the rewards for the members of the pool, and therefore it is presented as an auxiliary (reward) functionality $\Gamma_{reward}$ in Section 5.4.2.

**Delegation and Stake Pools.** We utilize the UC Model for delegated PoS systems, as presented in Chapter 4. This framework partially fulfills our earlier desiderata. In particular, *Prevention of Double Stake Allocation* and *Public Verifiabilitty* are addressed by the certificate-based registration and revocation mechanisms. However, the remaining items do not seem immediately solved without further assumptions. For instance, if the members have the same proportion of shares, a standard threshold signature scheme could address more of our desiderata, e.g., *Offline and Online Participation*, *Pool Proportional Rewards*, *Joint Control of Rewards*, and *Robustness against Aborting*. Following, reconfiguration of the pool is accomplished by regenerating the registration certificate and the pool's threshold key. Other desiderata can be approached in a similar fashion.

Thus, our idea is to generalize the access structure of an efficient threshold signature scheme to add "weight" capabilities, such that the weights capture the pledged stake distribution among the pool's members.

## 5.3 UC Weighted Threshold Signature

In this section, we present the weighted threshold signature ideal functionality $\mathcal{F}_{wtss}$ (Figure 5.2). This functionality is used by the Collective Pool Protocol $\pi_{pool}$ to collectively sign certificates and new blocks. The functionality $\mathcal{F}_{wtss}$ is inspired by Almansa *et al.* [ADN06], which is in turn inspired by Canetti [Can03]. However, unlike Almansa *et al.*and similar to Canetti, during signature verification we consider the case of a corrupted signer, i.e., a set of parties such that the majority (of weights) is corrupted.

$\mathcal{F}_{wtss}$ interacts with a set of $n$ parties. Each party $\mathcal{P}_i$ is associated with an integer $w_i$, i.e., its weight. $\mathcal{F}_{wtss}$ also keeps the following, initially empty, tables: i) pubkeys: tuples $\langle sid, \mathsf{vk} \rangle$ of $sid$ and a public key vk; ii) sigs: tuples $(m, \sigma, \mathsf{vk}, f)$ of message $m$, a signature $\sigma$, a public key vk, and a verification bit $f$. The mapping $\omega[\mathcal{P}] \rightarrow w_{\mathcal{P}}$ denotes the weight of a party $\mathcal{P}$, while the term $\omega$ also denotes the set of keys the participating parties.

As highlighted in the definition, *completeness, consistency*, and *unforgeability* are enforced upon verification, whereas *threshold completeness* is enforced upon signature generation. Hence, it should be infeasible to issue a signature unless using keys with enough weight, i.e., above the threshold $T$.

## 5.4 The Collective Stake Pool

Our analysis is based on the UC Framework, following Canetti's formulation of the "real world" [Can00]. Specifically, we define the collective pool ideal functionality $\mathcal{F}_{pool}$, which distills the required (operational and security) properties; for readability, $\mathcal{F}_{pool}$ is divided in two parts, *management* and *consensus participation*. The ideal functionality is realized — in the "real world" — by the distributed protocol $\pi_{pool}$, which employs various established cryptographic primitives, and, therefore, $\pi_{pool}$ can described with auxiliary functionalities. Before proceeding with the functionality's definition, we first describe the hybrid execution of $\pi_{pool}$ and its building blocks.

Weighted Threshold Signature Functionality $\mathcal{F}_{wtss}$

Each message is associated with $sid = \langle \mathcal{P}, \omega, T, sid' \rangle$, where $\mathcal{P}$ is the set of parties, $\omega$ is a mapping of parties to weights, $T$ is the collective signature weight threshold, and $sid'$ is a unique identifier.

**Key Generation:** Upon receiving $(\mathsf{KeyGen}, \mathsf{sid})$ from every honest party $\mathcal{P} \in \mathcal{P}$, send $(\mathsf{KeyGen}, \mathsf{sid}, \mathcal{P})$ to $\mathcal{S}$. Upon receiving a response $(\mathsf{KeyGen}, \mathsf{sid}, \mathsf{vk})$ from $\mathcal{S}$, record $\langle sid, \mathsf{vk} \rangle$ to pubkeys and send $(\mathsf{KeyGen}, \mathsf{sid}, \mathsf{vk})$ to every party in $\mathcal{P}$. Following, all messages that do not contain the established $sid$ are ignored.

**Signature Generation:** Upon receiving $(\mathsf{Sign}, \mathsf{sid}, m)$ from a party $\mathcal{P}$, forward it to $\mathcal{S}$. After a subset of parties $\mathcal{P}' \subseteq \mathcal{P}$ has submitted a Sign message for the same $m$, and upon receiving $(\mathsf{Sign}, \mathsf{sid}, m, \sigma)$ from $\mathcal{S}$, check that $\sum_{\mathcal{P} \in \mathcal{P}'} \omega[\mathcal{P}] > T$ (*Note: This condition guarantees threshold completeness.*) Next, if $(m, \sigma, \mathsf{vk}, 0) \notin$ sigs (for the key vk that corresponds to $sid$ in pubkeys), record $(m, \sigma, \mathsf{vk}, 1)$ to sigs and reply with $(\mathsf{Sign}, \mathsf{sid}, m, \sigma)$.

**Signature Verification:** Upon receiving $(\mathsf{Verify}, \mathsf{sid}, m, \sigma, \mathsf{vk}')$ from $\mathcal{P}$, forward it to $\mathcal{S}$. Upon receiving $(\mathsf{Verified}, \mathsf{sid}, m, \sigma, \phi)$ from $\mathcal{S}$, set $f$ as next:

1. If $\mathsf{vk}' = \mathsf{vk}$ and $(m, \sigma, \mathsf{vk}, 1) \in$ sigs, $f = 1$. (*This guarantees completeness.*)
2. Else, if $\mathsf{vk}' = \mathsf{vk}$, the aggregate weight of the corrupted parties in $\mathcal{P}$ is strictly less than $T$, and $(m, \sigma, \mathsf{vk}, 1) \notin$ sigs, $f = 0$ and record $(m, \sigma, \mathsf{vk}, 0)$ to sigs. (*This guarantees unforgeability, if the aggregate weight of the corrupted parties is below the threshold.*)
3. Else, if $(m, \sigma, \mathsf{vk}', b) \in$ sigs, $f = b$. (*This guarantees consistency.*)
4. Else, $f = \phi$ and record $(m, \sigma, \mathsf{vk}', f)$ to sigs.

Finally, send $(\mathsf{Verified}, \mathsf{sid}, m, \sigma, \mathsf{vk}', f)$ to $\mathcal{P}$.

Figure 5.2: Weighted Threshold Signature Ideal Functionality

### 5.4.1 Hybrid Protocol Execution

The protocol $\pi_{pool}$ is performed by $n$ parties, where each party $\mathcal{P}_i$ holds two pairs of keys: $(\mathsf{vk}_{\mathcal{P}_i}, \mathsf{sk}_{\mathcal{P}_i})$ for issuing transactions, and $(\mathsf{vk}_{s_i}, \mathsf{sk}_{s_i})$ for staking operations, e.g., issuing delegation certificates (cf. [KKL20]). The public key $\mathsf{vk}_i$ is also used to generate an address $\alpha_i$. Each pool member $\mathcal{P}_i$ pledges the funds of an address $\alpha_i$ (which it owns) to the pool. These funds are the player's stake in the pool and form the player's weight in the weight distribution mapping $\omega$.

We assume the members' stake, i.e., their weight $w_i$ in the pool, is public. Therefore, the weight distribution mapping $\omega$ is also public. Furthermore, each member of the pool has its own signature key, and can issue standard signatures through a standard signature scheme. A weighted version for a threshold signature scheme follows by having each party holding as many shares, of the original threshold scheme, as its weight. This approach has the extra advantage that security guarantees of the original scheme are carried straightforwardly into the weighted version.

Additionally, our construction relies on the consensus sub-protocol $\pi_{consensus}$ to validate a transaction by the elected committee. Specifically, the collective stake pool protocol is parameterized by: i) the validation predicate Validate, ii) the permutation algorithm $a_{perm}$, and iii) a consensus sub-protocol $\pi_{consensus}$.

Our (modular) protocol is described in a hybrid world with auxiliary functionalities for established primitives. The functionality $\mathcal{F}_{BC}$ [HZ10] provides a reliable broadcast channel to all parties; $\mathcal{F}_{corewallet}$ (cf. Chapter 4) enables delegation to the pool; $\mathcal{F}_{wtss}$ (cf. Section 5.3) enables weighted threshold signature operations; the Smart Contract Functionality $\Gamma_{reward}$ realizes the reward distribution mechanism; $\overline{\mathcal{G}}_{simpleLedger}$ is a global Ledger Functionality [KKK21b]. Finally, we use $\mathsf{HYBRID}^{pool}_{\pi_{pool},\mathcal{A},\mathcal{Z}}$ to denote the $\{\overline{\mathcal{G}}_{simpleLedger}, \mathcal{F}_{BC}, \mathcal{F}_{corewallet}, \mathcal{F}_{wtss}, \Gamma_{reward}\}$-hybrid execution of $\pi_{pool}$ in the (global) UC Framework.

### 5.4.2 Part 1: Stake Pool Management

The functionality's first part (Figure 5.4) includes all operations that are not consensus-oriented. First, establishing a stake pool consists of two parts, defined as corresponding interfaces in the ideal functionality. The pool's members gather and jointly decide to create a staking pool; they contact each other, e.g., via off-chain direct channels, agree on the pool's parameters, and generate its key. Importantly, the participants are aware of the total number of participants in the pool, as well as their weights. Then, the mem-

bers of the pool perform a setup protocol and register the new pool via a registration certificate, which is signed by the pool's key and published on the ledger. Following, the pool receives rewards for participating in the consensus protocol. The rewards are managed by a smart contract and, at any point, each each party can withdraw their part, which is proportional to the internal stake distribution. Finally, to close the pool, the members sign and publish a revocation certificate.

In more detail, the functionality $\mathcal{F}_{pool}$ interacts with $n$ parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and is parameterized by:

- the validation predicate Validate$(\cdot, \cdot)$ which, given a transaction $\tau$ and a chain $\mathcal{C}$, defines whether $\tau$ can be appended to $\mathcal{C}$ (as part of a block);
- the algorithm blockify which, given a set of transactions, serializes them (deterministically) in a block;
- the probability $\Pi^{\theta,t,n}$ that the elected committee, responsible for a transaction's verification, is corrupted, dependent on the subselection parameter $\theta$ and the number of corrupted parties $t$ out of $n$ total parties.

It also keeps the following, initially empty, variables: i) the signature threshold $T$; ii) the public key $\mathsf{vk}_{pool}$; iii) the reward address $\alpha_{reward}$; iv) the set of valid and unpublished transactions mempool; v) a mapping of parties to weights $W$; vi) a table of signatures sigs.

**Gathering and Registration.** The first step in creating a pool is the gathering of parties, in order to collectively create the pool's public key $\mathsf{vk}_{pool}$. Following, the parties create and publish on the ledger the registration certificate $\mathsf{cert}_{reg}$, which contains the following:

- $\omega$: a mapping identifying each member's weight;
- $\alpha_{reward}$: the address which accumulates the pool's rewards;
- $\mathsf{vk}_{pool}$: the pool's threshold public key;
- $\sigma_{pool}$: the signature of $\langle \omega, \alpha_{reward} \rangle$ created by $\mathsf{vk}_{pool}$.

**Reward Withdrawal.** During the life cycle of the pool, a member may want to withdraw the rewards received up to that point. As per the desiderata of Section 5.1, any party should be able to do so, without the explicit permission of the other pool's members. Additionally, the rewards that each party receives should be proportional to

its stake, i.e., its weight within the collective pool. Reward withdrawal is implemented as the smart contract functionality $\Gamma_{reward}$. The contract is initialized with the weight distribution of the pool's members and each member's public key. We assume that the contract is associated with an address and can receive funds, similar to real-world smart contract systems like Ethereum [Woo14]. The state transition functionality $\Gamma_{reward}$ is defined in Figure 5.3 (following the ledger formalization of Section 5.2.2).

Reward Smart Contract Functionality $\Gamma_{reward}$

$\Gamma_{reward}$ maintains a mapping $\omega$, of parties to weights, and a variable $b$.

**Initialization:** Upon receiving $(\text{init}, \text{sid}, \omega')$, forward it to $\mathcal{S}$. Upon receiving a response $(\text{init-ok}, \text{sid}, \alpha_{sc})$, set $\omega \leftarrow \omega'$ and return $(\text{init-ok}, \text{sid}, \alpha_{sc})$.

**Balance Update:** On receiving $(\text{transaction}, \text{sid}, \tau)$ from $\mathcal{U}$, such that $\tau = \langle \alpha_s, \alpha_r, v, f \rangle$, if $\alpha_s = \alpha_{sc}$ set $b := b - v$, else if $\alpha_r = \alpha_{sc}$ set $b := b + v$.

**Withdrawal:** Upon receiving $(\text{withdraw}, \text{sid}, \alpha, f)$ from the party $\mathcal{P}$, set $r = \frac{w_p}{\sum_{p' \in \omega} w_{p'}} \cdot b$ and return $(\text{transaction}, \text{sid}, \langle \alpha_{sc}, \alpha, r, f \rangle)$.

Figure 5.3: The pool's Reward Smart Contract Functionality.

**Closing.** Eventually, the members halt the operation of the pool. In order to do so, they revoke the pool's registration by jointly producing a revocation certificate $\text{cert}_{rev}$. The certificate is relatively simple, containing a timestamp $x$ announcing the end of the pool and signed by the pool's public key $\text{vk}_{pool}$.

The first part of our functionality definition is given by Figure 5.4, whereas the management routines, i.e., the first part of the description, of our protocol construction is given by Figure 5.5.

### 5.4.3 Part 2: Participation in Consensus

After a pool is set up, the functionality's second part (Figure 5.7) considers participation in the system, i.e., *validating transactions* and *issuing blocks*. The pool members continuously monitor the network for new transactions, which they collect, validate, and organize in a *mempool*. As mentioned in the introduction, the pool members *remain online* for the entirety of the execution to perform the pool's operations. Specifically, when the pool is elected to participate, the mempool's transactions are serialized and

Collective Pool Functionality $\mathcal{F}_{pool}^{T,\omega}$ (first part)

**Gathering:** Upon receiving $(\text{gather}, \text{sid})$ from $\mathcal{P}$, forward it to $\mathcal{S}$. After every party $\mathcal{P}_i, i \in [1,n]$ has submitted gather, upon receiving from $\mathcal{S}$ $(\text{gather-ok}, \text{sid}, \mathsf{vk}_{pool})$, store $T$ and $\mathsf{vk}_{pool}$, add all party-weight pairs $(\mathcal{P}_i, \omega_i)$ to $W$, and reply with $(\text{gather-ok}, \text{sid}, \mathsf{vk}_{pool})$ to all parties.

**Pool Registration:** Upon receiving $(\text{register}, \text{sid}, W)$ from $\mathcal{P}$, forward it to $\mathcal{S}$. After all parties $\mathcal{P}_i, i \in [1,n]$ have submitted register, upon receiving from $\mathcal{S}$ $(\text{register-ok}, \text{sid}, \alpha_{reward}, \sigma_{pool})$, set $\text{cert}_{reg} = \langle (W, \alpha_{reward}, \mathsf{vk}_{pool}, \sigma_{pool}) \rangle$. Then check if $\forall (m, \sigma, b') \in \text{sigs} : \sigma \neq \sigma_{pool}, (\text{cert}_{reg}, \sigma_{pool}, 0) \notin \text{sigs}$; if the checks hold, insert $(\text{cert}_{reg}, \sigma_{pool}, 1)$ to sigs. Finally, store $\alpha_{reward}$ and reply with $(\text{register-ok}, \text{sid}, \text{cert}_{reg})$.

**Reward Withdrawal:** Upon receiving the message $(\text{withdraw}, \text{sid}, \alpha, f)$ from $\mathcal{P}_i$, forward it to $\mathcal{S}$. Then, compute $R = \frac{w_{\mathcal{P}_i}}{\sum_{j=1}^{n} w_{\mathcal{P}_j}} \cdot R_{pool}$, where $R_{pool}$ is the funds of address $\alpha_{sc}$ as defined in $\overline{\mathcal{G}}_{simpleLedger}$. Finally, return $(\text{transaction}, \text{sid}, \langle \alpha_{sc}, \alpha, R, f \rangle)$.

**Closing:** Upon receiving $(\text{close}, \text{sid}, x)$ from $\mathcal{P}$, forward it to $\mathcal{S}$. After a set of parties $B$ has submitted close for the same $x$, if $\sum_{\mathcal{P} \in B} w_{\mathcal{P}} > T$, upon receiving $(\text{close-ok}, \text{sid}, \sigma_{pool})$ from $\mathcal{S}$, check if $\forall (m, \sigma, b') \in \text{sigs} : \sigma \neq \sigma_{pool}, (x, \sigma_{pool}, 0) \notin \text{sigs}$; if the checks hold, insert $(x, \sigma_{pool}, 1)$ to sigs. Finally, return to all parties $(\text{close-ok}, \text{sid}, \text{cert}_{rev})$, with $\text{cert}_{rev} = \langle x, \sigma_{pool} \rangle$.

Figure 5.4: The first part of the Collective Pool Functionality, parameterized with threshold $T$ and weight mapping $\omega$, refers to the creation and management of the pool (the second part is given by Figure 5.7).

Collective Pool Protocol $\pi_{pool}^{T,\omega}$ (first part)

**Gathering:** Upon receiving $(\mathsf{gather}, \mathsf{sid})$, send $(\mathsf{KeyGen}, \mathsf{sid})$ to $\mathcal{F}_{wtss}$, with $sid$ containing the weight mapping $\omega$ and the threshold $T$. Upon receiving the reply $(\mathsf{KeyGen}, \mathsf{sid}, \mathsf{vk}_{pool})$, return $(\mathsf{gather\text{-}ok}, \mathsf{sid}, \mathsf{vk}_{pool})$.

**Pool Registration:** Upon receiving $(\mathsf{register}, \mathsf{sid}, W)$, send $(\mathsf{init}, \mathsf{sid}, W)$ to $\Gamma_{reward}$ and wait for the reply $(\mathsf{init\text{-}ok}, \mathsf{sid}, \alpha_{reward})$. Then, set $m = (W, \alpha_{reward})$ and send $(\mathsf{Sign}, \mathsf{sid}, m)$ to $\mathcal{F}_{wtss}$. Upon receiving a reply $(\mathsf{Sign}, \mathsf{sid}, m, \sigma_{pool})$, return $(\mathsf{register\text{-}ok}, \mathsf{sid}, \mathsf{cert}_{reg})$, where $\mathsf{cert}_{reg} = \langle (W, \alpha_{reward}, \mathsf{vk}_{pool}, \sigma_{pool}) \rangle$.

**Reward Withdrawal:** Upon receiving $(\mathsf{withdraw}, \mathsf{sid}, \alpha, f)$, forward it to $\Gamma_{reward}$. Upon receiving a response $(\mathsf{transaction}, \mathsf{sid}, \langle \alpha_{sc}, \alpha, r, f \rangle)$ return it.

**Closing:** Upon receiving $(\mathsf{close}, \mathsf{sid}, x)$, send $(\mathsf{Sign}, \mathsf{sid}, x)$ to $\mathcal{F}_{wtss}$. Upon receiving a reply $(\mathsf{Sign}, \mathsf{sid}, x, \sigma_{pool})$, return $(\mathsf{close\text{-}ok}, \mathsf{sid}, \mathsf{cert}_{rev})$ with $\mathsf{cert}_{rev} = \langle x, \sigma_{pool} \rangle$.

Figure 5.5: The first part of the Collective Pool Protocol, which describes the set of management operations (the second part is given by Figure 5.8).

published in a block. Under PoS, the pool participates proportionally to its aggregated member and delegated stake.

To improve performance, we define a distributed mechanism for transaction verification, i.e., a *distributed mempool*. Such load balancing mechanism increases efficiency by requiring only a subset of the pool's members to verify each transaction. Notably, this is in contrast to the standard practice of Bitcoin mining pools, where the pool's operator decides the transactions to be mined by its members; instead, our approach further reduces these trust requirements.

To construct a distributed mempool, we consider a subselection mechanism to identify the parties that verify each transaction. This mechanism should be: a) *non-interactive* b) *deterministic*, c) *balanced*, i.e., every party should be chosen with the same probability. Subselection is secure if a majority of the elected committee is honest. However, since the adversary may corrupt some pool members, this may not always be the case. We model this uncertainty via the probability $\Pi^{\theta,t,n}$, which depends on the size of the committee and the power of the adversary among the pool's members.

A straightforward way to implement subselection is to assume that the pool's members are ordered in a well-defined manner, e.g., lexicographically. Given the ordered list $L = [\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_n]$ of pool members, we use a permutation algorithm $a_{perm}(\cdot,\cdot,\cdot)$, which takes two arguments, i) a transaction $\tau$, ii) a chain $\mathcal{C}$, and iii) the ordered list of pool members $L$, and outputs a pseudorandom permuted list $L_\tau$. For every transaction $\tau$ and a given chain $\mathcal{C}$, the committee responsible for verification consists of the $\theta$ first members in $L_\tau$. Naturally, this proposal is rather simple, so alternative, e.g., VRF-based, mechanisms could be proposed to improve performance.

We note that using $\mathcal{C}$ during the subselection mechanism is important to avoid adaptive attacks. Specifically, the chain $\mathcal{C}$ simulates a randomness beacon, such that at least one of its last $u$ blocks is honest, for some parameter $u$. If $\mathcal{C}$ was not used, the adversary could construct a malicious transaction in such way that the subselected committee would also be malicious. By using $\mathcal{C}$ as a seed to the pseudorandom permutation, the adversary's ability to construct such malicious transaction is limited. Alternatively, cryptographic sortition [GHM+17a] could be employed to fully handle adaptive adversaries.

The (honest) members need to always have the same view of the distributed mempool; this is achieved via authenticated broadcast. Assuming a Public Key Infrastracture (PKI), as is our setting, it is possible to achieve deterministic authenticated broadcast in $t+1$ rounds for $t$ adversarial parties [LSP82, PSL80a, DS83]. Each time a party adds a transaction to its mempool, it broadcasts it, such that, at any point in time, the honest

members of the pool have the same view of the network w.r.t. the canonical chain and the mempool of unconfirmed transactions. We remind that, as shown by Garay *et al.* [GKKZ11], $\mathcal{F}_{BC}$ can be implemented to ensure adaptive corruptions using commitments. We note that, in existing distributed ledgers, the order with which transactions are added to the mempool does not affect the choice when creating a new block; for instance, transactions of a new block are typically chosen based on a fee-per-byte score. If the order of transactions is pertinent, a stronger primitive like Atomic Broadcast [DSU04] could be employed.

Following, the committee employs a consensus sub-protocol to agree on the transaction's validity. When a party $\mathcal{P}$ retrieves a new transaction $\tau$ from the network, it broadcasts it as above. Then, each party computes the permuted list $L_\tau$. Each party, which is in the validation committee for $\tau$, computes locally the validation predicate and submits its output to the consensus protocol. The consensus protocol should offer *strong validity*, i.e., if all honest parties should have the same input bit, they should output this bit. Finally, the output of the consensus protocol is broadcast to the rest of the pool. To verify the committee's actions, a party may request the transcript of the consensus sub-protocol.

Finally, to compute the probability of electing an honest committee, we have a hypergeometric distribution, with population size $n$ and $n-t$ honest parties, where a sample of parties of size $\theta$ is chosen *without replacement*. Thus, the probability of honest committee majority is: $\Pi^{\theta,t,n} = 1 - \sum_{v=\lfloor \frac{\theta+1}{2} \rfloor}^{\min(\theta,t)} \frac{\binom{t}{v} \cdot \binom{n-t}{\theta-v}}{\binom{n}{\theta}}$. Figure 5.6 provides further intuition on the probability w.r.t. the subselection parameter $\theta$.

Following, Figure 5.7 defines the second part of our functionality, while Figure 5.8 presents the second part of our protocol.

We note that our design satisfies most of the desiderata outlined in Section 5.1. Some (e.g., pool proportional rewards or stake reallocation) are dependent on the underlying ledger system's details, therefore are outside of our scope; nevertheless, our design does not pose restrictions in capturing them. The reward functionality $\Gamma_{reward}$ handles the reward-specific desiderata, while $\mathcal{F}_{pool}$'s first part (Figure 5.4) covers the requirements for permissioned access and closing of the pool. However, $\mathcal{F}_{pool}$'s handling of stake reallocation and updating of the pool's parameters could be more dynamic, as it currently requires closing and re-creating a pool with the new parameters.

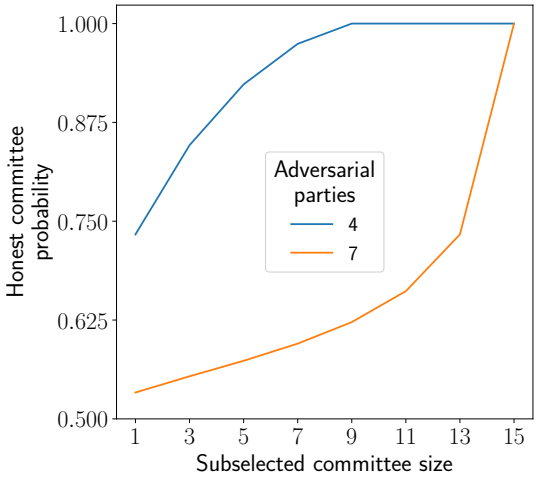


Figure 5.6: The probability of subselecting an honest committee w.r.t. the committee size $\theta$, $n = 15$ total parties and $\lfloor \frac{n-1}{3} \rfloor$ and $\lfloor \frac{n-1}{2} \rfloor$ adversarial parties.

## 5.5 Security Analysis

We now assess the security of our collective pool design. Specifically, Theorem 5 shows that the protocol $\pi_{pool}$ securely realizes the ideal functionality $\mathcal{F}_{pool}$, assuming that the employed cryptographic primitives are secure and that the adversarial power within the pool is properly bounded.

**Theorem 5.** *The protocol $\pi_{pool}$, parameterized by a validation predicate* Validate*, a permutation algorithm $a_{perm}$, and a consensus protocol $\pi_{consensus}$ (cf. Definition 4) securely realizes $\mathcal{F}_{pool}$ with the hybrid execution* $\text{HYBRID}^{pool}_{\pi_{pool},\mathcal{A},\mathcal{Z}}$ *in the global $\overline{\mathcal{G}}_{simpleLedger}$ model, and $\Pi^{\theta,t,n} = 1 - \sum_{v=\lfloor \frac{\theta+1}{2} \rfloor}^{\min(\theta,t)} \frac{\binom{t}{v} \cdot \binom{n-t}{\theta-v}}{\binom{n}{\theta}}$, assuming $\sum_{\mathcal{P} \in P_{\mathcal{A}}} w_{\mathcal{P}} < T$, where $\theta$ is the subselection parameter for transaction verification, $P_{\mathcal{A}}$ is the set of $t$ corrupted parties out of $n$ total parties, $\omega$ is the weight distribution of the $n$ parties, and $T$ is the signature threshold.*

*Proof.* The proof is constructed in the UC Framework, so it is simulation-based. As such, we will show that the environment $\mathcal{Z}$ cannot efficiently distinguish between two executions, the ideal and the real. The simulator $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}_{pool}$ in the ideal execution, whereas $\mathcal{A}$ interacts with $\pi_{pool}$ in the real execution. We will show that, if $\pi_{pool}$ does not securely realize the ideal functionality $\mathcal{F}_{pool}$, when instantiated with the parameters defined in the theorem, then at least one of the conditions is violated.

First, we provide the construction for the simulator. $\mathcal{S}$ runs internally a copy of the adversary $\mathcal{A}$. $\mathcal{S}$ forwards any inputs received from the environment $\mathcal{Z}$ to the internal

Collective Pool Functionality $\mathcal{F}_{pool}^{T,\omega}$ (second part)

**Transaction Verification:** Upon receiving $(\text{transaction}, \text{sid}, \tau, \theta)$ from $\mathcal{P}_i$, forward it to $\mathcal{S}$. Then send READ to $\overline{\mathcal{G}}_{simpleLedger}$ on behalf of $\mathcal{P}_i$ and wait for the reply $\mathcal{C}$. Following, set $t$ as the number of corrupted parties; with probability $\Pi^{\theta,t,n}$ set $b :=$ Validate$(\tau, \mathcal{C})$, otherwise (with probability $1-\Pi^{\theta,t,n}$), send $(\text{transaction-ver}, \text{sid}, \tau)$ to $\mathcal{S}$, wait for a reply $(\text{transaction-ok}, \text{sid}, \mathcal{C}, \tau, f)$, and set $b := f$. Finally, if $b = 1$, insert $\tau$ to mempool and send $(\text{transaction}, \text{sid}, \mathcal{C}, \tau, b)$ to all parties.

**Mempool Update:** Upon receiving $(\text{transaction}, \text{sid}, \mathcal{C}', \tau, 1)$ from $\mathcal{P}_i$, forward it to $\mathcal{S}$. Then send READ to $\overline{\mathcal{G}}_{simpleLedger}$ on behalf of $\mathcal{P}_i$ and wait for the reply $\mathcal{C}$. If $\mathcal{C}' \prec \mathcal{C}$ and $\mathcal{P}_i$ is honest, insert $\tau$ to mempool and return $(\text{mempool-updated}, \text{sid}, \tau)$.

**Block issuing:** Upon receiving $(\text{issue-block}, \text{sid})$ from a party $\mathcal{P}$, forward it to $\mathcal{S}$. When a set of parties $\mathbb{P}$ has submitted $(\text{issue-block}, \text{sid})$, if $\sum_{j\in[1,m]} W[\mathcal{P}_j] > T$, then for every party $\mathcal{P}_i \in \mathbb{P}$, send READ to $\overline{\mathcal{G}}_{simpleLedger}$ on behalf of $\mathcal{P}_i$ and wait for the reply $\mathcal{C}_i$. If all received chains equal, i.e., are the same chain $\mathcal{C}$, remove every $\tau$ in mempool that also exists in $\mathcal{C}$. Then, set $b =$ blockify$(\text{mempool})$, send $(\text{issue-block}, \text{sid}, b)$ to $\mathcal{S}$, and wait for the reply $(\text{issue-block}, \text{sid}, b, \sigma_{pool})$. Following, check if $\forall (m, \sigma, b') \in T : \sigma \neq \sigma_{pool}, (b, \sigma_{pool}, 0) \notin T$; if the checks hold, insert $(b, \sigma_{pool}, 1)$ to $T$. Finally, reply with $(\text{block}, \text{sid}, b, \sigma_{pool})$.

Figure 5.7: The second part of the proposed Pool Functionality, which defines the consensus participation operations.

Collective Pool Protocol $\pi_{pool}^{T,\omega}$ (second part)

**Transaction Verification:** Upon receiving $(\text{transaction}, \text{sid}, \tau, \theta)$, send READ to $\overline{\mathcal{G}}_{simpleLedger}$ and wait for the reply $\mathcal{C}$. Then, set $b = \text{Validate}(\mathcal{C}, \tau)$, compute $L' = a_{perm}(\tau, \mathcal{C}, L)$ and initiate protocol $\pi_{consensus}$ with the $\theta$ first parties in $L'$ with input $b$. Upon computing the output of $\pi_{consensus}$, $\beta$, send $(\text{transaction}, \text{sid}, \mathcal{C}, \tau, \beta)$ to $\mathcal{F}_{BC}$ and return it.

**Mempool Update:** Upon receiving $(\text{transaction}, \text{sid}, \mathcal{C}', \tau, 1)$, $\mathcal{P}_i$, send READ to $\overline{\mathcal{G}}_{simpleLedger}$ and wait for the reply $\mathcal{C}$. If $\mathcal{C}' \prec \mathcal{C}$, insert $\tau$ to mempool and return $(\text{mempool-updated}, \text{sid}, \tau)$.

**Block Issuing:** Upon receiving $(\text{issue-block}, \text{sid})$, send READ to $\overline{\mathcal{G}}_{simpleLedger}$ and wait for the reply $\mathcal{C}$. For every $\tau$ in mempool, if $\tau$ is also in $\mathcal{C}$, then remove $\tau$ from mempool. Next, set $b = \text{blockify}(\text{mempool})$ and send $(\text{Sign}, \text{sid}, b)$ to $\mathcal{F}_{wtss}$. Upon receiving a reply $(\text{Sign}, \text{sid}, b, \sigma_{pool})$, return $(\text{block}, \text{sid}, b, \sigma_{pool})$.

Figure 5.8: The second part of our protocol, which describes the set of operations for consensus participation.

copy of $\mathcal{A}$, and vice versa, and behaves as follows:

- *Gathering*: Upon receiving the message $(\text{gather}, \text{sid})$ for all parties $\mathcal{P}_i, i \in [1, n]$ from $\mathcal{F}_{pool}$, send $(\text{KeyGen}, \text{sid})$ to $\mathcal{F}_{wtss}$ with the appropriate $sid$. Upon receiving the reply $(\text{KeyGen}, \text{sid}, \text{vk}_{pool})$, record vk, and return $(\text{gather-ok}, \text{sid}, \text{vk})$ to $\mathcal{F}_{pool}$.

- *Pool Registration*: Upon receiving the $n$ messages $(\text{register}, \text{sid}, \text{members})$ from $\mathcal{F}_{pool}$, send $(\text{init}, \text{sid}, \text{members})$ to $\Gamma_{reward}$ and wait for $(\text{init-ok}, \text{sid}, \alpha_{reward})$. Then send $(\text{Sign}, \text{sid}, m)$ to $\mathcal{F}_{wtss}$ with $m = (\text{members}, \alpha_{reward})$. Upon receiving a reply $(\text{Sign}, \text{sid}, m, \sigma_{pool})$, register $(m, \sigma_{pool})$ and return to $\mathcal{F}_{pool}$ the message $(\text{register-ok}, \text{sid}, \alpha_{reward}, \sigma_{pool})$.

- *Closing*: Upon receiving $(\text{close}, \text{sid}, x)$ from $\mathcal{F}_{pool}$, on behalf of a set of parties $\mathbb{P}$, send the message $(\text{Sign}, \text{sid}, x)$ to $\mathcal{F}_{wtss}$ on behalf of each party in $\mathbb{P}$. Upon receiving a reply $(\text{Sign}, \text{sid}, x, \sigma_{pool})$, record $\sigma_{pool}$ and return $(\text{close-ok}, \text{sid}, \sigma_{pool})$ to $\mathcal{F}_{pool}$.

- *Transaction Verification*: Upon receiving $(\text{transaction-ver}, \text{sid}, \tau)$ from $\mathcal{F}_{pool}$, for-

ward it to the internal copy of $\mathcal{A}$, wait for the output $(\text{transaction-ok}, \text{sid}, \mathcal{C}, \tau, f)$ from $\mathcal{A}$ and forward it to $\mathcal{F}_{pool}$.

- *Block issuing*: Upon receiving $(\text{issue-block}, \text{sid}, b)$ from $\mathcal{F}_{pool}$, send $(\text{Sign}, \text{sid}, b)$ to $\mathcal{F}_{wtss}$. Upon receiving a reply $(\text{Sign}, \text{sid}, b, \sigma_{pool})$ and record $(b, \sigma_{pool})$. Finally, return $(\text{issue-block}, \text{sid}, b, \sigma_{pool})$ to $\mathcal{F}_{pool}$.

- *Party corruption*: When the adversary $\mathcal{A}$ corrupts a party $\mathcal{P}$, $\mathcal{S}$ corrupts the same party in the ideal process and hands to $\mathcal{A}$ its internal state.

- *Global ledger update*: When $\mathcal{A}$ sends $(\text{ADVANCE}, \mathcal{P}, \Sigma)$ to the global ledger $\overline{\mathcal{G}}_{simpleLedger}$, $\mathcal{S}$ does so in the ideal world; similarly, when $\mathcal{A}$ sends $(\text{EXTEND}, b)$ to $\overline{\mathcal{G}}_{simpleLedger}$, so does $\mathcal{S}$.

- *Signature generation*: When the adversary $\mathcal{A}$ requests a signature on some message $m$, $\mathcal{S}$ sends $(\text{Sign}, \text{sid}, m)$ to $\mathcal{F}_{wtss}$; upon receiving the reply $(\text{Sign}, \text{sid}, m, \sigma)$, it returns $\sigma$ to $\mathcal{A}$.

The first observation is that $\mathcal{S}$ needs to ensure that a party $\mathcal{P}$ has the same view of the ledger as in the real world. Therefore, it advances parties only when the real world adversary $\mathcal{A}$ does so.

To prove the theorem, we assume that $\pi_{pool}$ does not realize $\mathcal{F}_{pool}$, i.e., there exists adversary $\mathcal{A}$ such that, for every simulator $\mathcal{S}$, there exists environment $\mathcal{Z}$ that can distinguish between the ideal world (of $\mathcal{F}_{pool}$ and $\mathcal{S}$) and the real world (of $\pi_{pool}$ and $\mathcal{A}$). Following, we show that $\mathcal{S}$ violates the security of one of the primitives used by $\pi_{pool}$, i.e., the consensus protocol $\pi_{consensus}$ and the weighted threshold signature scheme $\Sigma_{\text{thresh}}$.

We build an algorithm $\mathcal{D}$ that breaks the security of the cryptographic primitives as follows. $\mathcal{D}$ runs a simulated copy of $\mathcal{Z}$ and simulates for $\mathcal{Z}$ the ideal environment, i.e., $\mathcal{D}$ acts both as $\mathcal{F}_{pool}$ and $\mathcal{S}$ on $\mathcal{Z}$'s messages.

Similar to $\mathcal{S}$, $\mathcal{D}$ runs a simulated copy of $\mathcal{A}$. When running *Gathering* to obtain the threshold keys, instead of running ThreshKeyGen, $\mathcal{D}$ hands $\mathcal{A}$ the public key vk which is obtained as the input from $\mathcal{S}$. To obtain a signature $\sigma$ on a message $m$, $\mathcal{D}$ hands $m$ to its oracle, instead of using ThreshSign. When $\mathcal{A}$ advances the state of party $\mathcal{P}$ in the global ledger $\overline{\mathcal{G}}_{simpleLedger}$, $\mathcal{D}$ does so as well.

Regarding the consensus subprotocol $\pi_{consensus}$, we consider the case when $\mathcal{Z}$ activates an uncorrupted party $\mathcal{P}$ with input a transaction $\tau$ via the interface *Transaction Verification*. At that point, $\mathcal{D}$ computes $b = \text{Validate}(\mathcal{C}, \tau)$, where $\mathcal{C}$ is the state of party

$\mathcal{P}$ in the global ledger $\overline{\mathcal{G}}_{simpleLedger}$. Next, $\mathcal{D}$ checks the output $b'$ in the real world (where $\mathcal{A}$ operates). If the majority of the committee elected to validate $\tau$ is honest and $b \neq b'$, then $\mathcal{D}$ retrieves the transcript of $\pi_{consensus}$, run for the validation of $\tau$ by $\mathcal{A}$, and outputs it (observe that this transcript is represents an execution of $\pi_{consensus}$ where its security breaks).

To analyze the success probability of $\mathcal{D}$, we consider the event $E$, where $b \neq b'$, as defined above. Since $\pi_{consensus}$ is secure as long as a majority of participants is honest, the executions of the real world, i.e., the interaction of $\mathcal{Z}$ with $\mathcal{A}$ and $\pi_{pool}$, and the ideal world (resp. $\mathcal{S}$ and $\mathcal{F}_{pool}$) are statistically close. If we are guaranteed that $\mathcal{Z}$ distinguishes between the two executions, then $E$ occurs with non-negligible probability. Finally, from the point of view of $\mathcal{A}$ and $\mathcal{Z}$, the interaction with $\mathcal{D}$ is the same as with an interaction with protocol $\pi_{pool}$ in the real world.

Regarding the weighted threshold signatures, we note that the functionality $\mathcal{F}_{pool}$ performs the same checks regarding signature issuing as $\mathcal{F}_{wtss}$. In fact, only signature generation is performed by the collective pool; signature verification should be employed when advancing the ledger state, i.e., upon adopting new blocks, or when validating a certificate. Therefore, the security of $\mathcal{F}_{wtss}$ ensures that $\mathcal{Z}$ cannot distinguish the two executions (real vs. ideal world) w.r.t. the weighted threshold signatures.

Regarding block issuing we consider the event $E'$, where a set of parties controlling a majority of the pool's stake initiate block issuing, but no signed block is output. In that case, either the signature issuing of $\Sigma_{\mathsf{thresh}}$ fails or the parties locally produce a different block $b$, i.e., their mempool is not synchronized. Regarding the former, the same analysis on signature issuing as above applies. Regarding the latter, if two honest parties hold a different mempool at the point when blockify is used, either their ledger state $\mathcal{C}$ is different or their mempool is different. This implies that $\mathcal{F}_{BC}$ fails for at least one transaction $\tau$, i.e., an honest party inserts $\tau$ in its mempool and, after $\tau$ is sent to $\mathcal{F}_{BC}$, at least one other honest party fails to also insert it to its mempool. However, this is impossible, since the simulator ensures the former and $\mathcal{F}_{BC}$ ensures the latter.

Finally, the permutation algorithm $a_{perm}$ is executed locally by each party, therefore the adversary cannot affect its output. Additionally, the probability $\Pi^{\theta,t,n}$ is computed following the analysis of Section 5.4.3. □

## 5.6 Incentives Analysis

Although, as shown in Theorem 5, $\pi_{pool}$ is secure, it is unclear whether rational users will opt for using it. In this section, we discuss the incentive compatibility of $\pi_{pool}$. We identify its shortcomings and propose a minor change, such that rational members cannot gain more rewards by deviating from it.

First, we consider the cost of each operation performed in $\pi_{pool}$. Signing operations does not incur any cost, thus pool registration and revocation are cost-free. Block production depends on the internal workings of blockify. For instance, solving the Knapsack problem can be expensive, while a greedy algorithm that prioritizes high-fee transactions is typically not. Therefore, without loss of generality, we also assume that block production is cost-free. However, both mempool update and transaction verification incur costs $C_{mu}$ and $C_{tv}$ respectively. A mempool consists of millions of transactions and verifying them requires an accurate view of the ledger. Thus, both objects may require significant amounts of computations and storage.[2]

We focus on the *profit* of each member, i.e., the rewards subtracted by the cost of executing $\pi_{pool}$. The core observation is that a member $\mathcal{P}$ receives $R_{\mathcal{P}} = \frac{w_{\mathcal{P}}}{\sum_{\mathcal{P}' \in \omega} w_{\mathcal{P}'}}$ of the total pool's rewards *regardless of its performance*. For instance, if $\mathcal{P}$ acts only on the pool's creation, it still receives its proportional share of rewards for the blocks produced by the rest of the pool. Therefore, as long as a member believes that the other members act honestly, it is incentivized to abstain and minimize its cost, thus maximizing its profit. Naturally, if all parties follow this strategy, the pool produces no blocks and receives no rewards.

A possible solution to the Free Rider problem above is to penalize a party for misbehaving. However, identifying misbehavior is not straightforward. For instance, a party $\mathcal{P}$ who inputs $0$ to $\pi_{consensus}$ for a transaction $\tau$ may do so either because the transaction is invalid or because it didn't perform validation and input $0$ by default. Our approach is to penalize a party when diverging from the rest of the pool. In the previous example, if the output of $\pi_{consensus}$ is $1$, then $\mathcal{P}$ incurs a fixed penalty $P_{tv}$. Similarly, if a party fails to sign a new block then it incurs a (fixed) penalty $P_{mu}$. The penalty amount, which is withheld from $\mathcal{P}$, is then distributed equally among the other pool members. To incentivize $\mathcal{P}$ to follow $\pi_{pool}$ the penalties should be high enough; specifically, it should hold $P_{tv} > C_{tv}$ and $P_{mu} > C_{mu}$. If all parties follow $\pi_{pool}$, diverting incurs a

[2] As of January 2021, the Bitcoin chain is roughly $320$GB and increases linearly over time. (`https://www.blockchain.com/charts/blocks-size`)

cost $P_{mu} - C_{mu} > 0$ (resp. $P_{tv} - C_{tv} > 0$), thus the new protocol is an equilibrium.

Finally, penalties can be automatically enforced via an interface to the smart contract $\Gamma_{reward}$ which, given a proof of misbehavior, reduces the misbehaving party's rewards accordingly. For transaction verification, a proof of misbehavior is the transcript of $\pi_{consensus}$, which describes the consensus sub-protocol's execution. For block issuing, we can use a threshold signature scheme with identifiable aborts [GG20, CGG$^+$20], which allows to identify the parties that do not participate in the signing of a block.

# Chapter 6

# Efficient Global State Management

Distributed ledgers implement a storage layer, on top of which a shared state is maintained in a decentralized manner. In ledger systems, this state consists of three objects: i) the public ledger, i.e., the list of transactions which form the system's history; ii) the mempool, i.e., the set of, yet unpublished, transactions; iii) the active state which, in systems like Bitcoin, consists of the UTxO set. To support thousands (or millions) of participants, a decentralized system's state management should be well-designed, primarily focused on minimizing the shared state. Our work focuses on the third type, as poorly designed management often leads to performance issues and even Denial-of-Service (DoS) attacks. In Ethereum, during a 2016 DoS attack, an attacker added 18 million accounts to the state, increasing its size by 18 times [Wil16]. Bitcoin saw similar spam attacks in 2013 [VTM14] and 2015 [Bit15], when millions of outputs were added to the UTxO set.

Mining nodes and full nodes incur costs for maintaining the shared state in the Bitcoin network. This cost pertains to the resources (i.e., CPU, disk, network bandwidth, memory) that are consumed with every transaction transmitted, validated, and stored. An expensive part of a transaction is the newly created outputs, which are added to the in-memory UTxO set. As the system's scale increases, the cost of maintaining the UTxO set gradually leads to a shared-state bloat, which makes the cost of running a full node prohibiting.

Moreover, the system's incentives, which are promoted via transaction fees, only deteriorate the problem. For example, assume two transactions $\tau_A$ and $\tau_B$: $\tau_A$ spends $5$ inputs and creates $1$ output, while $\tau_B$ spends $1$ input and creates $2$ outputs. Assuming the size of a UTxO is equal to the size of consuming it (200 bytes) and that transaction fees are $30$ satoshi per byte, $\tau_A$ costs $30 \times 200 \times (5+1) = 36000$ satoshi and $\tau_B$ costs

$30 \times 200 \times (1+2) = 18000$ satoshi. Although $\tau_B$ burdens the UTxO set by creating a net delta of $(2-1=1)$ new UTxO, while $\tau_A$ reduces the shared state by consuming $(1-5=-4)$ UTxOs, $\tau_B$ is cheaper in terms of fees. Clearly, the existing fee scheme penalizes the consumption of multiple inputs, disincentivizing minimizing the shared state.

**Related Work.** The problem of unsustainable growth of the UTxO set has concerned developers for years. It has been discussed in community articles [FvW19, IH19], some [And15] offering estimations on the level of inefficiency in Bitcoin. Additionally, research papers [PDNHJ18, DPNH17, Nic14, EOB19] have analyzed Bitcoin's and other cryptocurrencies' UTxO sets to gain further insight. Engineering efforts, e.g., in Bitcoin Core's newer releases [Max17], have also focused on improving performance by reducing the UTxO memory requirements. Various solutions have been proposed to reduce the state of a UTxO ledger, e.g., consolidation of outputs [Wik20] can help reduce the cost of spending multiple small outputs. Alternatively, Utreexo [Dry19], uses cryptographic accumulators to reduce the size of the UTxO set in memory, while BZIP [JLG+19] explores lossless compression of the UTxO set.

An important notion in this line of research is the "stateless blockchain" [Tod16]. Such blockchain enables a node to participate in transaction validation without storing the entire state of the blockchain, but only a short commitment to it. Chepurnoy *et al.* [CPZ18] employ accumulators and vector commitments to build such blockchain. Concurrently, Boneh *et al.* [BBF18] introduce batching techniques for accumulators in order to build a stateless blockchain with a trustless setup which requires constant amount of storage. We consider an orthogonal problem, i.e., constructing transactions in an incentive-compatible manner that minimizes the state, so these tools can act as building blocks in our proposed techniques.

The role of fees in blockchain systems has also been a topic of interest in recent years. Luu *et al.* [LTKS15] explored incentives in Ethereum, focusing on incentivizing miners to correctly verify the validity of scripts run on this "global consensus computer". Möser and Böhme [MB15] investigate Bitcoin fees empirically and observe that users' behavior depends primarily on the client software, rather than a rational cost estimation. Finally, in an interesting work, Chepurnoy *et al.* [CKM19] propose a fee structure that considers the storage, computation, and network requirements; their core idea is to classify each transaction on one of the three resource types and set its fees accordingly.

**Contributions.** Our work investigates techniques that minimize the shared state of the distributed ledger, i.e., the in-memory UTxO set. Our approach is twofold: a) we propose transaction optimization techniques which, when employed by wallets, help reduce the shared state's cost; b) propose a novel fee scheme that incentivizes "shared state-friendly" transactions.

In particular, we propose a UTxO model, which abstracts UTxO ledgers and enables evaluating the cost of a ledger's shared state. We then propose a transaction optimization framework, based on three levels of optimization: a) a declarative (rule-based) level, b) a logical/algebraic (cost-based) level, and c) a physical/algorithmic (cost-based) level. Following, we propose three transaction optimization techniques based on the aforementioned optimization levels: a) a rule-driven optimal total order of transactions (the *last-payer rule*), b) a logical transaction transformation (the *2-for-1 transformation*), and c) a novel *input selection* algorithm that minimizes the UTxO set increase, i.e., favors consumption over creation of UTxOs. We then define the transaction optimization problem and propose a 3-step dynamic programming algorithm to approximate the optimal solution. Finally, we define the state efficiency property that a fee function should have, in order to correctly reflect a transaction's shared-state cost, and propose a state efficient fee function for Bitcoin.

## 6.1 A UTxO Model

We abstract a distributed ledger as a state machine on which parties act. Specifically, we consider only *payments*, i.e., value transfers of *fungible* assets between parties.

Initially, we assume a ledger state $\mathcal{S}_{init}$, on which a *transaction* is applied to move the ledger to a new state. Transactions that may be applied on a state are *valid*, following a validation predicate. Each transaction is unique and moves the system to a unique state; with hindsight, we assume that the ledger never transitions to the same state (cf. Definition 18), i.e., valid transactions do not form cycles. Figure 6.1 provides intuition via a simple ledger model.

Our formalism is similar to chimeric ledgers [Zah18], though focused on UTxO-based ledgers. Following, we provide some basic definitions in a "top-down" approach, starting with the ledger $\mathcal{L}$, which is an ordered list of transactions; our notation of functions is the one typically used in functional programming languages, for example a function $f : A \to B \to C$ takes two input parameters of type $A$ and $B$ respectively and returns a value of type $C$.

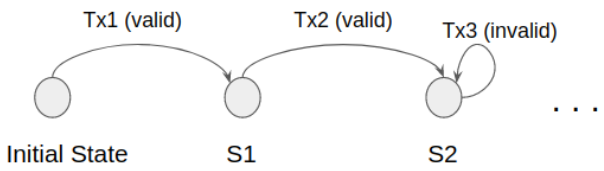


Figure 6.1: A decentralized state machine model for a distributed ledger.

**Definition 14.** *A ledger $\mathcal{L}$ is a list of valid transactions:* $\mathcal{L} \stackrel{def}{=} List[Transaction]$.

A transaction $\tau$ transitions the system from one state to another. UTxO-based transactions are thus a product of *inputs*, which define the ownership of assets, and *outputs*, which define the rules of re-transferring the acquired value.

**Definition 15.** *A UTxO-based transaction $\tau$ is :* $Transaction \stackrel{def}{=} (inputs : Set[Input], outputs : List[UTxO], forge : Value, fee : Value)$

An *unspent transaction output (UTxO)* represents the ownership of some value from a party, which is represented via an *address* $\alpha$. Intuitively, in the real world, an output is akin to owning a physical coin of an arbitrary denomination.

**Definition 16.** *A UTxO is defined as follows:* $UTxO \stackrel{def}{=} (\alpha : Address, value : Value, created : Timestamp)$.

A transaction's input is a reference to a UTxO, i.e., an output that is owned by the party that creates the transaction. An input consists of two objects: i) the *id* of the transaction that created it (typically its hash) and ii) an index, which identifies the specific output among all UTxOs of the referenced transaction.

**Definition 17.** *An input is defined as:* $Input \stackrel{def}{=} (id : Hash, index : Int)$.

Given an input and a ledger, three functions retrieve: i) the corresponding output, ii) the corresponding transaction, and iii) the input value. All returned values are wrapped in `Option`, denoting that a value may not be returned.

- $\text{UTxO} : \text{Input} \rightarrow \mathcal{L} \rightarrow \texttt{Option}[\text{UTxO}]$
- $\tau : \text{Input} \rightarrow \mathcal{L} \rightarrow \texttt{Option}[Transaction]$
- $value : \text{Input} \rightarrow \mathcal{L} \rightarrow \texttt{Option}[Value]$

A transaction defines some value that is given as a *fee* to the *miner*, i.e., the party who publishes the transaction into the ledger $\mathcal{L}$. We require that all transactions must preserve value as follows: $\tau.\mathit{forged} + \sum_{i \in \tau.\mathit{inputs}} \mathit{value}(i, \mathcal{L}) = \tau.\mathit{fee} + \sum_{o \in \tau.\mathit{outputs}} o.\mathit{value}$. We note that this applies only on standard transactions, not "coinbase" transactions which create new coins.

Finally, we define the ledger's state $\mathcal{S}$. $\mathcal{S}$ comprises the *UTxO set*, i.e., the set of all outputs of transactions whose value has not been re-transferred and can be used as inputs to new transactions.

**Definition 18.** *The ledger's state is defined as:* $\mathit{State} \stackrel{def}{=} \mathit{Set}[\mathit{Input}]$.

We now return to the state machine model. A transaction is applied on a ledger state $\mathcal{S}_1$ and results in a ledger state $\mathcal{S}_2$ via the function:

$$\text{txRun} : \mathit{Transaction} \rightarrow \mathit{LedgerState} \rightarrow \mathit{LedgerState}$$

An ordered list of transactions $\mathbb{T} = [\tau_1, \tau_2, ..., \tau_N]$ can be applied sequentially on state $\mathcal{S}_1$ to transit to state $\mathcal{S}_N$: $\mathcal{S}_N = (\text{txRun}(\tau_N).\ ...\ .\text{txRun}(\tau_2).\text{txRun}(\tau_1))(\mathcal{S}_1)$, assuming the function composition operator (.).

Finally, every ledger state $\mathcal{S}$ corresponds to some cost $C$. We assume a cost function, which assigns a signed integer of cost units to a ledger state.

$$\text{cost} : \mathit{LedgerState} \rightarrow \mathit{Cost}$$

This function is employed in Definition 19, which defines a transaction's cost; minimizing this cost will be the target of our optimization. Observe that the transaction's cost might be negative, e.g., if the transaction reduces the state.

**Definition 19.** *The cost of a transaction $\tau$ applied to a state $\mathcal{S}$ is the difference between the cost of the final state minus the cost of the initial state:*

$$\text{costTx} : \mathit{Transaction} \rightarrow \mathit{LedgerState} \rightarrow \mathit{Cost}$$
$$\text{costTx}(\tau, \mathcal{S}) = \mathit{cost}(\text{txRun}(\tau, \mathcal{S})) - \mathit{cost}(\mathcal{S})$$

*The cost of an ordered list of transactions $[T]$ applied to a state $\mathcal{S}$ is the difference between the cost of the final state minus the cost of the initial state:*

$$\text{costTotTx} : [\mathit{Transaction}] \rightarrow \mathit{LedgerState} \rightarrow \mathit{Cost}$$
$$\text{costTotTx}([T], \mathcal{S}) = \mathit{cost}((\text{txRun}(\tau_N).\ ...\ .\text{txRun}(\tau_2).\text{txRun}(\tau_1))(\mathcal{S}_1)) - \mathit{cost}(\mathcal{S})$$

We note that cost represents the size of the ledger's state. However, our model is generic enough to accommodate alternative cost designs as well. For instance, cost could represent the computational effort of producing or verifying the state, such that a cost unit would be a computational cycle. Therefore, our analysis would also be directly applicable in that case, by accordingly adapting some parts of the subsequent optimization framework like the heuristics.

## 6.2 Transaction Optimization

The purpose of a distributed ledger is to execute payments, i.e., transfer value from one party to another via transactions. Multiple transactions can perform the same transfer of value between two parties. Such transactions are *equivalent* in terms of their final result, i.e., transferring some value between parties A and B, but may vary in their cost to the ledger state. *Transaction optimization* is the problem of finding the equivalent transaction with minimum cost; our work is heavily inspired by the seminal research on database query optimization [Ioa96].

The cost difference between equivalent transactions may be significant. For example, assume that Alice wants to give Bob $100$ coins and owns a UTxO of $100$ coins and $100$ UTxOs of $1$ coin each. Consider the two equivalent plans: 1) Alice spends the single UTxO of value $100$ and creates $100$ outputs of value $1$ for Bob; 2) Alice spends the $100$ UTxOs of $1$ coin value and defines a single UTxO of value $100$ to transfer to Bob. The cost of the two approaches exemplifies the ledger state impact that equivalent transactions may have. The first plan increases the ledger's state by $99$ UTxOs, while the second decreases it by the same amount.

Following, we use the terms plan and transaction interchangeably, i.e., an alternative plan that achieves the same goal is expressed as an alternative, equivalent transaction. Definition 20 describes transaction equivalency, while Definition 21 defines equivalency between two ordered lists of transactions.

**Definition 20.** *Transactions $\tau_1, \tau_2$ are equivalent (denoted $\tau_1 \equiv \tau_2$) if, when applied to the same state $\mathcal{S}_A$ of a ledger $\mathcal{L}$, they result in states $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, with the same total accumulated value per unique address $\alpha$:*

$$\forall \alpha \in A \quad \sum_{\substack{i \in \mathcal{S}_1 \\ o_i = \mathsf{UTxO}(i, \mathcal{L}) \\ o_i.\mathit{address} = \alpha}} o_i.\mathit{value} = \sum_{\substack{j \in \mathcal{S}_2 \\ o_j = \mathsf{UTxO}(j, \mathcal{L}) \\ o_j.\mathit{address} = \alpha}} o_j.\mathit{value}$$

*where $A$ is the set of all addresses of the parties participating in the ledger system.*

**Definition 21.** *Two different totally ordered sets of the same $N$ transactions $[T_i]$ and $[T_j]$ are equivalent (denoted as $[T_i] \equiv [T_j]$) if, when applied to the same ledger state $\mathcal{S}_A$ of a ledger $\mathcal{L}$, they result in states $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, where the total accumulated value per unique address $\alpha$ is the same in both states:*

$$\forall \alpha \in A \quad \sum_{\substack{i \in \mathcal{S}_1 \\ o_i = \text{UTxO}(i,\mathcal{L}) \\ o_i.address = \alpha}} o_i.value = \sum_{\substack{j \in \mathcal{S}_2 \\ o_j = \text{UTxO}(j,\mathcal{L}) \\ o_j.address = \alpha}} o_j.value$$

*where $A$ is the set of addresses of all participants in the distributed ledger system.*

Following, we define the basic logical operators for expressing a transaction and explore optimization techniques for compiling the optimal transaction plan.

### 6.2.1 Transaction Logical Operators - Ledger State Algebra

First, we introduce some basic *logical* operators, i.e., functions used to form a transaction. The operators are regarded as basic logical steps for executing a transaction, i.e., irrespective of their particular implementation. However, depending on their implementation, each step may correspond to different cost. The operators operate on and produce a state, forming — possibly equivalent (cf. Definition 20) — *transactions*. The operators and operands form a *ledger state algebra* and, as the state is a set of UTxOs (cf. Definition 18), all common set operators are applicable. In case of failure, they return the empty state $\emptyset$. The three operators are as follows:

- *Input Selection:* $\sigma_{(P_{id},V)} : LedgerState \rightarrow LedgerState$

  $\sigma_{(P_{id},V)}$ is a unary operator, which is given as input parameter a pair *(Party id, Value)*. *Party id* is an abstraction of a set of UTxOs, e.g., it could abstract a *wallet* that controls a set of addresses, each owning multiple UTxOs. When applied on a state $\mathcal{S}_i$, $\sigma_{(P_{id},V)}$ produces a new state $\mathcal{S}_f \subset \mathcal{S}_i$, where $\forall o \in \mathcal{S}_f : o \in P_{id}$ and $\sum_{o \in P_{id}} o.value \geq V$. Essentially, $\sigma$ is a filter over a state, selecting the UTxOs with aggregate value larger than, or equal to the input $V$.

- *Output Creation* $\pi_{[(a_1,v_1),\ldots,(a_n,v_n)]} : LedgerState \rightarrow LedgerState$

  $\pi_{[(a_1,v_1),\ldots,(a_n,v_n)]}$ is a unary operator, which is given a set of *(Address, Value)* pairs and is applied on a state $\mathcal{S}_i$. It produces a new UTxO set $\mathcal{S}_f$ with $\mathcal{S}_f \cap \mathcal{S}_i = \emptyset$, i.e.,

$\mathcal{S}_f$ includes only new UTxOs. Also $\forall o \in \mathcal{S}_f : (o.address, \sum_{o.address} o.value) \in [(a_1, v_1), ... a_n, v_n)]$, i.e., the aggregate output value per address is equal to the input parameter. We require that value is preserved, i.e., the total value in $\mathcal{S}_i$ is greater than (or equal to) the total value in $\mathcal{S}_f$; the value difference is is the miners' fee.

- *Transaction Validation* $\tau_{V_R,\mathcal{S}_i} : LedgerState \rightarrow LedgerState \rightarrow LedgerState$

  $\tau_{V_R,\mathcal{S}_i}$ is a binary operator that validates input and output states $\mathcal{S}_I, \mathcal{S}_O$, against a set of rules $V_R$, over an initial state $\mathcal{S}_G$. If validation succeeds, it returns an updated state $\mathcal{S}_f = (\mathcal{S}_G - \mathcal{S}_I) \cup \mathcal{S}_O$.

Figure 6.2 depicts the simplest transaction under our algebra, i.e., a tree with a root and two branches. The root is the transaction validation operator $\tau$, that receives two inputs: a) the set of selected inputs ($\sigma$ on the left branch) and b) the set of outputs to be created ($\pi$ on the right branch). We express this transaction algebraically as: $T = (\sigma_{Alice,V}) \langle \tau \rangle (\pi_{Bob,V})$, $\langle \tau \rangle$ being the infix validation operator.

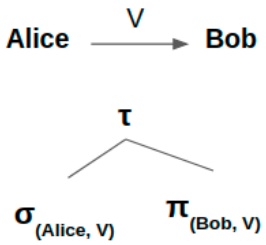


Figure 6.2: The simplest expression of a transaction.

Moving one step further, we assume three transactions $\tau_1$, $\tau_2$ and $\tau_3$. The execution of these transactions is *totally ordered*, i.e., $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$. Figure 6.3 depicts this expression. Here, $\tau_1$ is nested within $\tau_2$ and both are nested within $\tau_3$. Such tree is executed from bottom to top, therefore $\tau_2$ is given the ledger state generated after $\tau_1$ is executed; similarly, $\tau_3$ is given the ledger state generated after both $\tau_1$ and $\tau_2$ are executed. Given the above, we next define *subtransactions*; interestingly, transactions may spend outputs created from their subtransactions, thus we also define the notion of *correlated transactions*.

**Definition 22.** *A subtransaction is a transaction nested within a "parent" transaction; it is executed first, so its impact on the ledger state is visible to the parent.*

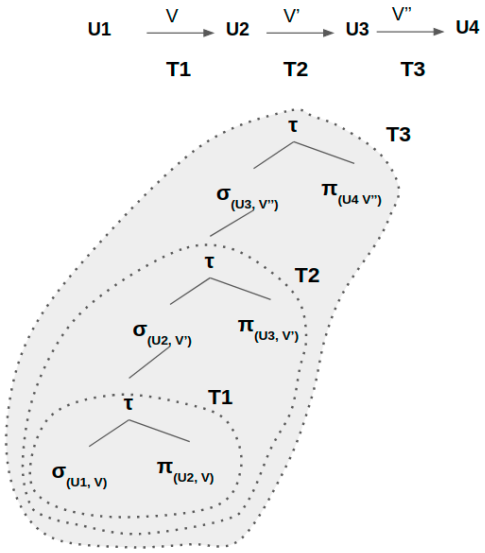


Figure 6.3: The expression tree entails a transaction execution total order.

**Definition 23.** *Two transactions $\tau_1, \tau_2$ are correlated, if $\tau_1$ is a subtransaction of $\tau_2$ and $\tau_2$ spends at least one output created by $\tau_1$.*

### 6.2.2 A Transaction Optimization Framework

We now identify different phases in the transaction optimization process; in a hypothetical *transaction optimizer* each phase would be a distinct module. These phases are different approaches to producing equivalent transactions. The phases operate on three levels of optimization: a) a declarative (rule-based) level, b) a logical/algebraic (cost-based) level, and c) a physical/algorithmic (cost-based) level, as depicted in Figure 6.4. The input of the process is a transaction set $[\tau_x]$, that we want to optimize, and while output is the optimal transaction $\tau_{x-Optimal}$.

**Rules.** This phase is declarative, as it does not depend on the cost; instead, when applied, it necessarily produces a better transaction. Essentially it consists of *heuristic rules* that are applied by default to produce an equivalent transaction; example of such rules are "create a single output per address" or "consume as many inputs and create as few outputs as possible".

**Algebraic Transformations.** These are transformations at the level of logical operators that define a transaction's execution. Generally the efficiency of such transformation is evaluated based on the entailed cost. Examples of such transformations are

the 2-for-1 transformation (cf. Definition 24) and different transaction orderings (cf. Definition 22).

**Methods and Structures.** This phase optimizes the algorithm that implements a logical operator. For instance, given two algorithms $A, B$ result in transaction costs $C_A, C_B$, if $C_A < C_B$ we would choose $A$; one such example is the different implementations of the input selection operator $\sigma$, as shown in Figure 6.5. Optimizations in this phase may also change the data structure used to access the underlying data, which in our case is the ledger state.

**Planning and Searching.** This phase employs a *searching strategy* to explore the available space of candidate solutions, i.e., equivalent transaction plans. This space consists of the transactions produced from the above phases, each evaluated based on their cost, under the available cost model.

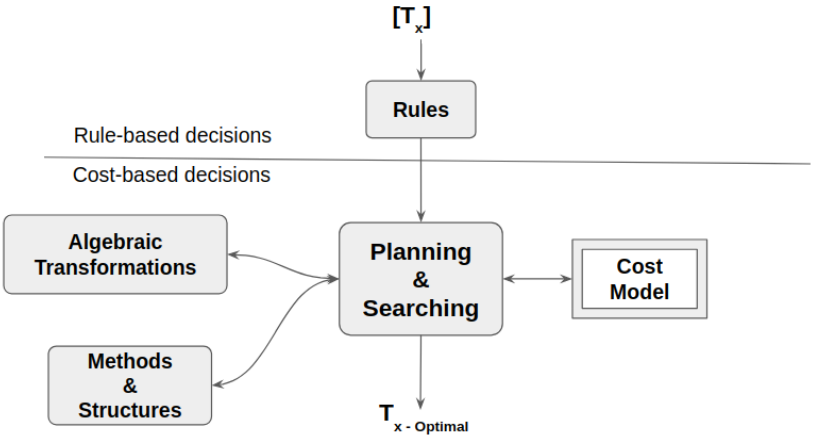


Figure 6.4: The transaction optimization process.

### 6.2.3 Transaction Optimization Techniques

In this section, we propose three transaction optimization techniques based on the aforementioned optimization levels: a) heuristic rule-based, b) logical/algebraic transformation cost-based, and c) physical/algorithmic cost-based.

#### 6.2.3.1 Input Selection Optimization

We demonstrate this technique with an example. Assume Alice wants to give Bob $5$ coins. Figure 6.5 depicts three equivalent transactions for implementing this payment.

Observe that each plan is represented as a tree, where the intermediate nodes are the previously defined logical operators (that act on a ledger state) and the leaf nodes are ledger states. We also assume that the state cost is the number of elements (UTxOs) in the state. The three transactions have the same structure, i.e., they are the same logical expression, but result to different ledger states with different costs. The transactions differ only in the output of the input selection operator ($\sigma_{(Alice,5)}$), a difference which may be attributed to different implementations of the operator.

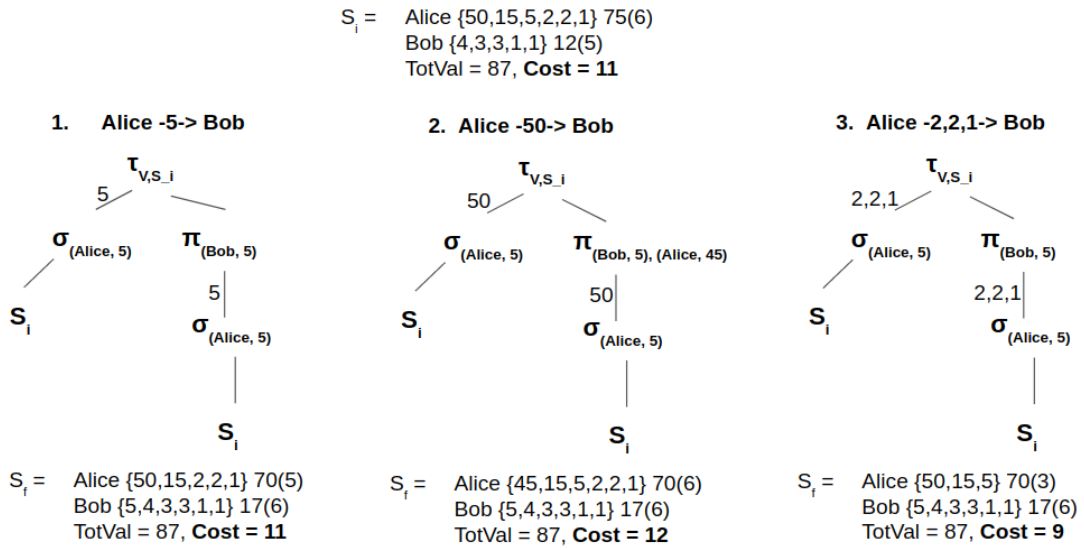


Figure 6.5: An example of three equivalent transactions that transfer 5 tokens from Alice to Bob but incur different state costs.

#### 6.2.3.2 The 2-for-1 Transformation

We again consider the example where Alice wants to give Bob $5$ coins. Figure 6.6 depicts a fourth, more complex, equivalent transaction. This transaction consists of two subtransactions (cf. Definition 22), where Alice first gives Bob $17$ coins and then receives $12$. When the first transaction is completed, an intermediate state ($S_i'$) is created, which is then given as input to the second transaction, that produces the final ledger state $\mathcal{S}_f$ of cost $3$. Observe that, although more complex, this transaction minimizes the final ledger state ($72\%$ cost reduction). Intuitively, this transaction spends all of Alice's outputs with the first sub-transaction and then does the same for Bob with the second sub-transaction. Therefore, the optimal cost does not depend on input selection (like the 3rd plan of Figure 6.5), but requires the combination of two transactions that implement a single payment, under a specific amount ($12$). Definition 24 provides a formal

specification of the *2-for-1* logical (algebraic) transformation.

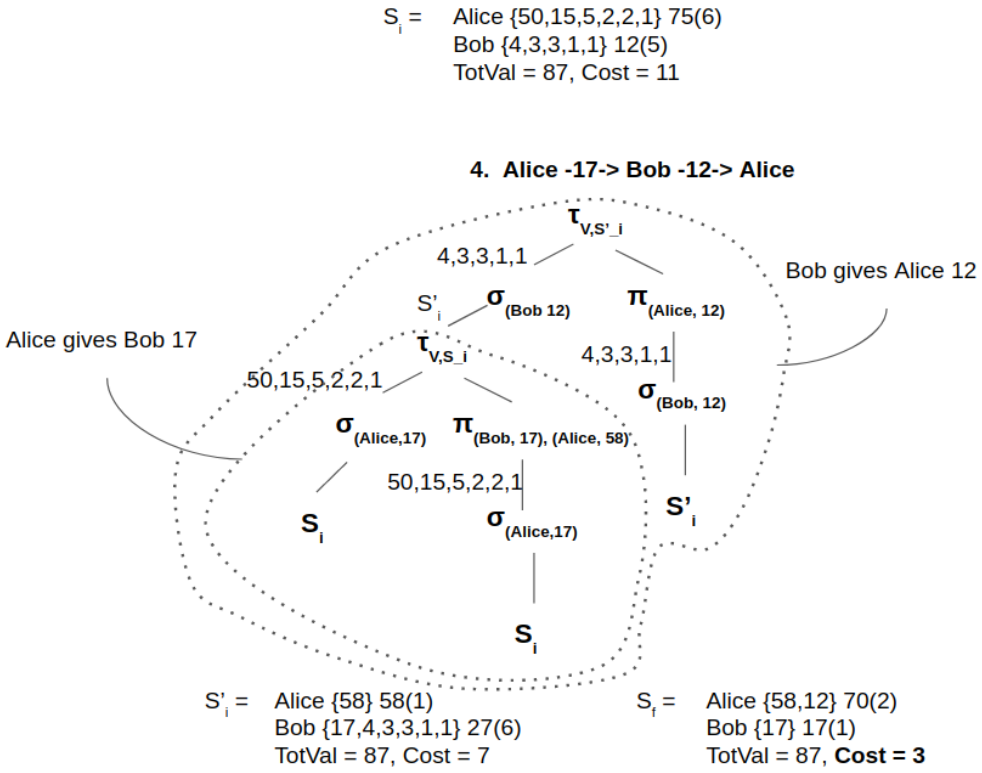


Figure 6.6: A 2-for-1 transaction that transfers 5 tokens from Alice to Bob.

**Definition 24.** *Given a transaction $\tau_1$, which transfers an amount $V$ from party A to B, the algebraic 2-for-1 transformation creates an equivalent transaction $\tau_2$, which consists of (a) a subtransaction, which transfers $V + V_c$ from party A to B and (b) an outer transaction, which transfers $V_c$ from party B to A.*

Figure 6.7 depicts the 2-for-1 algebraic transformation based on an amount $V_c$. To implement such a scheme we require an *atomic operation*, where the grouped transactions are executed simultaneously. One method to implement the atomic transfers is CoinJoin [Max13b], which was proposed for increasing the privacy in Bitcoin; in CoinJoin, the transaction is constructed and signed gradually by each party that contributes its inputs. A similar concept is Algorand's atomic transfers [Fus19], that achieves atomicity by grouping transactions under a common id.

Intuitively, 2-for-1 reduces the transaction's cost by also consuming UTxOs of the receiving party, instead of only consuming outputs of the sending party. Specifically, assume the initial state $\mathcal{S}_i = \{|A|, |B|\}$, where $|A|$ denotes the number of outputs owned by party A. When issuing a payment to B, party A can consume all outputs and consolidate its remaining value to a single UTxO, the "change" output. Such transaction

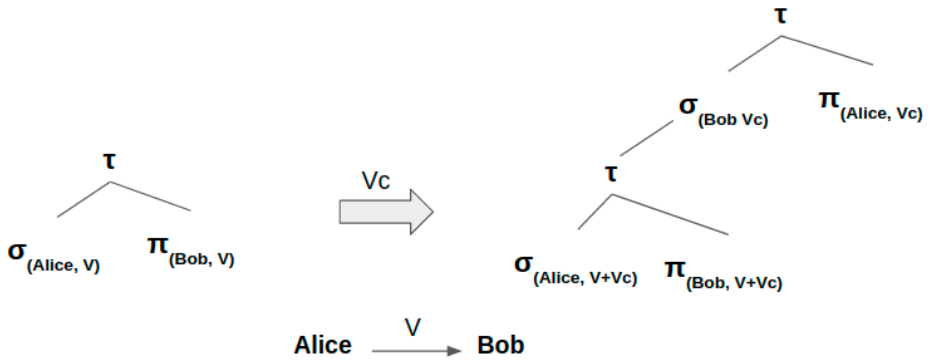


Figure 6.7: The 2-for-1 algebraic transformation.

results in state $\mathcal{S}_f = \{1, |B|+1\}$ with cost $cost(\mathcal{S}_f) = |B|+2$. If we apply the 2-for-1 transformation, the final state is $\mathcal{S}'_f = \{1+1, 1\}$ with a cost of $cost(\mathcal{S}'_f) = 3$; if $|B| > 1$, then $cost(\mathcal{S}'_f) < cost(\mathcal{S}_f)$. Therefore, if the receiving party has multiple outputs, this transformation creates a transaction with a smaller cost. Consequently, by giving the opportunity to the receiving party of a transaction to spend also its outputs, the 2-for-1 transformation *always* results in a greater shared state cost reduction than the individual un-transformed transaction in the case where there are no fee constraints and thus outputs can be spent freely; otherwise it is a cost-based decision.

#### 6.2.3.3 Transaction Total Ordering and the Last-Payer Heuristic Rule

Assume the following four transactions: (1) $T_1$: Alice $\xrightarrow{V_1}$ Charlie, (2) $T_2$: Bob $\xrightarrow{V_2}$ Charlie, (3) $T_3$: Eve $\xrightarrow{V_3}$ Alice, and (4) $T_4$: Eve $\xrightarrow{V_4}$ Bob, which are applied on an initial ledger state $\mathcal{S}_i = \{|Alice| = 5, |Bob| = 5, |Charlie| = 2, |Eve| = 3\}$ with cost $cost(\mathcal{S}_i) = 15$; as before, $|A|$ denotes the number of outputs owned by party A and the state cost is the number of all UTxOs.

A first execution order is as follows: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$. For simplicity and without loss of the generality, we assume that when a party pays, it always consumes all available outputs, thus having with a single output afterwards (the leftover balance). Similarly, when a party gets paid, the number of UTxOs that it owns increases by one. Next, we describe the ledger state changes each transaction is executed:

$\mathcal{S}_i = \{|Alice| = 5, |Bob| = 5, |Charlie| = 2, |Eve| = 3\}, cost = 15$

$T_1 : \{|Alice| = 1, |Bob| = 5, |Charlie| = 3, |Eve| = 3\}, cost = 12$

$T_2 : \{|Alice| = 1, |Bob| = 1, |Charlie| = 4, |Eve| = 3\}, cost = 8$

$T_3 : \{|Alice| = 2, |Bob| = 1, |Charlie| = 4, |Eve| = 1\}, cost = 8$
$T_4 : \{|Alice| = 2, |Bob| = 2, |Charlie| = 4, |Eve| = 1\}, cost = 9$

Assuming a different order, $T_3 \rightarrow T_4 \rightarrow T_1 \rightarrow T_2$, the state changes as follows:

$\mathcal{S}_i = \{|Alice| = 5, |Bob| = 5, |Charlie| = 2, |Eve| = 3\}, cost = 15$
$T_3 : \{|Alice| = 6, |Bob| = 5, |Charlie| = 2, |Eve| = 1\}, cost = 14$
$T_4 : \{|Alice| = 6, |Bob| = 6, |Charlie| = 2, |Eve| = 1\}, cost = 15$
$T_1 : \{|Alice| = 1, |Bob| = 6, |Charlie| = 3, |Eve| = 1\}, cost = 11$
$T_2 : \{|Alice| = 1, |Bob| = 1, |Charlie| = 4, |Eve| = 1\}, cost = 7$

Evidently, the different execution order results in different resulting state cost. Therefore, by changing the nesting order of the transactions in an expression tree, different plans may conduct the same payment with different cost.

Intuitively, parties should have the ability to consume outputs that are produced by the other transactions. For instance, regarding $T_1$ and $T_3$, the order $T_3 \rightarrow T_1$ is more cost effective ($cost = 10$) than $T_1 \rightarrow T_3$ ($cost = 11$), since Alice can consume the output created by Eve. Specifically, if in the last transaction where $\mathcal{P}$ participates, either as a sender or a receiver, $\mathcal{P}$ is the sender, then it can minimize its state cost; we call this the *last-payer heuristic rule*.

### 6.2.4 The Transaction Optimization Problem

Using the above ideas, we now formally define the transaction optimization problem as a typical *optimization problem*, assuming a set of available input selection algorithms $\{Sel_1, Sel_2, ..., Sel_l\}$.

**Definition 25.** *Given $N$ payments between $M$ parties $\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_M$ and a search space $\mathcal{S}$ of equivalent (cf. Definition 21), ordered lists of transaction plans that execute the $N$ payments, called candidate solutions, find the candidate $\tau \in \mathcal{S}$, such that $eval(\tau) \leq eval(\rho)$, for all $\rho \in \mathcal{S}$. Specifically:*

1. *A candidate $\rho \in \mathcal{S}$ is an ordered list of transaction plans*[1] $||T_1|| \rightarrow ||T_2|| \rightarrow \cdots \rightarrow ||T_k||$*, where the transaction plan of a transaction $T_x$ is the pair: $||T_x|| \stackrel{def}{=}$ (Logical Expression, Input Selection Algorithm).*

[1] We assume that transactions are non-correlated (cf. Definition 23) and that all orderings are equivalent (cf. Definition 21).

2. *The search space $\mathcal{S}$ is defined by all candidates $||T_1|| \rightarrow ||T_2|| \rightarrow \cdots \rightarrow ||T_k||$, where, for each transaction $T_i$, an input selection algorithm from $\{Sel_1, Sel_2, ..., Sel_l\}$ is chosen and, possibly, the 2-for-1 logical transformation (cf. Definition 24) is applied.*

3. *$eval$ evaluates the cost of every candidate $\rho \in \mathcal{S}$ (cf. Definition 19) as follows:*

$$eval : [Transaction] \rightarrow LedgerState \rightarrow Cost,$$
$$eval([T_1, T_2, ..., T_k], \mathcal{S}_{init}) =$$
$$cost((txRun(T_k).\ ...\ .txRun(T_2).txRun(T_1))(\mathcal{S}_{init})) - cost(\mathcal{S}_{init})$$

*where $cost(S) = |S|$ is the size of a ledger state (cf. Definition 18) and the functions $(txRun(T_k).\ ...\ .txRun(T_2).txRun(T_1))(\mathcal{S}_{init})$ output the final state after the list of ordered transactions is executed on the initial state $\mathcal{S}_{init}$, according to each individual transaction plan $||T_i||$.*

**Solving the Transaction Optimization Problem.** We now present a 3-step, dynamic programming algorithm, which solves the transaction optimization problem via an exhaustive search and dynamically pruning candidate solutions:

1. Create $N$ transactions $T_{ij}$, $i, j \in [1, M]$, corresponding to the payments ($\mathcal{P}_i \xrightarrow{V_{ij}} \mathcal{P}_j$), as follows:

$$T_{ij} = (\sigma_{\mathcal{P}_i, V_{ij}}(\mathcal{S}_{init}))\langle\tau\rangle(\pi_{\mathcal{P}_j, Vij}(\mathcal{S}_{init}))$$

   where $V_{ij}$ is the amount to be paid from $\mathcal{P}_i$ to $\mathcal{P}_j$. For each $T_{ij}$, find the input selection algorithm in $\{Sel_1, Sel_2, ..., Sel_l\}$ that minimizes $eval(T_{ij}, \mathcal{S}_{init})$. Then, enforce the *heuristic rule* to create a *single* output per recipient address for each transaction. At the end of this step, the algorithm outputs $N$ transaction plans, i.e., $N$ pairs of transaction's $T_{ij}$ logical expression and the chosen input selection algorithm:

$$||T_{ij}|| = ((\sigma_{\mathcal{P}_i, V_{ij}}(\mathcal{S}_{init}))\langle\tau\rangle(\pi_{\mathcal{P}_j, Vij}(\mathcal{S}_{init})),\ Sel_s)$$

2. On each transaction plan output of Step 1, perform a 2-for-1 transformation (cf. Definition 24). This step produces a transformed transaction as depicted in Figure 6.8, based on an amount $p \times V_{ij}$, where $p$ is a configuration parameter of the algorithm, typically in the range $0 < p \leq 1$. Then, for each of the two transactions

that comprise the 2-for-1 transformation, choose the input selection algorithm that minimizes the $eval$ function and enforce the heuristic rule of a *single* output per recipient address. Finally, accept the 2-for-1 transformed transaction only if its cost (given by $eval$) is smaller than the non-transformed transaction.

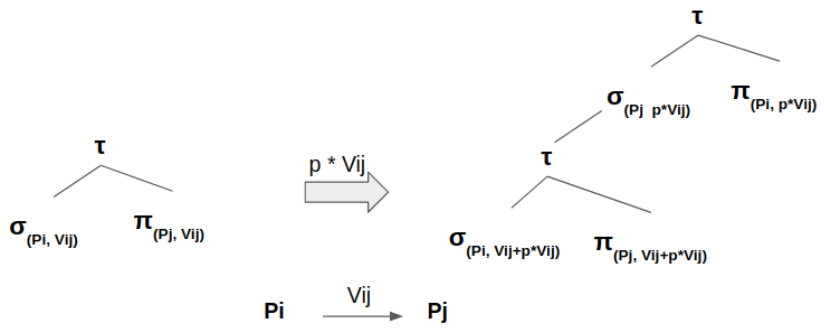


Figure 6.8: Applying the 2-for-1 transformation to each separate transaction.

At the end of this step, the algorithm outputs $k$ transaction plans, $k \geq N$, comprising of the 2-for-1 transformed and the non-transformed transactions, along with their input selection algorithms. Importantly, at this point, the algorithm has an optimal plan for each *individual* payment ($\mathcal{P}_i \xrightarrow{V_{ij}} \mathcal{P}_j$), based on an exhaustive search of solutions and cost-driven choices.

3. In this step, the algorithm finds the optimal total order to execute the $k$ transactions produced on Step 2. Since there exist $k!$ permutations, the search space is pruned using the *Last-Payer heuristic rule* (cf. Section 6.2.3). At the end of this step, the algorithm outputs a totally ordered set of transaction plans that solve the optimization problem, i.e., executes the $N$ payments with a minimum state ledger cost.

Our algorithm's *asymptotic time complexity* depends on its slowest element. For simplicity, we assume that applying a transaction on a ledger state requires constant time. Therefore, in Step 1, creating each of the $N$ transactions is $O(1)$. However, each of the $l$ input selection algorithms may have different complexity when parsing the $n$ UTxO inputs. For example, [KKK21a] shows that to minimize "change", i.e., to find inputs that exactly match the payment's output, requires solving a Knapsack problem, i.e., $O(2^n)$, but is not necessarily state efficient; instead, the paper proposes an input selection algorithm with complexity $O(n \log n)$. Step 2 has the same asymptotic complexity as Step

1, as performing the 2-for-1 transformation requires $O(1)$, as does the application of heuristics, while choosing an input selection algorithm is the same as in Step 1. Finally, Step 3 requires $O(N!)$, in the corner case when the heuristics do not prune the search space at all.

On the whole, assuming the complexity of the least efficient input selection algorithm is $O(\lambda(n)$, our algorithms complexity is $O(\lambda(n) + N!)$, where $n$ are the available inputs and $N$ is the number of payments. Theoretically speaking, this is a rather inefficient solution. In practice though, the wallets of regular users do not control many UTxOs, so $n$ is expectedly low. The same holds for $N$, as regular users are not expected to perform complex payments towards multiple third parties. Nonetheless, optimizing the employed modules, e.g., designing more efficient input selection algorithms and heuristics, offers an interesting path for future research.

Regarding *optimality*, the transaction plans produced in Step 2 are optimal w.r.t. executing the individual transactions, since the algorithm performs an exhaustive search for the minimum-cost solution. In Step 3, the algorithm is based on a heuristic rule (Last-Payer) to search for the optimal total order in a pruned (and thus manageable) space. Future work will evaluate the efficiency of this heuristic rule, i.e., how well it approximates the optimal solution, and explore further rules for achieving optimality.

## 6.3 State Efficiency in Bitcoin

We now define the *state efficiency* property. Our goal is to incentivize users to minimize the global state, without impacting the system's functionality. In that case, if all users are rational, i.e., operate following the incentives, then the state will be minimized as much as possible. Future work will explore the actual impact of deploying such incentives in real-world systems.

To achieve state efficiency, a transaction's fee should be proportional to the incurred state cost. In other words, the more a transaction increases the ledger's state, the higher its fees should be. Specifically, a transaction's fee should reflect: i) the transaction's size, i.e., the cost of storing a transaction permanently on the ledger and ii) the transaction's state cost. A distributed ledger's fee model should aim at incentivizing users to minimizing both storage types, i.e., the distributed ledger and the global state.

First, we define the *fee function* $F$, i.e., the function that assigns an (integer) fee on a transaction, given a ledger state: $F$ : *Transaction* $\rightarrow$ *LedgerState* $\rightarrow$ *Int*. Following, Definition 26 describes *state efficiency*. This property instructs the fee function to

(monotonically) increase fees, if a transaction increases the state. Intuitively, between two equivalent transactions, the transaction that incurs greater state cost should also incur a larger fee.

**Definition 26.** *A fee function $F$ is state efficient if*

$$\forall \mathcal{S} \in \mathbb{S} \forall \tau_1, \tau_2 \in \mathbb{T} \mid \tau_1 \equiv \tau_2 \wedge \mathit{costTx}(\tau_1, \mathcal{S}) > \mathit{costTx}(\tau_2, \mathcal{S}) : F(\tau_1, \mathcal{S}) > F(\tau_2, \mathcal{S})$$

*for transaction cost function (cf. Definition 19) and equivalence (cf. Definition 20).*

Evidently, if the utility of users is to minimize transaction fees, a state efficient fee function ensures that they are also incentivized to minimize the global state. Finally, Definition 27 sets *narrow state efficiency*, a special case of state efficiency which compares equivalent transactions that differ only in their inputs.

**Definition 27.** *A fee function $F$ is narrow state efficient if*

$$\begin{aligned} &\forall \mathcal{S} \in \mathbb{S} \forall \tau_1, \tau_2 \in \mathbb{T} \mid \\ \tau_1 \equiv \tau_2 \wedge \tau_1.\mathsf{outputs} = \tau_2.\mathsf{outputs} \wedge \mathit{costTx}(\tau_1, \mathcal{S}) &> \mathit{costTx}(\tau_2, \mathcal{S}) : \\ &F(\tau_1, \mathcal{S}) > F(\tau_2, \mathcal{S}) \end{aligned}$$

*for transaction cost function (cf. Definition 19) and equivalence (cf. Definition 20).*

**Bitcoin's State Management.** Bitcoin's consensus model does not consider fees. Specifically, the user decides a transaction's fees and the miners choose whether to include a transaction in a block. Therefore, it has been stipulated that the level of fees is the balance between the rational choices of miners, who supply the market with block space, and users, who demand part of said space [Bit20a].

In practice, users follow the client software's choice even when not needed [MB15], e.g., when blocks are not full. Similarly, miners usually follow the hard-coded software rules and may accept even zero-fee transactions. The reference rules of the Bitcoin Wiki [Bit20a] define the *fee rate* $x$, which is the fraction of fees per transaction size, Miners sort transactions based on this metric and solve the Knapsack problem to fill a new block with transactions that maximize it. Some notable alternatives also focus on fee rate [DCKT19, Riz15], while reference rules [Bit20a] used to also take into account the UTxO age.

As before, a transaction consists of inputs and outputs, i.e., old UTxOs which are spent and newly-created UTxOs. Inputs and UTxOs have a fixed size $\iota$ and $\omega$ respec-

tively.[2] The size of a transaction is the sum of its inputs and outputs, i.e., is a linear combination of $\iota$ and $\omega$, while a transaction's cost is the difference between the number of its UTxOs minus its inputs. Bitcoin's fee function is $F = \beta \cdot \mathsf{size}(\tau)$, where $\mathsf{size}(\tau)$ is $\tau$'s size in bytes and $\beta$ is a fixed fee per byte.[3]

We break the fee efficiency of $F$ via a counterexample. Assume two transactions which are applied on the same ledger state $\mathcal{S}$; for ease of notation, in the rest of the section $F(\tau)$ denotes $F(\tau, \mathcal{S})$. First, $\tau_1$ has $1$ input and $1$ output, so its state cost is $\mathsf{costTx}(\tau_1, \mathcal{S}) = 0$ and its fee is $F(\tau_1) = \beta \cdot (\iota + \omega)$. Second, $\tau_2$ has $2$ inputs and $1$ output, i.e., its state cost is $\mathsf{costTx}(\tau_2, \mathcal{S}) = -1$, since it decreases the state; however, its fee is $F(\tau_2) = \beta \cdot (2 \cdot \iota + \omega) = F(\tau_1) + \beta \cdot \iota$. Thus, although $\mathsf{costTx}(\tau_1) > \mathsf{costTx}(\tau_2)$, $\tau_2$'s fee is higher, since it is larger.

A better alternative fee function is the following: $F' = \beta \cdot \mathsf{size}(\tau) + \psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$. Note that this is state-efficient in our model for a sufficiently small value of $\beta$ (cf. Section 6.3.1). Observe with this function, when increasing the UTxO set, a user needs to pay an extra fee $\psi$ per UTxO. Given this change, the reference rules are updated so that, instead of the fee rate, miners use the scoring function: $\mathsf{score}(\tau) = \frac{\mathsf{fees}(\tau) - \psi \cdot \mathsf{costTx}(\tau, \mathcal{S})}{\mathsf{size}(\tau)}$, where $\mathsf{fees}(\tau)$ are $\tau$'s total fees. In market prices, $1$ byte of RAM costs approximately $\$3.35 \cdot 10^{-9}$ [McC20]. The average size of a Bitcoin UTxO is $61$ Bytes [DPNH19], so a single Bitcoin UTxO costs $\psi = 61 \cdot 3.35 \cdot 10^{-9} = \$2 \cdot 10^{-7}$. Given $10000$ full nodes[4], which maintain the ledger and keep the UTxO in memory, the cost becomes $\psi = \$0.002$; equivalently, denominated in Bitcoin[5], the cost of creating a UTxO is $\psi = 22$ satoshi.

This solution incorporates the operational costs of miners, thus it is the rational choice for miners who aim at maximizing their profit. Observe that, after subtracting the fees that relate to UTxO costs, the scoring mechanism behaves the same as the one currently used by Bitcoin miners. Therefore, if users wish to prioritize their transactions, they would again simply increase their transaction's fees; in that case, the UTxO portion of the fees (i.e., $\psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$) remains the same, hence higher fees result in a higher score, similar to the existing mechanism. Also we note that this mechanism is directly enforceable on Bitcoin without the need of a fork.

[2] This assumption slightly diverges from the real-world, where UTxOs are typically of varying size depending on the operations in the ScriptPubKey.

[3] $\beta = 0.0067\$/\text{byte}$ [September 2020] (`https://bitinfocharts.com`)

[4] `https://bitnodes.io` [July 2020]

[5] $1BTC = \$9000$ [`https://coinmarketcap.com`; July 2020]

### 6.3.1 A State Efficient Bitcoin

Intuitively, to make $F$ state efficient we force the creator of a UTxO to subsidize its consumption, i.e., to pay the user who later consumes it. Our fee function is again: $F' = \beta \cdot \mathsf{size}(\tau) + \psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$. Assume two transactions $\tau_1, \tau_2$ with $i_1, i_2$ inputs and $o_1, o_2$ outputs respectively:

$$\mathsf{costTx}(\tau_1) > \mathsf{costTx}(\tau_2) \Leftrightarrow o_1 - i_1 > o_2 - i_2 \Leftrightarrow o_2 - o_1 < i_2 - i_1 \tag{6.1}$$

$F'$ is state efficient (cf. Definition 26) if:

$$\begin{aligned}
F'(\tau_1) > F'(\tau_2) \Rightarrow \\
\mathsf{size}(\tau_1) \cdot \beta + \mathsf{costTx}(\tau_1) \cdot \psi > \mathsf{size}(\tau_2) \cdot \beta + \mathsf{costTx}(\tau_2) \cdot \psi \Rightarrow \\
(i_1 \cdot \iota + o_1 \cdot \omega) \cdot \beta + (o_1 - i_1) \cdot \psi > (i_2 \cdot \iota + o_2 \cdot \omega) \cdot \beta + (o_2 - i_2) \cdot \psi \Rightarrow \\
(o_1 - i_1) \cdot \psi - (o_2 - i_2) \cdot \psi > (i_2 \cdot \iota + o_2 \cdot \omega) \cdot \beta - (i_1 \cdot \iota + o_1 \cdot \omega) \cdot \beta \Rightarrow \\
(i_2 - i_1 + o_1 - o_2) \cdot \psi > ((i_2 - i_1) \cdot \iota + (o_2 - o_1) \cdot \omega) \cdot \beta \overset{(6.1)}{\Longrightarrow} \\
\psi > \frac{(i_2 - i_1) \cdot \iota + (o_2 - o_1) \cdot \omega}{(i_2 - i_1) - (o_2 - o_1)} \cdot \beta
\end{aligned} \tag{6.2}$$

If $F'$ is narrow state efficient, then $o_1 = o_2$ and the inequality is simplified:

$$\psi > \iota \cdot \beta \tag{6.3}$$

We turn again to the previous example. For transaction $\tau_1$, with $1$ input and $1$ output, $F'(\tau_1) = (\iota + \omega) \cdot \beta$ and for transaction $\tau_2$, with 2 inputs and $1$ output, $F'(\tau_2) = (2 \cdot \iota + \omega) \cdot \beta - \psi = F'(\tau_1) + \beta \cdot \iota - \psi$. Since Inequalities 6.2 and 6.3 ensure that $\psi > \iota \cdot \beta$, the size fee of the extra input in $\tau_2$ is offset by the extra fee $\psi$, which is paid by the user who creates it. Again to evaluate these variables we consider market prices. The size of a typical, pay-to-script-hash or pay-to-public-key-hash, UTxO is $34$ Bytes [Bit20b], while the size of consuming it is $146$ bytes. Therefore, to make and make present-day Bitcoin (narrow) state efficient, we can set $\omega = 34$, $\iota = 146$, $\beta = 0.0067$\$, and thus $\psi > 0.0978$\$.

However, this approach presents a number of challenges. To enforce $F'$, the fee policy should be incorporated in the consensus protocol and a transaction's validity will depend on its amount of fees. As long as $F'(\tau) > 0$, i.e., a transaction cannot have negative fees, the fee function can be enforced via a *soft* fork. Specifically, this change is backwards compatible, as miners that do not adopt this change will still accept transactions that follow the new fee scheme. However, if $\mathsf{costTx}(\tau) \ll 0$ and possibly

$F'(\tau) < 0$, to implement $F'$ we need to establish a "pot" of fees. When a user creates $\tau$ with fee $F' = \beta \cdot \mathsf{size}(\tau) + \psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$, the first part ($\beta \cdot \mathsf{size}(\tau)$) is awarded to the miners as before. The second part ($\psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$) is deposited to (or, in case of negative cost, withdrawn from) the pot. In case of negative cost, the transaction defines a special UTxO for receiving the reimbursement. At any point in time, the size of the pot is directly proportional to the UTxO set. Observe that the miners receive the same rewards as before, so their business model is not affected by this change. Finally, the cost of flooding the system with UTxOs increases by $\psi$ per UTxO which, depending on $\psi$, can render attacks ineffective.

# Chapter 7

# Blockchain Nash Dynamics

With the advent of Bitcoin [Nak08a] the economic aspects of consensus protocols came to the forefront. While classical literature in consensus primarily dealt with "error models", such as fail-stop or Byzantine [PSL80b], the pressing question post-Bitcoin is whether the incentives of the participants align with what the consensus protocol asks them to do.

Motivated by this, existing literature pursued various research paths. One line of work investigated whether the Bitcoin protocol is an equilibrium under certain conditions [KDF13, KKKT16b]. Another, pinpointed protocol deviations that can be more profitable for some players, assuming others follow the protocol [ES14, SSZ16, JLG+14, CKWN16]. Some works proposed tweaks towards improving the underlying blockchain protocol in various settings [FKO+19, KLOS19], game-theoretic studies of pooling behavior [LBS+15, CKWN16, AW19], as well as equilibria involving abstaining from the protocol [FKKP19] in high cost scenaria. Going beyond consensus, economic mechanisms have also been considered in the context of multi-party computation [KMB15, DDL19, DDL18], to disincentivize "cheating". Finally, a large body of research was dedicated to optimizing particular attacks; respresentative works i) identify optimal selfish mining strategies [SSZ16]; ii) propose a framework [GKW+16] for quantitatively evaluating blockchain parameters and identifies optimal strategies for selfish mining and double-spending, taking into account network delays; iii) propose alternative strategies [NKMS15], that are more profitable than selfish mining.

Though the above works provide some glimpses on how Bitcoin and related protocols behave from a game-theoretic perspective, they still offer very little guidance on how to design new consensus protocols. This is a problem of high importance, given the current negative light shed on Bitcoin's perceived energy inefficiency and carbon

footprint [MN21] that asks for alternative protocols. Proof-of-Stake (PoS) ledgers is currently the most prominent alternative to Bitcoin's Proof-of-Work (PoW) mechanism. While PoW requires computational effort to produce valid messages, i.e., blocks which are acceptable by the protocol, PoS relies on each party's stake, i.e., the assets they own, and each block is created at (virtually) no cost. Interestingly, while it is proven that PoS protocols are Byzantine resilient [KRDO17, CGMV18, GHM+17b, BG17] and are even equilibriums under certain conditions [KRDO17], their security is heavily contested by proponents of PoW protocols via an economic argument. In particular, the argument termed the *nothing-at-stake* attack [LABK17, Eth18b, Mar] asserts that parties who maintain a PoS ledger will opt to produce conflicting blocks, whenever possible, to maximize their expected rewards.

What merit do these criticisms have? Participating in a blockchain protocol is a voluntary action that involves a participant downloading the software, committing some resources, and running the software. Furthermore, especially given the open source nature of these protocols, nothing prevents the participant from modifying the behaviour of the software in some way and engage with the other parties following a modified strategy. There are a number of adjustments that a participant can do which are undesirable, e.g., i) run the protocol intermittently instead of continuously; ii) not extend the most recent ledger of transactions they are aware of; iii) extend simultaneously more than one ledger of transactions. One can consider the above as fundamental *infractions* to the protocol rules and they may have serious security implications, both in terms of the consistency and the liveness of the underlying ledger.

To address these issues, many blockchain systems introduce additional mechanisms on top of Bitcoin incentives, frequently with only rudimentary game theoretic analysis. These include: i) rewards for "uncle blocks" in Ethereum; ii) stake delegation [Com18] in Eos and Polkadot, where users assign their participation rights to delegates, as well as stake pools in Cardano [KKL20]; iii) penalties [BG17, BRLP19] for misbehavior in Ethereum 2.0, that enforce forfeiture of large deposits (referred to as *"slashing"*) if a party misbehaves, in the sense of being offline or using their cryptographic keys improperly. The lack of thorough analysis of these mechanisms is of course a serious impediment to the wider adoption of these systems.

For instance, forfeiting funds may happen inadvertently, due to server misconfiguration or software and hardware bugs [Kha21]. A party that employs a redundant configuration with multiple replicas, to increase its crash-fault tolerance, may produce conflicting blocks if, due to a faulty configuration or failover mechanism, two replicas

come alive simultaneously. Similarly, if a party employs no failover mechanism and experiences network connectivity issues, it may fail to participate. Finally, software or hardware bugs can always compromise an – otherwise safe and secure – configuration. This highlights the flip side of such penalty mechanisms: participants may choose to not engage, (e.g., to avoid the risk of forfeiting funds, or because they do not own sufficient funds to make a deposit), or, if they do engage, they may steer clear of fault-tolerant sysadmin practices, such as employing a failover replica, which could pose quality of service concerns and hurt the system in the long run.

The above considerations put forth the fundamental question that motivates our work: *How effective are blockchain protocol designs in disincentivizing particular infractions?* In more detail, the question we ask is whether selfish behavior can lead to deviations, starting with a given blockchain protocol as the initial point of reference of honest — compliant — behavior.

## Contributions

Our main question relates to the Nash-dynamics of blockchain protocols. In the classical Nash-dynamics problem [Ros73], the question is whether selfish players performing step-wise payoff improving moves lead the system to an equilibrium, and in how many steps this may happen; e.g., [FPT04] provides an important case of congestion games. In this perspective, the action space can be seen as a directed graph, with vertices representing vectors of player strategies and edges corresponding to player moves.

In this work, we adapt Nash dynamics to the setting of blockchain protocols, with a particular focus on the study of specific undesirable protocol infractions. Importantly, instead of asking for convergence, we ask whether the "cone" in the directed graph positioned at the protocol contains any strategies that belong to a given set of infractions $\mathcal{X}$ (cf. Figure 7.1). If the cone is free of infractions, then the protocol is said to be $\mathcal{X}$-compliant. Motivated by Chien and Sinclair [CS11], we consider $\epsilon$-Nash-dynamics, i.e., considering only steps in the graph that improve the participant payoff more than $\epsilon$. Armed with this model, we investigate a number of protocols, both in the PoW and PoS setting, from a compliance perspective. Notably, we provide indicative values for $\epsilon$, beyond which a protocol is not compliant. Nonetheless, our work is complimentary to research that investigates optimality of particular attacks; specifically, these works could be used in conjunction with our model to provide tighter, if not optimal, bounds on $\epsilon$.

First, Section 7.2 describes our model of *compliant* strategies and protocols. A strat-

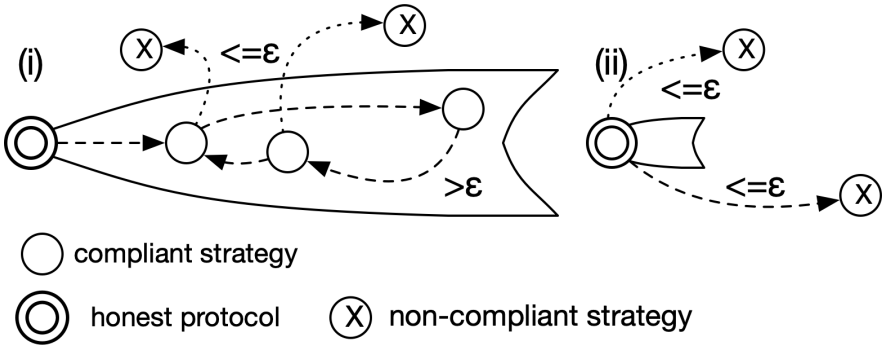


Figure 7.1: Illustration of a compliant protocol that does not exhibit an equilibrium (i), vs a protocol which is an approximate Nash equilibrium (ii).

egy is compliant if a party that employs it never violates a predicate $\mathcal{X}$, which captures well-defined types of deviant behavior. Accordingly, a protocol is compliant if, assuming a starting point where no party deviates, no party will eventually employ a non-compliant strategy, assuming sequential unilateral defections. Section 7.3 specifies compliance for blockchain protocols, under an infraction predicate that captures abstaining and producing conflicting blocks, and two types of utility, absolute rewards and profit. Following, we explore different reward schemes and families of protocols. First, Section 7.4 shows that *fair* rewards, i.e., which depend only on a party's mining or staking power, result in compliance w.r.t. rewards alone (i.e., when costs are negligible), but non-compliance w.r.t. profit (rewards minus costs). Next, we explore block-proportional rewards, i.e., which depend on the blocks adopted by an impartial observer of the system. Section 7.5.1 shows that PoW systems are compliant w.r.t. rewards. Section 7.5.2.1 shows that PoS systems, which enforce that a single party participates at a time, are compliant, under a synchronous network, but non-compliant under a lossy network. Section 7.5.2.2 shows that PoS systems, which allow multiple parties to produce blocks for the same time slot, are not compliant. Notably, our negative results show that a party can gain a *non-negligible* reward by being non-compliant. Finally, we evaluate compliance under various externalities, i.e., an exchange rate, which models real-world prices, and external rewards, which come as a result of successful attacks. We show that, historically, the market's response is not sufficient to disincentivize attacks, so penalties should be necessary, for the level of which we provide estimations based on the ledger's parameters and the market's expected behavior.

## 7.1 The Setting

We consider a distributed protocol $\Pi$, which is executed by all parties in a set $\mathbb{P}$, and which is performed over a number of time slots. Our analysis is restricted on executions where every party $\mathcal{P} \in \mathbb{P}$ is activated on each time slot. The activation is scheduled by an environment $\mathcal{Z}$, which also provides the parties with inputs.

We use $\kappa$ to denote $\Pi$'s security parameter, and $\mathsf{negl}(\cdot)$ to denote that a function is negligible, i.e., asymptotically smaller than the inverse of any polynomial. By $[n]$, we denote the set $\{1, ..., n\}$. The expectation of a random variable $X$ is denoted by $E[X]$.

### 7.1.1 Network Model

We assume a peer-to-peer network. Specifically, parties do not communicate via point-to-point connections, but rather use a variant of the *diffuse* functionality defined in [GKL15], described as follows.

**Diffuse Functionality.** The functionality initializes a variable *slot* to $1$, which is readable from all parties. In addition, it maintains a string $\mathsf{Receive}_{\mathcal{P}}()$ for each party $\mathcal{P}$. Each party $\mathcal{P}$ is allowed to fetch the contents of $\mathsf{Receive}_{\mathcal{P}}()$ at the beginning of each time slot. To diffuse a (possibly empty) message $m$, $\mathcal{P}$ sends to the functionality $m$ and an integer index $i_m$, which indicates the message delivery priority of $m$ (see below). In turn, the functionality records $m$ and $i_m$. On each slot, every party completes its activity by sending a special Complete message to the functionality. When all parties submit Complete, the functionality delivers the messages, which are diffused during this slot, as follows. First, it groups all messages based on their associated index and sorts them, with messages with smaller index having higher priority. Then, it randomizes the order of each group's messages, i.e., which are associated with the same index. Therefore, eventually all messages are sorted in a well-defined order. Subsequently, the functionality includes all messages in the $\mathsf{Receive}_{\mathcal{P}}()$ string of every party $\mathcal{P} \in \mathbb{P}$, following the aforementioned order. Hence, all parties receive all diffused messages, without any information on each message's creator. Finally, the functionality increases the value of *slot* by $1$.

**Lossy Diffuse Functionality.** The lossy diffuse functionality is similar to the above variant, with one difference: it is parameterized with a probability $d$, which defines the probability that a message $m$ is dropped, i.e., it is not delivered to any recipient. This

functionality aims to model the setting where a network with stochastic delays is used by an application, where users reject messages which are delivered with delay above a (protocol-specific) limit. For example, various protocols, like Bitcoin [Nak08a], resolve message conflicts based on the order of delivery; thus, delaying a message for long enough, such that a competing message is delivered beforehand, is equivalent to dropping the message altogether.

### 7.1.2 Approximate Nash Equilibrium

An approximate Nash equilibrium is a common tool for expressing a solution to a non-cooperative game involving $n$ parties $\mathcal{P}_1, ..., \mathcal{P}_n$. Each party $\mathcal{P}_i$ employs a strategy $S_i$. The strategy is a set of rules and actions the party makes, depending on what has happened up to any point in the game, i.e., it defines the part of the entire distributed protocol $\Pi$ performed by $\mathcal{P}_i$. There exists an "honest" strategy, defined by $\Pi$, which parties may employ; for ease of notation, $\Pi$ denotes both the distributed protocol and the honest strategy. A *strategy profile* is a vector of all players' strategies.

Each party $\mathcal{P}_i$ has a game *utility* $U_i$, which is a real function that takes as input a strategy profile. A strategy profile is an $\epsilon$-Nash equilibrium when no party can increase its utility more than $\epsilon$ by *unilaterally* changing its strategy (Definition 28).

**Definition 28.** *Let $\epsilon$ be a non-negative real number and $\mathbb{S}$ be the set of all strategies a party may employ. Also let $\sigma^* = (S_i^*, S_{-i}^*)$ be a strategy profile of $\mathbb{P}$, where $S_i^*$ is the strategy followed by $\mathcal{P}_i$ and $S_{-i}^*$ denotes the $n-1$ strategies employed by all parties except $\mathcal{P}_i$. We say that $\sigma^*$ is an $\epsilon$-Nash equilibrium w.r.t. a utility vector $\bar{U} = \langle U_1, ..., U_n \rangle$ if:*

$$\forall \mathcal{P}_i \in \mathbb{P} \; \forall S_i \in \mathbb{S} \;\; \{S_i^*\} : U_i(S_i^*, S_{-i}^*) \geq U_i(S_i, S_{-i}^*) - \epsilon$$

## 7.2 Compliance Model

We assume a distributed protocol $\Pi$, which is performed by a set of parties $\mathbb{P}$ over a number of time slots. In our analysis, each party $\mathcal{P} \in \mathbb{P}$ is associated with a number $\mu_{\mathcal{P}} \in [0, 1]$. $\mu_{\mathcal{P}}$ identifies $\mathcal{P}$'s percentage of participation power in the protocol, e.g., its votes, hashing power, staking power, *etc.* ; consequently, $\sum_{\mathcal{P} \in \mathbb{P}} \mu_{\mathcal{P}} = 1$.

### 7.2.1 Basic Notions

A protocol's execution $\mathcal{E}_{\mathcal{Z},\sigma,r}$ at a given time slot $r$ is probabilistic and parameterized by the environment $\mathcal{Z}$ and the strategy profile $\sigma$ of the participating parties. As discussed, $\mathcal{Z}$ provides the parties with inputs and schedules their activation. For notation simplicity, when $r$ is omitted, $\mathcal{E}_{\mathcal{Z},\sigma}$ refers to the end of the execution, which occurs after polynomially many time slots.

An *execution trace* $\mathfrak{I}_{\mathcal{Z},\sigma,r}$ on a time slot $r$ is the value that the random variable $\mathcal{E}_{\mathcal{Z},\sigma,r}$ takes for a fixed environment $\mathcal{Z}$ and strategy profile $\sigma$, and for fixed random coins of $\mathcal{Z}$, each party $\mathcal{P} \in \mathbb{P}$, and every protocol-specific oracle (see below). A party $\mathcal{P}$'s view of an execution trace $\mathfrak{I}^{\mathcal{P}}_{\mathcal{Z},\sigma,r}$ consists of the messages that $\mathcal{P}$ has sent and received until slot $r$. For notation simplicity, we will omit the subscripts $\{\mathcal{Z},\sigma,r\}$ from both $\mathcal{E}$ and $\mathfrak{I}$, unless required for clarity.

The protocol $\Pi$ defines two components, which are related to our analysis: (1) the oracle $\mathcal{O}_{\Pi}$, and (2) the "infraction" predicate $\mathcal{X}$. We present these two components below.

**The Oracle $\mathcal{O}_{\Pi}$.** The oracle $\mathcal{O}_{\Pi}$ provides the parties with the core functionality needed to participate in $\Pi$. For example, in a Proof-of-Work (PoW) system $\mathcal{O}_{\Pi}$ is the random or hashing oracle, whereas in an authenticated Byzantine Agreement protocol $\mathcal{O}_{\Pi}$ is a signing oracle. On each time slot, a party can perform at most a polynomial number of queries to $\mathcal{O}_{\Pi}$; in the simplest case, each party can submit a single query per slot. Finally, $\mathcal{O}_{\Pi}$ is *stateless*, i.e., its random coins are decided upon the beginning of the execution and its responses do not depend on the order of the queries.

**The Infraction Predicate $\mathcal{X}$.** The infraction predicate $\mathcal{X}$ abstracts the deviant behavior that the analysis aims to capture. Specifically, $\mathcal{X}$, when given the execution trace and a party $\mathcal{P}$, responds with $1$ only if $\mathcal{P}$ deviates from the protocol in some well-defined manner. Definition 29 provides the core generic property of $\mathcal{X}$, i.e., that honest parties never deviate. For ease of notation, we will next simply write $\mathcal{X}$ unless required for clarity. With hindsight, our analysis will focus on infraction predicates that capture either producing conflicting messages or abstaining from the protocol.

**Definition 29** (Infraction Predicate Property)**.** *The infraction predicate $\mathcal{X}$ has the property that, for every execution trace $\mathfrak{I}$ and for every party $\mathcal{P} \in \mathbb{P}$, if $\mathcal{P}$ employs the (honest) strategy $\Pi$ then $\mathcal{X}(\mathfrak{I},\mathcal{P}) = 0$.*

We stress that Definition 29 implies that $\mathcal{X}$ being $0$ is a necessary, but *not* a sufficient condition, for a party to be honest. Specifically, for all honest parties $\mathcal{X}$ is always $0$, but $\mathcal{X}$ might also be $0$ for a party that deviates from $\Pi$, in such way that is not captured by $\mathcal{X}$. In that case, we say that the party employs an $\mathcal{X}-$compliant strategy (Definition 30). A strategy profile is $\mathcal{X}-$compliant if all of its strategies are $\mathcal{X}-$compliant; naturally, the "all honest" profile $\sigma_{\Pi}$, i.e., when all parties employ $\Pi$, is – by definition – $\mathcal{X}-$compliant.

**Definition 30** (Compliant Strategy)**.** *Let $\mathcal{X}$ be an infraction predicate. A strategy $S$ is $\mathcal{X}$-compliant if and only if $\mathcal{X}(\mathfrak{I}, \mathcal{P}) = 0$ for every party $\mathcal{P}$ and for every trace $\mathfrak{I}$ where $\mathcal{P}$ employs $S$.*

**The observer $\Omega$.** We assume a special party $\Omega$, the *(passive) observer*. This party does not actively participate in the execution, but it runs $\Pi$ and observes the protocol's execution. Notably, $\Omega$ is *always online*, i.e., it bootstraps at the beginning of the execution and is activated on every slot, in order to receive diffused messages. Therefore, the observer models a user of the system, who frequently uses the system but does not actively participate in its maintenance. Additionally, at the last round of the execution, the environment $\mathcal{Z}$ activates only $\Omega$, in order to receive the diffused messages of the penultimate round and have a complete point of view.

As mentioned in Section 7.1.2, we assume a utility $U_i$ for every party $\mathcal{P}_i$, so the utility vector defines the payoff values for each party. In our examples, a party's utility depends on two parameters: i) the party's rewards, which are distributed by the protocol, from the point of view of $\Omega$; ii) the cost of participation.

### 7.2.2 Compliant Protocols

To define the notion of an $(\epsilon, \mathcal{X})$-compliant protocol $\Pi$, we require two parameters: (i) the associated infraction predicate $\mathcal{X}$ and (ii) a non-negative real number $\epsilon$. Following Definition 30, $\mathcal{X}$ determines the set of compliant strategies that the parties may follow in $\Pi$. Intuitively, $\epsilon$ specifies the sufficient gain threshold after which a party switches strategies. In particular, $\epsilon$ is used to define when a strategy profile $\sigma'$ is *directly reachable* from a strategy profile $\sigma$, in the sense that $\sigma'$ results from the unilateral deviation of a party $\mathcal{P}_i$ from $\sigma$ and, by this deviation, the utility of $\mathcal{P}_i$ increases more than $\epsilon$. Generally, $\sigma'$ is *reachable* from $\sigma$, if $\sigma'$ results from a "path" of strategy profiles, starting from $\sigma$, which are sequentially related via direct reachability. Finally, we define the *cone* of a profile $\sigma$ as the set of all strategies that are reachable from $\sigma$, including $\sigma$ itself.

Given the above definitions, we say that $\Pi$ is $(\epsilon, \mathcal{X})$-compliant if the cone of the "all honest" strategy profile $\sigma_\Pi$ contains only profiles that consist of $\mathcal{X}-$compliant strategies. Thus, if a protocol is compliant, then the parties may (unilaterally) deviate from the honest strategy only in a compliant manner, as dictated by $\mathcal{X}$.

Formally, first we define "reachability" between two strategy profiles, as well as the notion of a "cone" of a strategy profile w.r.t. the reachability relation, and then we define a compliant protocol w.r.t. its associated infraction predicate.

**Definition 31.** *Let: i) $\epsilon$ be a non-negative real number; ii) $\Pi$ be a protocol run by parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$; iii) $\bar{U} = \langle U_1, \ldots, U_n \rangle$ be a utility vector, where $U_i$ is the utility of $\mathcal{P}_i$; iv) $\mathbb{S}$ be the set of all strategies a party may employ. We provide the following definitions.*

1. *Let $\sigma, \sigma' \in \mathbb{S}^n$ be two strategy profiles where $\sigma = \langle S_1, \ldots, S_n \rangle$ and $\sigma' = \langle S'_1, \ldots, S'_n \rangle$. We say that $\sigma'$ is directly $\epsilon$-reachable from $\sigma$ w.r.t. $\bar{U}$, if there exists $i \in [n]$ s.t. (i) $\forall j \in [n] \backslash \{i\} : S_j = S'_j$ and (ii) $U_i(\sigma') > U_i(\sigma) + \epsilon$.*

2. *Let $\sigma, \sigma' \in \mathbb{S}^n$ be two distinct profiles. We say that $\sigma'$ is $\epsilon$-reachable from $\sigma$ w.r.t. $\bar{U}$, if there exist profiles $\sigma_1, \ldots, \sigma_k$ such that (i) $\sigma_1 = \sigma$, (ii) $\sigma_k = \sigma'$, and (iii) $\forall j \in [2, k]$ it holds that $\sigma_j$ is directly $\epsilon$-reachable from $\sigma_{j-1}$ w.r.t. $\bar{U}$.*

3. *For every strategy profile $\sigma \in \mathbb{S}^n$ we define the $(\epsilon, \bar{U})$-cone of $\sigma$ as the set:*

$$\mathsf{Cone}_{\epsilon,\bar{U}}(\sigma) := \{\sigma' \in \mathbb{S}^n \mid (\sigma' = \sigma) \vee (\sigma' \text{ is } \epsilon\text{-reachable from } \sigma \text{ w.r.t. } \bar{U})\} .$$

**Definition 32.** *Let: i) $\epsilon$ be a non-negative real number; ii) $\Pi$ be a protocol run by the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$; iii) $\mathcal{X}$ be an infraction predicate; iv) $\bar{U} = \langle U_1, \ldots, U_n \rangle$ be a utility vector, where $U_i$ is the utility of party $\mathcal{P}_i$; v) $\mathbb{S}$ be the set of all strategies a party may employ; vi) $\mathbb{S}_\mathcal{X}$ be the set of $\mathcal{X}-$*compliant *strategies.*

*A strategy profile $\sigma \in \mathbb{S}^n$ is $\mathcal{X}-$*compliant *if $\sigma \in (\mathbb{S}_\mathcal{X})^n$.*

*The $(\epsilon, \bar{U})$-cone of $\Pi$, denoted by $\mathsf{Cone}_{\epsilon,\bar{U}}(\Pi)$, is the set $\mathsf{Cone}_{\epsilon,\bar{U}}(\sigma_\Pi)$, i.e., the set of all strategies that are $\epsilon$-reachable from the "all honest" strategy profile $\sigma_\Pi = \langle \Pi, \ldots, \Pi \rangle$ w.r.t. $\bar{U}$, also including $\sigma_\Pi$.*

*$\Pi$ is $(\epsilon, \mathcal{X})$-compliant w.r.t. $\bar{U}$ if $\mathsf{Cone}_{\epsilon,\bar{U}}(\Pi) \subseteq (\mathbb{S}_\mathcal{X})^n$, i.e., all strategy profiles in the $(\epsilon, \bar{U})$-cone of $\Pi$ are $\mathcal{X}-$*compliant.

Proposition 1 shows that Definition 32 expresses a relaxation of the standard approximation Nash equilibrium (Definition 28).

**Proposition 1.** *Let: i) $\epsilon$ be a non-negative real number; ii) $\Pi$ be a protocol run by the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$; iii) $\mathcal{X}$ be an infraction predicate; iv) $\bar{U} = \langle U_1, \ldots, U_n \rangle$ be a utility vector, with $U_i$ the utility of $\mathcal{P}_i$. If $\Pi$ is an $\epsilon$-Nash equilibrium w.r.t. $\bar{U}$ (i.e., $\sigma_\Pi = \langle \Pi, \ldots, \Pi \rangle$ is an $\epsilon$-Nash equilibrium w.r.t. $\bar{U}$), then $\Pi$ is $(\epsilon, \mathcal{X})$-compliant w.r.t. $\bar{U}$.*

*Proof.* Assume that $\sigma_\Pi$ is an $\epsilon$-Nash equilibrium w.r.t. $\bar{U}$ and let $\sigma' = (S'_1, \ldots, S'_n)$ be a strategy profile. We will show that $\sigma'$ is not directly $\epsilon$-reachable from $\sigma_\Pi$ w.r.t. $\bar{U}$.

Assume that there exists $i \in [n]$ s.t. $\forall j \in [n] \backslash \{i\} : S'_j = \Pi$. Since $\sigma_\Pi$ is an $\epsilon$-Nash equilibrium w.r.t. $\bar{U}$, it holds that $U_i(\sigma') \leq U_i(\sigma_\Pi) + \epsilon$. Therefore, Definition 31 is not satisfied and $\sigma'$ is not directly $\epsilon$-reachable from $\sigma_\Pi$ w.r.t. $\bar{U}$.

Since no profiles are directly $\epsilon$-reachable from $\sigma_\Pi$ w.r.t. $\bar{U}$, it is straightforward that there are no $\epsilon$-reachable strategy profiles from $\sigma_\Pi$ w.r.t. $\bar{U}$. The latter implies that the $(\epsilon, \bar{U})$-cone of $\Pi$ contains only $\sigma_\Pi$, i.e., $\text{Cone}_{\epsilon,\bar{U}}(\Pi) = \{\sigma_\Pi\}$. By Definitions 29 and 30, $\sigma_\Pi$ is $\mathcal{X}-$compliant, so we deduce that $\Pi$ is $(\epsilon, \mathcal{X})$-compliant w.r.t. $\bar{U}$. □

## 7.3 Blockchain Protocols

In this work, we focus on blockchain-based distributed ledger protocols. In the general case, a ledger defines a global state, which is distributed across multiple parties and is maintained via a consensus protocol. The distributed ledger protocol defines the validity rules which allow a party to extract the final ledger from its view. A blockchain is a distributed database, where each message $m$ is a block $\mathcal{B}$ of transactions and each transaction updates the system's global state. Therefore, at any point of the execution, a party $\mathcal{P}$ holds some view of the global state, which comprises of the blocks that $\mathcal{P}$ has adopted. We note that, if at least one valid block is diffused (w.r.t. the validity rules of the protocol), then every honest party can extract a final ledger from its execution view.

### 7.3.1 The Setting

Every blockchain protocol $\Pi$ defines a *message validity* predicate $\mathcal{V}$. A party $\mathcal{P}$ accepts a block $\mathcal{B}$, received during a slot $r$ of the execution, if it holds that $\mathcal{V}(\mathfrak{I}_r^{\mathcal{P}}, \mathcal{B}) = 1$. For example, in Proof-of-WorK (PoW) systems like Bitcoin [Nak08a], a block is valid if its hash is below a certain threshold, while in Proof-of-Stake (PoS) protocols like Ouroboros [KRDO17], a block is valid if it has been created by a specific party, according to a well-known leader schedule. In all cases, a block $\mathcal{B}$ is valid only if the party that creates it submits a query for $\mathcal{B}$ to the oracle $\mathcal{O}_\Pi$ beforehand.

Each block $\mathcal{B}$ is associated with the following metadata: i) an index *index*$(\mathcal{B})$; ii) the party *creator*$(\mathcal{B})$ that created $\mathcal{B}$; iii) a set *ancestors*$(\mathcal{B}) \subseteq \mathfrak{I}^{\textit{creator}(\mathcal{B})}$, i.e., blocks in the view of *creator*$(\mathcal{B})$ (at the time of $\mathcal{B}$'s creation) referenced by $\mathcal{B}$.
Message references are implemented as hash pointers, given a hash function H employed by the protocol. Specifically, each block $\mathcal{B}$ contains the hash of all blocks in the referenced blocks *ancestors*$(\mathcal{B})$. Blockchain systems are typically bootstrapped via a global common reference string, i.e., a "genesis" block $\mathcal{B}_G$. Therefore, the blocks form a hash tree, stemming from $\mathcal{B}_G$. *index*$(\mathcal{B})$ is the height of $\mathcal{B}$ in the hash tree. If $\mathcal{B}$ references multiple messages, i.e., belongs to multiple tree branches, *index*$(\mathcal{B})$ is the height of the longest one.

The protocol also defines the *message equivalency operator*, $\equiv$. Specifically, two messages are equivalent if their hashes match, i.e., $m_1 \equiv m_2 \Leftrightarrow \mathsf{H}(m_1) = \mathsf{H}(m_2)$. At a high level, two equivalent messages are interchangeable by the protocol.

**Infraction Predicate.** In our analysis of blockchain systems, we consider two types of deviant behavior (Definition 33): i) creating conflicting valid messages, ii) abstaining. We choose these predicates because they may lead to non-compliance in interesting use cases. For the former, conflicting valid messages, i.e., chain forks, are the primary reason for failure to achieve consensus, so exploring when a party is incentivized to initiate a fork is particularly interesting. For the latter, increased and continuous participation typically helps the safety of deployed systems, as motivating more users to participate in a compliant manner increases the level of power that an adversary needs to reach in order to break a system's security.

**Definition 33** (Blockchain Infraction Predicate)**.** *Given a party $\mathcal{P}$ and an execution trace $\mathfrak{I}_{\mathcal{Z},\sigma,r}$, we define the following infraction predicates:*

1. *conflicting predicate:* $\mathcal{X}_{conf}(\mathfrak{I}_{\mathcal{Z},\sigma,r}, \mathcal{P}) = 1$ *if* $\exists \mathcal{B}, \mathcal{B}' \in \mathfrak{I}_{\mathcal{Z},\sigma,r} : \mathcal{V}(\mathfrak{I}^{\mathcal{P}}_{\mathcal{Z},\sigma,r}, \mathcal{B}) = \mathcal{V}(\mathfrak{I}^{\mathcal{P}}_{\mathcal{Z},\sigma,r}, \mathcal{B}') = 1 \wedge \textit{index}(\mathcal{B}) = \textit{index}(\mathcal{B}') \wedge \mathcal{B} \not\equiv \mathcal{B}' \wedge \textit{creator}(\mathcal{B}) = \textit{creator}(\mathcal{B}') = \mathcal{P}$*;*

2. *abstaining predicate:* $\mathcal{X}_{abs}(\mathfrak{I}_{\mathcal{Z},\sigma,r}, \mathcal{P}) = 1$ *if $\mathcal{P}$ makes no queries to the oracle $\mathcal{O}_{\Pi}$ during slot $r$;*

3. *blockchain predicate:* $\mathcal{X}_{bc}(\mathfrak{I}_{\mathcal{Z},\sigma,r}, \mathcal{P}) = 1$ *if it holds* $(\mathcal{X}_{conf}(\mathfrak{I}_{\mathcal{Z},\sigma,r}, \mathcal{P}) = 1) \vee (\mathcal{X}_{abs}(\mathfrak{I}_{\mathcal{Z},\sigma,r}, \mathcal{P}) = 1)$.

**Remark.** *Preventing conflicting messages is not the same as resilience against sybil attacks [Dou02]. Sybil resilience mechanisms restrict an attacker from creating multiple identities, in order to participate in a distributed system. In comparison, our infraction predicate ensures that a user does not increase their utility by violating the infraction conditions for each identity they control. Therefore, a system may be compliant but not sybil resilient, e.g., if a party can participate via multiple identities but it cannot increase its utility by producing conflicting messages, and vice versa.*

Finally, at the end of the execution, the observer $\Omega$ outputs a chain $\mathcal{C}_{\Omega,\mathfrak{I}}$. Typically, this is the longest valid chain, i.e., the longest branch of the tree that stems from genesis $\mathcal{B}_G$.[1] In case multiple longest chains exist, a choice is made either at random or following a chronological ordering of messages. The number of messages in $\mathcal{C}_{\Omega,\mathfrak{I}}$ that are created by a party $\mathcal{P}$ is denoted by $M_{\mathcal{P},\mathfrak{I}}$.

### 7.3.2 Utility: Rewards and Costs

For each execution, the blockchain protocol defines a number of total rewards, which are distributed among the participating parties. For each party $\mathcal{P}$, these rewards are expressed via the *reward random variable* $R_{\mathcal{P},\mathcal{E}}$. For a specific trace $\mathfrak{I}$, the random variable takes a non-negative real value, denoted by $R_{\mathcal{P},\mathfrak{I}}$. Intuitively, $R_{\mathcal{P},\mathfrak{I}}$ describes the rewards that $\mathcal{P}$ receives from the protocol from the point of view of the observer $\Omega$, i.e., w.r.t. the blocks accepted by $\Omega$.

Our analysis restricts to systems where rewards are distributed to parties if and only if the genesis block is extended by at least one block during the execution, in which case at least one party receives a non-negative amount of rewards (Assumption 1).

**Assumption 1.** *Let $\mathfrak{I}$ be an execution trace. If no block is produced during $\mathfrak{I}$, then it holds that $\forall \mathcal{P} \in \mathbb{P} : R_{\mathcal{P},\mathfrak{I}} = 0$. If at least one block is produced during $\mathfrak{I}$, then it holds that $\exists \mathcal{P} \in \mathbb{P} : R_{\mathcal{P},\mathfrak{I}} \neq 0$.*

In addition to rewards, a party's utility is affected by cost. Specifically, the *cost random variable* $C_{\mathcal{P},\mathcal{E}}$ expresses the operational cost that $\mathcal{P}$ during an execution $\mathcal{E}$. For a fixed trace $\mathfrak{I}$, $C_{\mathcal{P},\mathfrak{I}}$ is a non-negative real value. Our analysis is restricted to cost schemes which are *linearly monotonically increasing* in the number of queries that a party makes to the oracle $\mathcal{O}_\Pi$, with no queries incurring zero cost (Assumption 2). Intuitively, this

[1] For simplicity we assume that the longest chain (in terms of blocks) also contains the most hashing power, which is the metric used in PoW systems like Bitcoin.

assumption considers the electricity cost of participation, while the cost of equipment and other operations, such as parsing or publishing messages, is zero.

**Assumption 2.** *For every execution trace $\mathfrak{I}$, a party $\mathcal{P}$'s cost is $C_{\mathcal{P},\mathfrak{I}} = 0$ if and only if it performs no queries to $\mathcal{O}_{\Pi}$ in every time slot. Else, if during $\mathfrak{I}$ a party $\mathcal{P}$ performs $t$ queries, then its cost is $C_{\mathcal{P},\mathfrak{I}} = t \cdot \lambda$, for some fixed parameter $\lambda$.*

Next, we define two types of utility. The first type, *Reward*, considers the expected rewards that a party receives when the cost is near $0$, while the second type, *Profit*, considers rewards minus participation cost.

**Definition 34.** *Let $\sigma$ be a strategy profile and $\mathcal{E}_{\sigma}$ be an execution during which parties follow $\sigma$. We define two types of blockchain utility $U_{\mathcal{P}}$ of a party $\mathcal{P}$ for $\sigma$:*

1. *Reward:* $U_{\mathcal{P}}(\sigma) = E[R_{\mathcal{P},\mathcal{E}_{\sigma}}]$
2. *Profit:* $U_{\mathcal{P}}(\sigma) = E[R_{\mathcal{P},\mathcal{E}_{\sigma}}] - E[C_{\mathcal{P},\mathcal{E}_{\sigma}}]$

For the computation of $U_{\mathcal{P}}$, the environment $\mathcal{Z}$ is fixed, while the expectation of the random variables $R_{\mathcal{P},\mathcal{E}_{\sigma}}$ and $C_{\mathcal{P},\mathcal{E}_{\sigma}}$ is computed over the random coins of $\mathcal{Z}$, $\mathcal{O}_{\Pi}$, and every party $\mathcal{P} \in \mathbb{P}$.

In the following sections, we evaluate the compliance of various Proof-of-Work (PoW) and Proof-of-Stake (PoS) blockchain protocols w.r.t. two types of rewards, *fair* and *block-proportional*.

## 7.4 Fair Rewards

As described in Section 7.2, each party $\mathcal{P}$ controls a percentage $\mu_{\mathcal{P}}$ of the system's participating power. Although this parameter is set at the beginning of the execution, it is not always public. For instance, a party could obscure its amount of hashing power by refraining from performing some queries. In other cases, each party's power is published on the distributed ledger and, for all executions, can be extracted from the observer's chain. A prime example of such systems is non-anonymous PoS ledgers, where each party's power is denoted by its assets, which are logged in real time on the ledger.

These systems, where power distribution is public, can employ a special type of rewards, *fair rewards*.[2] Specifically, the system defines a fixed, total number of rewards $\mathcal{R} > 0$. At the end of an execution, if at least one block is created, each party $\mathcal{P}$ receives

[2]The term *fairness* is an allusion to Fruitchains [PS17a].

a percentage $\xi(\mu_{\mathcal{P}})$ of $\mathcal{R}$, where $\xi(\cdot) : [0,1] \to [0,1]$; in the real world, $\xi$ is usually the identity function. If no blocks are created during the execution, then every party gets $0$ rewards.

Intuitively, fair rewards compensate users for investing in the system. Unless no block is created (which typically happens with very small probability when the parties honestly follow the protocol), the level of rewards depends *solely* on a party's power, and not in the messages diffused during the execution. Definition 35 formally describes the family of fair reward random variables.

**Definition 35** (Fair Rewards)**.** *For a total number of rewards $\mathcal{R} \in \mathbb{R}_{>0}$, a fair reward random variable $R_{\mathcal{P},\mathcal{E}}$ satisfies the following:*

1. $\forall \mathfrak{I}\, \forall \mathcal{P} \in \mathbb{P} : R_{\mathcal{P},\mathfrak{I}} = \begin{cases} \xi(\mu_{\mathcal{P}}) \cdot \mathcal{R}, & \text{if at least one valid block is produced during } \mathfrak{I} \\ 0, & \text{otherwise} \end{cases}$
2. $\sum_{\mathcal{P} \in \mathbb{P}} \xi(\mu_{\mathcal{P}}) = 1$

*where $\xi : [0,1] \to [0,1]$.*

As shown in Theorem 6, blockchains with fair rewards are $(\epsilon, \mathcal{X})$-compliant for the utility Reward (cf. Definition 34), where $\epsilon$ is typically a small value and $\mathcal{X}$ is an arbitrary associated infraction predicate. Intuitively, since a party is rewarded the same amount regardless of their protocol-related actions, nobody can increase their rewards by deviating from the honest strategy (as long as at least one block is produced).

**Theorem 6.** *Let: i) $\Pi$ be a blockchain protocol run by the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$; ii) $\mathcal{X}$ be any infraction predicate associated with $\Pi$; iii) $\bar{U} = \langle U_1, \ldots, U_n \rangle$ be a utility vector, where $U_i$ is the utility Reward of party $\mathcal{P}_i$; iv) $\mathcal{R}$ be the total rewards distributed by the protocol; v) $\xi : [0,1] \to [0,1]$ be a fair reward function; vi) $\alpha$ be the probability that no blocks are produced when all parties follow the honest strategy. Then, $\Pi$ is $(\epsilon, \mathcal{X})$-compliant w.r.t. $\bar{U}$, for $\epsilon := \alpha \cdot \max_{j \in [n]} \{\xi(\mu_{\mathcal{P}_j}) \cdot \mathcal{R}\}$.*

*Proof.* By Definition 35 and the definition of $\alpha$, for the "all honest" strategy profile $\sigma_\Pi := \langle \Pi, \ldots, \Pi \rangle$, we have that $\Pr[R_{\mathcal{P}_i, \mathcal{E}_{\sigma_\Pi}} = \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R}] = 1 - \alpha$ and $\Pr[R_{\mathcal{P}_i, \mathcal{E}_{\sigma_\Pi}} = 0] = \alpha$, for every $i \in [n]$. Therefore, for every $i \in [n]$, $U_i(\sigma_\Pi) = E[R_{\mathcal{P}_i, \mathcal{E}_{\sigma_\mathcal{P}}}] = (1-\alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R}$.

Assume that for some $i \in [n]$, $\mathcal{P}_i$ unilaterally deviates from $\Pi$ by employing a different strategy $S_i$. In this case, we consider the strategy profile $\sigma = \langle S_1, \ldots, S_n \rangle$ where $S_j = \Pi$ for $j \in [n] \backslash \{i\}$. Since $U_i$ is the utility *Reward* under fair rewards with $\mathcal{R}, \xi(\cdot)$, we have that for all random coins of the execution $\mathcal{E}_\sigma$, the value of the reward random

variable $R_{\mathcal{P}_i,\mathcal{E}_\sigma}$ is no more than $\xi(\mu_{\mathcal{P}_i})\cdot\mathcal{R}$. Consequently, $U_i(\sigma)\leq\xi(\mu_{\mathcal{P}_i})\cdot\mathcal{R}$, and so we have that

$$U_i(\sigma)\leq U_i(\sigma_\Pi)+\alpha\cdot\xi(\mu_{\mathcal{P}_i})\cdot\mathcal{R}\leq U_i(\sigma_\Pi)+\alpha\cdot\max_{j\in[n]}\{\xi(\mu_{\mathcal{P}_j})\cdot\mathcal{R}\}\ .$$

If $\epsilon:=\alpha\cdot\max_{j\in[n]}\{\xi(\mu_{\mathcal{P}_j})\cdot\mathcal{R}\}$ and since $i$ and $S_i$ are arbitrary, we conclude that $\Pi$ is a $\epsilon$-Nash equilibrium w.r.t. $\bar{U}$. Thus, by Proposition 1, $\Pi$ is $(\epsilon,\mathcal{X})$-compliant w.r.t. $\bar{U}$. □

Theorem 6 is consistent with the incentives' analysis of [KRDO17] under fair rewards. However, when introducing operational costs to analyze profit, a problem arises: a user can simply abstain and be rewarded nonetheless. Such behavior results in a "free-rider problem" [Bau04], where a user reaps some benefits while not under-paying them or not paying at all. Theorem 7 formalizes this argument and shows that a blockchain protocol, associated with the abstaining infraction predicate $\mathcal{X}_{abs}$ (cf. Definition 33), under fair rewards is *not* $(\epsilon,\mathcal{X}_{abs})$-compliant w.r.t. utility *Profit*, for reasonable values of $\epsilon$.

**Theorem 7.** *Let: i) $\Pi$ be a blockchain protocol run by the parties $\mathcal{P}_1,\ldots,\mathcal{P}_n$; ii) $\mathcal{X}_{abs}$ be the abstaining infraction predicate; iii) $\bar{U}=\langle U_1,\ldots,U_n\rangle$ be a utility vector, where $U_i$ is the utility Profit of party $\mathcal{P}_i$; iv) $\mathcal{R}$ be the total rewards distributed by the protocol; v) $\xi:[0,1]\to[0,1]$ be a fair reward function; vi) $\alpha$ be the probability that no blocks are produced when all parties follow the honest strategy.*

*For $i\in[n]$, also let the following: i) $C^{\perp}_{\mathcal{P}_i}$ be the minimum cost of $\mathcal{P}_i$ across all possible execution traces where $\mathcal{P}_i$ employs $\Pi$; ii) $\beta_i$ be the probability that no blocks are produced when $\mathcal{P}_i$ abstains throughout the entire execution and all the other parties follow $\Pi$.*

*Then, for every $\epsilon\geq 0$ such that $\epsilon<\max_{i\in[n]}\{C^{\perp}_{\mathcal{P}_i}-(\beta_i-\alpha)\cdot\xi(\mu_{\mathcal{P}_i})\cdot\mathcal{R}\}$, the protocol $\Pi$ is not $(\epsilon,\mathcal{X}_{abs})$-compliant w.r.t. $\bar{U}$.*

*Proof.* By Definition 35 and the definition of $\alpha$, for the "all honest" strategy profile $\sigma_\Pi:=\langle\Pi,\ldots,\Pi\rangle$, we have that $\Pr[R_{\mathcal{P}_i,\mathcal{E}_{\sigma_\Pi}}=\xi(\mu_{\mathcal{P}_i})\cdot\mathcal{R}]=1-\alpha$ and $\Pr[R_{\mathcal{P}_i,\mathcal{E}_{\sigma_\Pi}}=0]=\alpha$, for every $i\in[n]$. Since $\Pi$ is an $\mathcal{X}_{abs}$-compliant strategy, if $\mathcal{P}_i$ follows $\Pi$ then it does not abstain, i.e., it makes queries to $\mathcal{O}_\Pi$. Therefore, the cost of $\mathcal{P}_i$ is at least $C^{\perp}_{\mathcal{P}_i}$ and for $\sigma_\Pi$, the utility *Profit* $U_i(\sigma_\Pi)$ is no more than

$$U_i(\sigma_\Pi)=E[R_{\mathcal{P}_i,\mathcal{E}_{\sigma_\Pi}}]-E[C_{\mathcal{P}_i,\mathcal{E}_{\sigma_\Pi}}]\leq(1-\alpha)\cdot\xi(\mu_{\mathcal{P}_i})\cdot\mathcal{R}-C^{\perp}_{\mathcal{P}_i}\ .$$

Now assume that $\mathcal{P}_i$ unilaterally deviates by following the "always abstain" strategy, $S_{\mathsf{abs}}$, which is of course not $\mathcal{X}_{abs}$-compliant. Then, $\mathcal{P}_i$ makes no queries to $\mathcal{O}_\Pi$ and,

by Assumption 2, its cost is $0$. Let $\sigma_i$ be the strategy profile where $\mathcal{P}_i$ follows $S_{\mathsf{abs}}$ and every party $\mathcal{P} \neq \mathcal{P}_i$ follows $\Pi$. By the definition of $\beta_i$, we have that $\Pr[R_{\mathcal{P}_i,\mathcal{E}_{\sigma_i}} = \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R}] = 1 - \beta_i$ and $\Pr[R_{\mathcal{P}_i,\mathcal{E}_{\sigma_i}} = 0] = \beta_i$.

By the definition of $U_i$, it holds that

$$U_i(\sigma_i) = E[R_{\mathcal{P}_i,\mathcal{E}_{\sigma_i}}] - E[C_{\mathcal{P}_i,\mathcal{E}_{\sigma_i}}] = (1-\beta_i) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R} - 0 \,.$$

Assume that $C^{\perp}_{\mathcal{P}_i} - (\beta_i - \alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R} > 0$. Then, for $\epsilon_i < C^{\perp}_{\mathcal{P}_i} - (\beta_i - \alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R}$, we have that

$$\begin{aligned}
U_i(\sigma_i) &= (1-\beta_i) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R} = \\
&= (1-\alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R} + (\alpha - \beta_i) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R} = \\
&= (1-\alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R} - C^{\perp}_{\mathcal{P}_i} + C^{\perp}_{\mathcal{P}_i} - (\beta_i - \alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R} \geq \\
&\geq U_i(\sigma_{\Pi}) + C^{\perp}_{\mathcal{P}_i} - (\beta_i - \alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R} > \\
&> U_i(\sigma_{\Pi}) + \epsilon_i
\end{aligned}$$

Let $i^* \in [n]$ be such that $C^{\perp}_{\mathcal{P}_{i^*}} - (\beta_{i^*} - \alpha) \cdot \xi(\mu_{\mathcal{P}_{i^*}}) \cdot \mathcal{R}\} = \max_{i \in [n]} \{C^{\perp}_{\mathcal{P}_i} - (\beta_i - \alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R}\}$. By the above, for every $0 \leq \epsilon < C^{\perp}_{\mathcal{P}_{i^*}} - (\beta_{i^*} - \alpha) \cdot \xi(\mu_{\mathcal{P}_{i^*}}) \cdot \mathcal{R}\}$, we have that $\sigma_{i^*}$ is directly $\epsilon$-reachable from $\sigma_{\Pi}$ w.r.t. $\bar{U}$.

Thus, $\sigma_{i^*}$ is a non $\mathcal{X}_{abs}$-compliant strategy profile that is in $\mathsf{Cone}_{\epsilon,\bar{U}}(\Pi)$. Consequently, for $0 \leq \epsilon < \max_{i \in [n]} \{C^{\perp}_{\mathcal{P}_i} - (\beta_i - \alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R}\}$, it holds that $\mathsf{Cone}_{\epsilon,\bar{U}}(\Pi) \subsetneq (\mathbb{S}_{\mathcal{X}_{abs}})^n$, i.e., the protocol $\Pi$ is not $(\epsilon, \mathcal{X}_{abs})$-compliant w.r.t. $\bar{U}$. □

Before concluding, let us examine the variables of the bound $\max_{i \in [n]} \{C^{\perp}_{\mathcal{P}_i} - (\beta_i - \alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R}\}$ of Theorem 7. We note that, in the context of blockchain systems, a "party" is equivalent to a unit of power; therefore, a party $\mathcal{P}$ that controls $\mu_{\mathcal{P}}$ of the total power, in effect controls $\mu_{\mathcal{P}}$ of all "parties" that participate in the blockchain protocol.

To discuss $\alpha$ and $\beta_i$, we first consider the liveness property [GK20b] of blockchain protocols. Briefly, if a protocol guarantees liveness with parameter $u$, then a transaction which is diffused on slot $r$ is part of the (finalized) ledger of every honest party on round $r + u$. Therefore, assuming that the environment gives at least one transaction to the parties, if a protocol $\Pi$ guarantees liveness unless with negligible probability $\mathsf{negl}(\kappa)$,[3] then at least one block is created during the execution with overwhelming probability (in $\kappa$).

Now, we consider the probabilities $\alpha$ and $\beta_i$. The former is negligible, since consensus protocols typically guarantee liveness against a number of crash (or Byzantine) faults,

[3] Recall that $\kappa$ is $\Pi$'s security parameter, while $\mathsf{negl}(\cdot)$ is a negligible function.

let alone if all parties are honest. The latter, however, depends on $\mathcal{P}_i$'s percentage of power $\mu_{\mathcal{P}_i}$. For instance, consider Ouroboros, which is secure if a deviating party $\mathcal{P}_i$ controls less than $\frac{1}{2}$ of the staking power and all others employ $\Pi$. Thus, if $\mu_{\mathcal{P}_i} = \frac{2}{3}$ and $\mathcal{P}_i$ abstains, the protocol cannot guarantee liveness, i.e., it is probable that no blocks are created. However, if $\mu_{\mathcal{P}_i} = \frac{1}{4}$, then liveness is guaranteed with overwhelming probability; hence, even if $\mathcal{P}_i$ abstains, at least one block is typically created. Corollary 2 generalizes this argument, by showing that, if enough parties participate, then at least one of them is small enough, such that its abstaining does not result in a system halt, hence it is incentivized to be non-compliant.

**Corollary 2.** *Let $\Pi$ be a blockchain protocol, with security parameter $\kappa$, which is run by $n$ parties, under the same considerations of Theorem 7. Additionally, assume that $\Pi$ has liveness with security threshold $\frac{1}{x}$ in the following sense: for every strategy profile $\sigma$, if $\sum_{\mathcal{P}\in\mathbb{P}_{-\sigma}} \mu_{\mathcal{P}} < \frac{1}{x}$, where $\mathbb{P}_{-\sigma}$ is the set of parties that deviate from $\Pi$ when $\sigma$ is followed, then $\Pi$ guarantees liveness with overwhelming (i.e., $1 - \mathsf{negl}(\kappa)$) probability.*

*If $x < n$, then for (non-negligible) values $\epsilon < \max_{i\in[n]}\{C^{\perp}_{\mathcal{P}_i}\} - \mathsf{negl}(\kappa)$, $\Pi$ is not $(\epsilon, \mathcal{X}_{abs})$-compliant w.r.t. $\bar{U}$.*

*Proof.* First, since $\Pi$ guarantees liveness even under some byzantine faults, $\alpha = \mathsf{negl}(\kappa)$.

Second, if $x < n$, then there exists $i \in [n]$ such that $\mu_{\mathcal{P}_i} < \frac{1}{x}$. To prove this, if $n > x$ and $\forall j \in [n] : \mu_{\mathcal{P}_j} \geq \frac{1}{x}$ then $\sum_{j\in[n]} \mu_{\mathcal{P}_i} \geq \frac{n}{x} > 1$. This contradicts to the definition of the parties' participating power (cf. Section 7.2), where it holds that $\sum_{j\in[n]} \mu_{\mathcal{P}_j} = 1$.

Now consider the strategy profile $\sigma$ where $\mathcal{P}_i$ abstains and all the other parties honestly follow $\Pi$. Then, by definition, $\mathbb{P}_{-\sigma} = \{\mathcal{P}_i\}$ and therefore,

$$\sum_{\mathcal{P}\in\mathbb{P}_{-\sigma}} \mu_{\mathcal{P}} = \mu_{\mathcal{P}_i} < \frac{1}{x}\,.$$

Thus, by the assumption for $\Pi$, we have that if the parties follow $\sigma$, then $\Pi$ guarantees liveness with $1 - \mathsf{negl}(\kappa)$ probability. Hence, $\beta_i = \mathsf{negl}(\kappa)$. Finally, since $\xi(\mu_{\mathcal{P}_i}) \in [0, 1]$ and $\mathcal{R}$ is a finite value irrespective of the parties' strategy profile, the value $(\beta_i - \alpha) \cdot \xi(\mu_{\mathcal{P}_i}) \cdot \mathcal{R}$ is also negligible in $\kappa$. □

The minimal cost $C^{\perp}_{\mathcal{P}_i}$ of honest participation for $\mathcal{P}_i$ depends on the blockchain system's details. In PoW systems, where participation consists of repeatedly performing computations, cost increases with the percentage of mining power; for instance, controlling $51\%$ of Bitcoin's mining power for $1$ hour costs $\$700,000$.[4] In PoS systems, cost

[4] `https://www.crypto51.app` [May 2021]

is typically irrespective of staking power, since participation consists only of monitoring the network and regularly signing messages; for example, running a production-grade Cardano node costs $\$140$ per month[5]. Therefore, taking Corollary 2 into account, the upper bound $\max_{i\in[n]}\{C^{\perp}_{\mathcal{P}_i}\} - \mathsf{negl}(\kappa)$ of $\epsilon$ is typically rather large for PoS systems.

The free-rider hazard is manifested in Algorand[6], a cryptocurrency system that follows the Algorand consensus protocol [CGMV18, GHM+17b] and employs fair rewards, as defined above. Its users own "Algo" tokens and transact over a ledger maintained by "participation nodes", which run the Algorand protocol and extend the ledger via blocks. Each user receives a fixed reward[7] per Algo token they own [Fou20], awarded with every block. Users may run a participation node, but are not rewarded [FMJR20] for doing so, and participation is proportional to the amount of Algos that the user owns. Therefore, a party that owns a few Algos will expectedly abstain from participation in the consensus protocol.

In conclusion, fair rewards in PoS protocols may incentivize users to abstain. In turn, this can impact the system's performance, e.g., delaying block production and transaction finalization. In the extreme case, when multiple parties can change their strategy simultaneously, it could result in a "tragedy of the commons" situation [Llo33], where *all* users abstain and the system grinds to a halt.

**Remark.** *This section illustrates a difference between PoW and PoS. In PoS systems, each party's power is registered on the ledger (in the form of assets), without requiring any action from each party. In PoW, a party's power becomes evident only after the party puts their hardware to work. This is demonstrated in Fruitchains [PS17a], a PoW protocol where each party receives a reward (approximately) proportional to their mining power. However, the PoW protocol cannot identify each party's power, unless the party participates and produces some blocks (or "fruits"). Therefore, since the proof of Theorem 7 relies on abstaining, this result does not directly translate to Fruitchains, or similar PoW protocols.*

## 7.5 Block-Proportional Rewards

The arguably most common type of rewards in blockchain systems is *block-proportional* rewards. Specifically, each party is rewarded proportionally to the number of blocks

[5] https://forum.cardano.org/t/realistic-cost-to-operate-stake-pool/40056 [September 2020]

[6] `https://algorand.foundation`

[7] For a weekly execution, the reward per owned Algo is $0.001094$ Algos. [`https://algoexplorer.io/rewards-calculator` [October 2020]

that it contributes to the final chain, which is output at the end of the execution.

Block-proportional rewards are a generalization of the *proportional allocation rule*, which, for example, is employed in Bitcoin. The proportional allocation rule states that a party $\mathcal{P}$'s *expected* rewards of a single block are $\mu_{\mathcal{P}}$. As shown by Chen *et al.* [CPR19], this is the unique allocation rule that satisfies a list of desirable properties, namely: i) non-negativity, ii) budget-balance, iii) symmetry, iv) sybil-proofness, and v) collusion-proofness.

Our work expands the scope by considering proportional rewards w.r.t. blocks for the entirety of the execution. Specifically, Definition 36 describes block-proportional rewards, where a party $\mathcal{P}$'s rewards are *strictly monotonically increasing* on the number of blocks that $\mathcal{P}$ contributes to the chain output by the observer $\Omega$. The definition considers a *proportional reward function* $\varrho(\cdot,\cdot)$ that takes as input the chain of $\Omega$ and $\mathcal{P}$ and outputs a value in $[0,1]$.

**Definition 36** (Block-Proportional Rewards)**.** *For an execution trace $\mathfrak{I}$, let $\mathcal{C}_{\Omega,\mathfrak{I}}$ be the chain output by $\Omega$ and $\mathcal{R}_{\Omega,\mathfrak{I}} \in \mathbb{R}_{\geq 0}$ be the total number of rewards which are distributed by the protocol, according to $\Omega$. Let $M_{\mathcal{P},\mathfrak{I}}$ be the number of blocks in the chain output by $\Omega$ which are produced by $\mathcal{P}$. A block-proportional reward random variable $R_{\mathcal{P},\mathcal{E}}$ satisfies the following conditions:*

1. $\forall \mathfrak{I}\ \forall \mathcal{P} \in \mathbb{P} : R_{\mathcal{P},\mathfrak{I}} = \varrho(\mathcal{C}_{\Omega,\mathfrak{I}}, \mathcal{P}) \cdot \mathcal{R}_{\Omega,\mathfrak{I}}$
2. $\forall \mathfrak{I} : \sum_{\mathcal{P}\in\mathbb{P}} \varrho(\mathcal{C}_{\Omega,\mathfrak{I}}, \mathcal{P}) = 1$
3. $\forall \mathfrak{I}\ \forall \mathcal{P}, \mathcal{P}' \in \mathbb{P} : M_{\mathcal{P},\mathfrak{I}} > M_{\mathcal{P}',\mathfrak{I}} \Rightarrow \varrho(\mathcal{C}_{\Omega,\mathfrak{I}}, \mathcal{P}) > \varrho(\mathcal{C}_{\Omega,\mathfrak{I}}, \mathcal{P}')$

### 7.5.1 Bitcoin

In this section, we study the Bitcoin [Nak08a] blockchain protocol. Bitcoin is a prime example of a family of protocols that links the amount of valid blocks, that each party can produce per execution, with the party's hardware capabilities. This family includes: i) Proof-of-Work-based protocols like Ethereum [Woo14], Bitcoin NG [EGSVR16], or Zerocash [BCG+14]; ii) Proof-of-Space [DFKP15] and Proof-of-Space-Time [MO19] protocols like SpaceMint [PKF+18] and Chia [AAC+17, CP18].

**Execution Details.** Typically, protocols from the aforementioned family enforce that, when all parties follow the protocol honestly, the expected percentage of blocks

created by a party $\mathcal{P}$ is $\mu_{\mathcal{P}}$ of the total blocks produced by all parties during the execution. Along the lines of the formulation in [GKL15, GKL17], in the Bitcoin protocol, each party $\mathcal{P}$ can make at most $\mu_{\mathcal{P}} \cdot q$ queries to the hashing oracle $\mathcal{O}_{\Pi}$ per time slot, where $q$ is the total number of queries that all parties can make to $\mathcal{O}_{\Pi}$ during a time slot. We note that when $\mathcal{P}$ follows the Bitcoin protocol, they perform exactly $\mu_{\mathcal{P}} \cdot q$ queries to the hashing oracle. Each query can be seen as an independent block production trial and is successful with probability $\delta$, which is a protocol-specific "mining difficulty" parameter.

From the point of view of the observer $\Omega$, a party $\mathcal{P}$ is rewarded a fixed amount $R$ for each block they contribute to the chain output by $\Omega$. Then, Bitcoin implements a special case of block-proportional rewards (cf. Definition 36), such that:

- The total number of rewards for $\mathfrak{I}$ is

$$\mathcal{R}_{\Omega,\mathfrak{I}} = \left|\mathcal{C}_{\Omega,\mathfrak{I}}\right| \cdot R = \Big(\sum_{\hat{\mathcal{P}} \in \mathbb{P}} M_{\hat{\mathcal{P}},\mathfrak{I}}\Big) \cdot R \,,$$

  where $|\cdot|$ denotes the length of a chain in blocks.

- The proportional reward function $\varrho(\cdot,\cdot)$ is defined as

$$\varrho(\mathcal{C}_{\Omega,\mathfrak{I}}, \mathcal{P}) = \frac{M_{\mathcal{P},\mathfrak{I}}}{\left|\mathcal{C}_{\Omega,\mathfrak{I}}\right|} = \frac{M_{\mathcal{P},\mathfrak{I}}}{\sum_{\hat{\mathcal{P}} \in \mathbb{P}} M_{\hat{\mathcal{P}},\mathfrak{I}}}$$

Thus, by Definition 36, we have that

$$\forall \mathfrak{I}\ \forall \mathcal{P} \in \mathbb{P} : R_{\mathcal{P},\mathfrak{I}} = \varrho(\mathcal{C}_{\Omega,\mathfrak{I}}, \mathcal{P}) \cdot \mathcal{R}_{\Omega,\mathfrak{I}} = M_{\mathcal{P},\mathfrak{I}} \cdot R \,. \tag{7.1}$$

In Bitcoin, on each time slot a party keeps a local chain, which is the longest among all available chains. If multiple longest chains exist, the party follows the chronological ordering of messages.

Following, we assume that none of the participating parties has complete control over message delivery, i.e., apart from the preference index (cf. Section 7.1.1). Therefore, when two parties $\mathcal{P}, \mathcal{P}'$ produce blocks for the same index on the same time slot, it may be unclear which is adopted by third parties that follow the protocol.

Furthermore, the *index* of each block $m$ is an integer that identifies the distance of $m$ from $\mathcal{B}_G$, i.e., its height in the tree of blocks. Blocks on the same height, but different branches, have the same index but are non-equivalent (recall, that two messages are equivalent if their hash is equal).

**Bitcoin is a Compliant Protocol w.r.t. Reward.** We prove that Bitcoin is a $(\Theta(\delta^2), \mathcal{X})$-compliant protocol under the utility *Reward*, where $\mathcal{X}$ is any associated infraction predicate. By Definition 34 and Eq. (7.1), we have that when parties follow the strategy profile $\sigma$, the utility $U_{\mathcal{P}}$ of party $\mathcal{P}$ is

$$U_{\mathcal{P}}(\sigma) = E\big[M_{\mathcal{P},\mathcal{E}_\sigma}\big] \cdot R\,, \tag{7.2}$$

where $M_{\mathcal{P},\mathcal{E}_\sigma}$ is the (random variable) number of blocks produced by $\mathcal{P}$ in the chain output by $\Omega$ and $R$ is the fixed amount of rewards per block. Our analysis considers typical values of the success probability $\delta$, sufficiently small such that $\delta \cdot q < 1$ (recall that $q$ is the total number of oracle queries available to all parties per slot).

We say that party $\mathcal{P}$ is *successful* during time slot $r$, if $\mathcal{P}$ manages to produce at least one block, i.e., at least one oracle query submitted by $\mathcal{P}$ during $r$ was successful. The time slot $r$ is *uniquely successful* for $\mathcal{P}$, if no other party than $\mathcal{P}$ manages to produce a block in $r$. We prove the following lemma.

**Lemma 4.** *Assume an execution trace $\mathfrak{I}$ of the Bitcoin protocol where all parties follow the honest strategy. Let $\mathcal{B}_1, \ldots, \mathcal{B}_k$ be a sequence of blocks produced by party $\mathcal{P} \in \mathbb{P}$ during a time slot $r$ that was uniquely successful for $\mathcal{P}$ in $\mathfrak{I}$. Then, $\mathcal{B}_1, \ldots, \mathcal{B}_k$ will be included in the chain output by the observer $\Omega$.*

*Proof.* Let $h$ be the height of $\mathcal{B}_1$. Then, for every $j \in \{1, \ldots, k\}$, the height of the block $\mathcal{B}_j$ is $h + j - 1$. Assume for the sake of contradiction that there is a $j^* \in [k]$ such that $\mathcal{B}_{j^*}$ is not in the observer's chain. Since each block contains the hash of the previous block in the chain of $\Omega$, the latter implies that the subsequence $\mathcal{B}_{j^*}, \ldots, \mathcal{B}_k$ is not in the observer's chain. There are two reasons that $\mathcal{B}_{j^*}$ is missing from the observer's chain.

1. The observer $\Omega$ never received $\mathcal{B}_{j^*}$. However, after the end of time slot $r$, $\Omega$ will be activated and fetch the messages included in its $\mathsf{Receive}_\Omega()$ string. Therefore, the case that $\Omega$ never received $\mathcal{B}_{j^*}$ cannot happen.

2. The observer has another block $\mathcal{B}'_{j^*}$ included in its chain that has the same height, $h + j^* - 1$, as $\mathcal{B}_{j^*}$. Since $r$ was uniquely successful for $\mathcal{P}$ in $\mathfrak{I}$, the block $\mathcal{B}'_{j^*}$ must have been produced in a time slot $r'$ that is different than $r$. Assume that $\mathcal{P}$ produced the block sequence $\mathcal{B}'_{j^*}, \ldots, \mathcal{B}'_{k'}$ during $r'$. We examine the following two cases:

   (a) $r > r'$: then $\mathcal{B}'_{j^*}$ was produced before $\mathcal{B}_{j^*}$, so in time slot $r' + 1$ all parties received (at least) the sequence $\mathcal{B}'_{j^*}, \ldots, \mathcal{B}'_{k'}$. All parties select the longest

chain, so the chain that they will select will have at least $h+k'-1 \geq h+j^*-1$ number of blocks in $r'+1$. Thus, for time slot $r \geq r'+1$ the parties submit queries for producing blocks which height is at least $h+k' > h+j^*-1$. So at time slot $r$, the party $\mathcal{P}$ cannot have produced a block which height is $\leq h+j^*-1$.

(b) $r < r'$: then $\mathcal{B}_{j^*}$ was produced before $\mathcal{B}'_{j^*}$. So in time slot $r+1$ all parties receive the sequence $\mathcal{B}_1, \ldots, \mathcal{B}_k$. Thus, they will adopt a chain with at least $h+k-1 \geq h+j^*-1$ number of blocks. Thus, for time slot $r' \geq r+1$ the parties submit queries for producing blocks with height at least $h+k > h+j^*-1$. Therefore, $\mathcal{P}$ cannot have produced a block of height $\leq h+j^*-1$ during time slot $r'$.

By the above, $\mathcal{B}_{j^*}$ is a block with height $h+j^*-1$ received by $\Omega$ and no other block with height $h+j^*-1$ is included in $\Omega$'s chain. Since $\Omega$ adopts the longest chain, there must be a block with height $h+j^*-1$ that is included in its chain. It is straightforward that this block will be $\mathcal{B}_{j^*}$, which leads to contradiction. □

Subsequently, we apply Lemma 4 to the proof of the main theorem of this subsection, stated below.

**Theorem 8.** *Let $N$ be the number of time slots of the execution and $\mathcal{X}$ be any associated infraction predicate. Let $\bar{U}$ be the utility vector where each party employs the utility Reward. Then, the Bitcoin protocol is $(\epsilon, \mathcal{X})$-compliant w.r.t. $\bar{U}$, for $\epsilon := \frac{NRq^2}{2}\delta^2$ .*

*Proof.* We will show that the Bitcoin protocol is an $\epsilon$-Nash equilibrium w.r.t. $\bar{U}$, for $\epsilon := \frac{NRq^2}{2}\delta^2$. By Proposition 1, the latter implies that the Bitcoin protocol is $(\epsilon, \mathcal{X})$-compliant w.r.t. $\bar{U}$.

Consider a protocol execution where all parties follow the honest strategy $\Pi$, with $\sigma_\Pi$ denoting the profile $\langle \Pi, \ldots, \Pi \rangle$. Let $\mathcal{P}$ be a party and $\mu_\mathcal{P}$ be its mining power. For $r \in [N]$, let $X^{\sigma_\Pi}_{\mathcal{P},r}$ be the random variable that is $1$ if the time slot $r$ is uniquely successful for $\mathcal{P}$ and $0$ otherwise. By protocol description, a party $\mathcal{P}'$ makes $\mu_{\mathcal{P}'} \cdot q$ oracle queries

during $r$, each with success probability $\delta$. Thus:

$$\begin{aligned}
&\mathsf{Pr}[X^{\sigma_\Pi}_{\mathcal{P},r} = 1] = \\
=&\,\mathsf{Pr}[\mathcal{P} \text{ is successful during } r]\cdot \\
&\qquad \cdot \mathsf{Pr}[\text{all the other parties produce no blocks in } r] = \\
=&\Big(1-(1-\delta)^{\mu_\mathcal{P} q}\Big)\cdot \prod_{\mathcal{P}'\neq\mathcal{P}} (1-\delta)^{\mu_{\mathcal{P}'} q} = \\
=&\Big(1-(1-\delta)^{\mu_\mathcal{P} q}\Big)\cdot (1-\delta)^{(1-\mu_\mathcal{P})q} = (1-\delta)^{(1-\mu_\mathcal{P})q} - (1-\delta)^q\,.
\end{aligned} \tag{7.3}$$

The random variable $X_{\mathcal{P},\mathcal{E}_{\sigma_\Pi}} := \sum_{r\in[N]} X^{\sigma_\Pi}_{\mathcal{P},r}$ expresses the number of uniquely successful time slots for $\mathcal{P}$. By Eq. (7.3), $X^{\Pi}_{\mathcal{P}}$ follows the binomial distribution with $N$ trials and probability of success $(1-\delta)^{(1-\mu_\mathcal{P})q} - (1-\delta)^q$. Therefore: $E\big[X_{\mathcal{P},\mathcal{E}_{\sigma_\Pi}}\big] = N\Big((1-\delta)^{(1-\mu_\mathcal{P})q} - (1-\delta)^q\Big)$.

Let $M_{\mathcal{P},\mathcal{E}_{\sigma_\Pi}}$ be the number of blocks produced by $\mathcal{P}$ included in the chain output by the observer $\Omega$. In a uniquely successful time slot $r$, $\mathcal{P}$ produces at least one block, and by Lemma 4, all the blocks that $\mathcal{P}$ produces during $r$ will be included in the chain output by the observer. Therefore, for all random coins $M_{\mathcal{P},\mathcal{E}_{\sigma_\Pi}} \geq X_{\mathcal{P},\mathcal{E}_{\sigma_\Pi}}$ and so it holds that:

$$E\big[M_{\mathcal{P},\mathcal{E}_{\sigma_\Pi}}\big] \geq E\big[X_{\mathcal{P},\mathcal{E}_{\sigma_\Pi}}\big] = N\Big((1-\delta)^{(1-\mu_\mathcal{P})q} - (1-\delta)^q\Big)\,. \tag{7.4}$$

Now assume that $\mathcal{P}$ decides to unilaterally deviate from the protocol, following a strategy $S$. Let $\sigma$ denote the respective strategy profile. Let $Z_{\mathcal{P},\mathcal{E}_\sigma}$ be the number of blocks that $\mathcal{P}$ produces by following $S$ and $M_{\mathcal{P},\mathcal{E}_\sigma}$ be the number of blocks produced by $\mathcal{P}$ that will be included in the chain output by $\Omega$. Clearly, for all random coins $M_{\mathcal{P},\mathcal{E}_\sigma} \leq Z_{\mathcal{P},\mathcal{E}_\sigma}$. Without loss of generality, we may assume that $\mathcal{P}$ makes all of their $N\mu_\mathcal{P} q$ available oracle queries (indeed, if $\mathcal{P}$ made less than $N\mu_\mathcal{P} q$ queries then on average it would produce less blocks). Thus, we observe that $Z_{\mathcal{P},\mathcal{E}_\sigma}$ follows the binomial distribution with $N\mu_\mathcal{P} q$ trials and probability of success $\delta$. Thus:

$$E\big[M_{\mathcal{P},\mathcal{E}_\sigma}\big] \leq \big[Z_{\mathcal{P},\mathcal{E}_\sigma}\big] = N\mu_\mathcal{P} q\delta\,. \tag{7.5}$$

By definition of $U_\mathcal{P}$ in Eq. (7.2) and Eq. (7.4) and (7.5), for fixed block rewards $R$ and for every strategy $S$ that $\mathcal{P}$ may follow, we have:

$$\begin{aligned}
U_\mathcal{P}(\sigma) - U_\mathcal{P}(\sigma_\Pi) &= E\big[M_{\mathcal{P},\mathcal{E}_\sigma}\big]\cdot R - E\big[M_{\mathcal{P},\mathcal{E}_{\sigma_\Pi}}\big]\cdot R \leq \\
&\leq N\mu_\mathcal{P} q\delta R - N\Big((1-\delta)^{(1-\mu_\mathcal{P})q} - (1-\delta)^q\Big)R\,.
\end{aligned} \tag{7.6}$$

By Bernoulli's inequality, we have that: $(1-\delta)^{(1-\mu_{\mathcal{P}})q} \geq 1-(1-\mu_{\mathcal{P}})q\delta$. Besides, by binomial expansion, and the assumption that $\delta \cdot q < 1$ we have that: $(1-\delta)^q \leq 1-q\delta+\frac{q^2}{2}\delta^2$. Thus, by applying the two above inequalities in Eq. (7.6), we get that:

$$\begin{aligned} &U_{\mathcal{P}}(\sigma)-U_{\mathcal{P}}(\sigma_{\Pi}) \leq N\mu_{\mathcal{P}}q\delta R - N\Big((1-\delta)^{(1-\mu_{\mathcal{P}})q}-(1-\delta)^q\Big)R \leq \\ \leq &N\mu_{\mathcal{P}}q\delta R - N\Big(1-(1-\mu_{\mathcal{P}})q\delta-\Big(1-q\delta+\frac{q^2}{2}\delta^2\Big)\Big)R = \\ = &N\mu_{\mathcal{P}}q\delta R - N\Big(\mu_{\mathcal{P}}q\delta-\frac{q^2}{2}\delta^2\Big)R = \frac{NRq^2}{2}\delta^2 \end{aligned} \tag{7.7}$$

By Eq. (7.7), Bitcoin is an $\epsilon$-Nash equilibrium for $\epsilon := \frac{NRq^2}{2}\delta^2$. □

The result of Theorem 8 shows that Bitcoin w.r.t. rewards is compliant, by proving that it is an approximate Nash equilibrium. This is in agreement with previous results [KDF13, KS19]. Similar results exist for Bitcoin w.r.t. profit [BGM+18, KS19], therefore compliance results for this utility are expected to also be achievable.

**Remark.** *A well-known result from the Selfish Mining attack [ES14, SSZ16] is that Bitcoin is not an equilibrium with respect to relative rewards. Therefore, we stress that the above analysis concerns the utility of absolute rewards, although evaluating compliance under relative rewards is an interesting open question.*

### 7.5.2 Proof-of-Stake

Proof-of-Stake (PoS) systems differ from Bitcoin in a few points. Typically, the execution of such systems is organized in *epochs*, each consisting of a fixed number $l_e$ of time slots. On each slot, a specified set of parties is eligible to participate in the protocol. Depending on the protocol, the leader schedule of each epoch may or may not be a priori public.

The core difference from PoW concerns the power $\mu_{\mathcal{P}}$ of each party. In PoS, $\mu_{\mathcal{P}}$ represents their *stake* in the system, i.e., the number of coins, that $\mathcal{P}$ owns. Stake is dynamic, therefore the system's coins may change hands and the leader schedule of each epoch depends on the stake distribution at the beginning of the epoch.[8] As with Bitcoin, each party participates proportionally to their power, so, on expectation, the ratio of slots for which $\mathcal{P}$ is leader, over the total number of the epoch's slots, is $\mu_{\mathcal{P}}$.

Moreover, in PoS protocols, the oracle $\mathcal{O}_{\Pi}$ does not perform hashing as in Bitcoin. Instead, it is parameterized by the leader schedule and typically performs signing.

[8] In reality, the snapshot of the stake distribution is retrieved at an earlier point of the previous epoch, but we can employ this simplified version without loss of generality

The signature output by $\mathcal{O}_{\Pi}$ is valid if and only if the input message is submitted by the leader of the slot, during which $\mathcal{O}_{\Pi}$ is responding. This introduces two important consequences: i) only the slot leader can produce valid messages during a given slot; ii) the leader can produce as many valid messages as the number of possible queries to $\mathcal{O}_{\Pi}$.

In the upcoming paragraphs we use the following notation:

- $C$: the cost of a single query to $\mathcal{O}_{\Pi}$;
- $R$: the (fixed) reward per block;
- $e$: the number of epochs in an execution;
- $l_e$: the number of slots per epoch;
- $\mu_{\mathcal{P},j}$: the power of party $\mathcal{P}$ on epoch $j$.

#### 7.5.2.1 Single-Leader Proof-of-Stake

As before, we analyze a representative of a family of protocols, in this case the PoS protocol Ouroboros [KRDO17]. This family includes systems like EOS[9], Steem[10], and Ouroboros BFT [KR18]. We again utilize the blockchain infraction predicates (cf. Definition 33). As shown in Section 7.4, fair rewards do not necessarily guarantee compliance; as Ouroboros itself does not define rewards, we now consider the same block-proportional rewards (cf. Definition 36) as Bitcoin (i.e., fixed rewards per block).

On each slot, Ouroboros defines a single party, the "slot leader", as eligible to create a valid message. Specifically, the protocol restricts that a leader cannot extend the chain with multiple blocks for the same slot, therefore all honest parties extend their chain by at most $1$ block per slot. The leader schedule is public and is computed at the beginning of each epoch via a secure, publicly verifiable Multi-Party Computation (MPC) sub-protocol, which cannot be biased by any single party. To prevent long-range attacks [But14], Ouroboros employs a form of rolling checkpoints ("a bounded-depth longest-chain rule" [KRDO17]), i.e., a party ignores forks that stem from a block older than a (protocol-specific) limit from the adopted chain's head.

Ouroboros is a representative of a family of protocols that demonstrates the following properties:

- the execution is organized in epochs;

[9] `https://developers.eos.io/welcome/latest/protocol/consensus_protocol`
[10] `https://steem.com/`

- within each epoch, a single party (the *leader*) is eligible to produce a message per index;
- a party which is online considers the blocks of each past epoch finalized (i.e., does not remove them in favor of a competing, albeit possibly longer, chain);
- no single party with power less than $\frac{1}{2}$ can bias the epoch's leader schedule.

**Synchronous network.** First, we assume a synchronous network (cf. Section 7.1.1). Theorem 9 shows that Ouroboros with block-proportional rewards under synchronous networks is $(\epsilon, \mathcal{X})$-compliant, $\mathcal{X}$ being any associated infraction predicate, for negligible $\epsilon$; this result is in line with the informal incentives' analysis of Ouroboros [KRDO17].

**Theorem 9.** *Assume i) a synchronous network (cf. Section 7.1.1), ii) any associated infraction predicate $\mathcal{X}$, and iii) that $\forall \mathcal{P} \in \mathbb{P} : \mu_{\mathcal{P}} < \frac{1}{2}$. Ouroboros with block-proportional rewards (cf. Definition 36, for fixed block reward $R$) is $(\epsilon, \mathcal{X})$-compliant (cf. Definition 32) w.r.t. utility Reward (cf. Definition 34) and, if $R > C$, it is also $(\epsilon, \mathcal{X})$-compliant w.r.t. utility Profit, in both cases for negligible $\epsilon$.*

*Proof.* To prove the theorem, it suffices to show that, if the assumptions hold, Ouroboros is an $\epsilon$-Nash equilibrium (cf. Proposition 1), i.e., no party can increase its reward more than $\epsilon$ by unilaterally deviating from the protocol, where $\epsilon = \mathsf{negl}(\kappa)$.

First, if all parties control a minority of staking power, no single party can bias the slot leader schedule for any epoch (unless with $\mathsf{negl}(\kappa)$ probability). Therefore, the (maximum) expected number of slots for which each party $\mathcal{P}$ is leader is $\sum_{j \in [1,e]} l_e \cdot \mu_{\mathcal{P},j}$, where $\mu_{\mathcal{P},j}$ is the percentage of staking power of $\mathcal{P}$ during the $j$-th epoch.

Second, if all parties follow $\Pi$, then the total expected rewards for each party $\mathcal{P}$ are $R \cdot \sum_{j \in [1,e]} l_e \cdot \mu_{\mathcal{P},j}$. This is a direct consequence of the network synchronicity assumption. Specifically, on slot $r$ the (single) leader $\mathcal{P}$ creates exactly one block $\mathcal{B}$, which extends the longest chain (adopted by $\mathcal{P}$). At the beginning of slot $r+1$, all other parties receive $\mathcal{B}$ and, since $\mathcal{B}$ is now part of the (unique) longest chain, all parties adopt it. Consequently, all following leaders will extend the chain that contains $\mathcal{B}$, so eventually $\mathcal{B}$ will be in the chain output by $\Omega$. Therefore, if all parties follow the protocol and no party can bias the leader schedule, then no party can increase its expected rewards by deviating from the protocol.

Regarding profit, a leader creates a block by performing a single query to $\mathcal{O}_{\Pi}$. Additionally, cost depends only on the number of such queries. Therefore, if the cost of

performing a single query is less than $R$, then the profit per slot is larger than $0$, so abstaining from even a single slot reduces the expected aggregate profit; therefore, all parties are incentivized to participate in all slots. □

**Lossy network.** Second, we assume a lossy network (cf. Section 7.1.1). Theorem 10 shows that Ouroboros with block proportional rewards is *not* compliant w.r.t. the conflicting infraction predicate $\mathcal{X}_{conf}$; specifically, it shows that $\epsilon$ is upper-bounded by a large value, which is typically non-negligible.

**Theorem 10.** *Assume i) a lossy network with (non-negligible) parameter $d$ (cf. Section 7.1.1), ii) the conflicting infraction predicate $\mathcal{X}_{conf}$ (cf. Definition 33), iii) that $\forall \mathcal{P}' \in \mathbb{P} : \mu_{\mathcal{P}'} < \frac{1}{2}$, and iv) that $\mathcal{P}$ is the party with maximum $\sum_{j\in[1,e]} \mu_{\mathcal{P},j}$ over all parties.*

*Ouroboros with block-proportional rewards (cf. Definition 36, for fixed block reward $R$) is not $(\epsilon, \mathcal{X}_{conf})$-compliant (cf. Definition 32), w.r.t. utility Reward for any $\epsilon < d \cdot (1-d)^2 \cdot R \cdot \sum_{j\in[1,e]} l_e \cdot \mu_{\mathcal{P},j}$, and is also not $(\epsilon, \mathcal{X}_{conf})$-compliant, w.r.t. utility Profit for any $\epsilon < (d \cdot (1-d)^3 \cdot R - C) \cdot \sum_{j\in[1,e]} l_e \cdot \mu_{\mathcal{P},j}$.*

*Proof.* To prove the statement we define the following event $E$ for party $\mathcal{P}$, when all parties follow $\Pi$ and $r$ is a slot where $\mathcal{P}$ is the leader. $\mathcal{P}$ extends its adopted longest chain $\mathcal{C}$ with a new block $\mathcal{B}$. However, $\mathcal{B}$ is dropped, i.e., it is not delivered to any party $\mathcal{P}' \neq \mathcal{P}$; this occurs with probability $d$, a parameter of the lossy network. Therefore, the leader of slot $r+1$, who does not adopt $\mathcal{B}$, creates a different, "competing" block $\mathcal{B}'$, which is not dropped (with probability $1-d$), thus all parties, except $\mathcal{P}$, adopt $\mathcal{B}'$. Finally, on round $r+2$, the slot leader $\mathcal{P}'' \neq \mathcal{P}$ produces a block $\mathcal{B}''$ which extends $\mathcal{B}'$ and $\mathcal{B}''$ is not dropped, therefore all parties – including $\mathcal{P}$ adopt, on round $r+3$, the – now longest – chain $\mathcal{C}||\mathcal{B}'||\mathcal{B}''$. The probability that $E$ occurs is $d \cdot (1-d)^2$, which is non-negligible. This follows from the fact that the probability that each block is dropped is independent. So, the expected rewards of $\mathcal{P}$ when everyone follows $\Pi$ are at most $(1 - d \cdot (1-d)^2) \cdot R \cdot \sum_{j\in[1,e]} l_e \cdot \mu_{\mathcal{P},j}$. However, as described above, the maximum rewards are $R \cdot \sum_{j\in[1,e]} l_e \cdot \mu_{\mathcal{P},j}$, i.e., they are strictly larger.

Now, consider the following strategy $S$. When party $\mathcal{P}$ is the slot leader, it creates $t$ messages, all extending its adopted longest chain; in other words, $\mathcal{P}$ forks the chain by producing $t$ competing blocks. Clearly, as long as at least one of the $t$ blocks is delivered, event $E$ will not occur, the leader of slot $r+1$ will adopt one of the blocks created by $\mathcal{P}$ and, eventually, $\mathcal{P}$ will receive the rewards that correspond to slot $r$. The probability that all $t$ blocks are dropped is $d^t$, while the expected rewards are

$(1-d^t\cdot(1-d)^2)\cdot R\cdot\sum_{j\in[1,e]}l_e\cdot\mu_{\mathcal{P},j}$. As $t$ increases, this probability tends to $0$ and the party's expected rewards tend to the maximum value $R\cdot\sum_{j\in[1,e]}l_e\cdot\mu_{\mathcal{P},j}$.

Therefore, for utility *Reward*, the profile $\sigma_{\mathcal{P},S}$, under which all parties except $\mathcal{P}$ employ $\Pi$ and $\mathcal{P}$ employs $S$, is directly reachable (cf. Definition 31) from the "all honest" profile $\sigma_{\Pi}$. Hence, since $S$ is non-compliant, so is $\Pi$, for values of $\epsilon <$. $R\cdot\sum_{j\in[1,e]}l_e\cdot\mu_{\mathcal{P},j}-(1-d\cdot(1-d)^2)\cdot R\cdot\sum_{j\in[1,e]}l_e\cdot\mu_{\mathcal{P},j}=d\cdot(1-d)^2\cdot R\cdot\sum_{j\in[1,e]}l_e\cdot\mu_{\mathcal{P},j}$.

Regarding cost, we consider the party $\mathcal{P}$ with the maximum power across the execution and the case when $t=2$, i.e., the simplest case of non-compliant deviation. The expected rewards of $\mathcal{P}$ are $(1-d^2\cdot(1-d)^2)\cdot R\cdot\sum_{j\in[1,e]}l_e\cdot\mu_{\mathcal{P},j}$, therefore its reward increase (instead of employing $\Pi$) is $((1-d^2\cdot(1-d)^2)-(1-d\cdot(1-d)^2))\cdot R\cdot\sum_{j\in[1,e]}l_e\cdot\mu_{\mathcal{P},j}=d\cdot(1-d)^3\cdot R\cdot\sum_{j\in[1,e]}l_e\cdot\mu_{\mathcal{P},j}$. Hence, given a cost $C$ per query, if the aggregate cost for performing these extra queries to $\mathcal{O}_{\Pi}$ is less than this gain, $\mathcal{P}$ is incentivized to deviate so. □

The lossy network analysis is particularly of interest, because it manifested in practice. On December 2019, Cardano released its Incentivized Testnet (ITN)[11]. On the ITN, Cardano stakeholders, i.e., users who owned Cardano's currency, participated in PoS by forming stake pools which produced blocks. The ITN used proportional rewards and the execution followed the Ouroboros model of epochs and slots. Specifically, each pool was elected as a slot leader proportionally to its stake and received its proportional share of an epoch's rewards based on its *performance*, i.e., the number of produced blocks compared to the number of *expected* blocks it should have produced. This incentivized pool operators to actively avoid abstaining, i.e., failing to produce a block when elected as slot leader. However, forks started to form while the network was unstable and lossy. This incentivized pools[12] to "clone" their nodes, i.e., run multiple node instances in parallel, thus increasing network connectivity, reducing packet loss, but also extending all possible forks. To make matters worse, this solution not only perpetuated forks but also created new ones, as clones would not coordinate and produced different blocks, even when extending the same chain.

**Remark.** *Although a lossy network may render a PoS protocol non-compliant, the same does not hold for PoW ledgers. As described in the proof of Theorem 10, a party produces multiple blocks per slot to maximize the probability that one of them is eventually output by $\Omega$. Notably, since the PoS protocol restricts that at most one block extends the longest chain*

[11] `https://staking.cardano.org/`

[12] For a (heated) discussion on this issue see Reddit: `https://www.reddit.com/r/cardano/comments/ekncza`

*per slot, these blocks are necessarily conflicting. However, PoW ledgers do not enforce such restriction; therefore, a party would instead create multiple consecutive (instead of parallel, conflicting) blocks, as covered in the proof of Theorem 8, which yields maximal expected rewards even under a lossy network.*

#### 7.5.2.2 Multi-Leader Proof-of-Stake

We now turn to Ouroboros Praos [DGKR18], the representative protocol of a family that includes Ouroboros Genesis [BGK+18], Peercoin [KN12], and Tezos' baking system [Tez20]. These are similar to the previous PoS family (cf. Section 7.5.2.1), but with one difference: it is possible that multiple parties are chosen as leaders for the same time slot. As Theorem 11 shows, Ouroboros Praos and the other members of this family are not compliant. The core argument of the proof is the same as with Ouroboros under a lossy network (Theorem 10). Specifically, assuming a randomized message delivery, a party is incentivized to produce multiple blocks to decrease the probability that a conflicting block, produced by a fellow slot leader, is adopted over their own. We note that, neither Ouroboros nor Ouroboros Praos enforce a choice policy in case of conflicting messages. However, parties typically utilize network delivery, opting for the message that arrives first.

**Theorem 11.** *Assume i) a synchronous network (cf. Section 7.1.1), ii) the conflicting infraction predicate $\mathcal{X}_{conf}$ (Definition 33), iii) that $\forall \mathcal{P}' \in \mathbb{P} : \mu_{\mathcal{P}'} < \frac{1}{2}$, and iv) that $\mathcal{P}$ is the party with maximum $\sum_{j\in[1,e]} \mu_{\mathcal{P},j}$ over all parties.*

*Ouroboros Praos with block-proportional rewards (Definition 36, for fixed block reward $R$) is not $(\epsilon, \mathcal{X}_{conf})$-compliant (Definition 32) w.r.t. utility Reward for (non-negligible) $\epsilon < \frac{1}{2} p_l \cdot R \cdot \sum_{j\in[1,e]} l_e \cdot \mu_{\mathcal{P},j}$, where $p_l$ is the (protocol-dependent) probability that 2 leaders are elected for the same slot, and is not $(\epsilon, \mathcal{X}_{conf})$-compliant w.r.t. utility Profit for any $\epsilon < (\frac{1}{6} p_l \cdot R - C) \cdot \sum_{j\in[1,e]} l_e \cdot \mu_{\mathcal{P},j}$.*

*Proof.* As with Theorem 10, we will define a bad event $E$, during which the expected rewards of party $\mathcal{P}$ are less if following $\Pi$, compared to a non-compliant strategy.

Let $r$ be a slot during which $\mathcal{P}$ is leader. Additionally, a different party $\mathcal{P}'$ is also a leader of $r$. Both parties create two different messages and both set the network priority to the maximum (cf. Section 7.1.1); we note that typically the protocol instructs the parties to diffuse the messages as soon as possible, which is equivalent to setting maximum priority. Following, the diffuse functionality delivers $\mathcal{B}'$ before $\mathcal{B}$, therefore all parties (possibly except $\mathcal{P}$) adopt $\mathcal{B}'$ and ignore $\mathcal{B}$.

Let $p_E$ be the probability that $E$ occurs. First, $p_E$ depends on the probability $p_l$ that multiple leaders exist alongside $\mathcal{P}$; $p_l$ depends on the protocol's leader schedule functionality, but should typically be non-negligible. Second, it depends on the order delivery of $\mathcal{B}, \mathcal{B}'$; since both have the same priority and the delivery is randomized, (cf. Section 7.1.1), the probability $p_n$ that $\mathcal{B}'$ is delivered before $\mathcal{B}$ is $p_n = \frac{1}{2}$. Therefore, it holds $p_E = p_l \cdot p_n = \frac{1}{2} p_l$, which is non-negligible.

Additionally, if $\sum_{j \in [1,e]} l_e \cdot \mu_{\mathcal{P},j}$ is the number of slots during which $\mathcal{P}$ is leader, the total expected rewards of $\mathcal{P}$, when everybody (including $\mathcal{P}$) follows $\Pi$, are at most $(1 - \frac{1}{2} p_l) \cdot R \cdot \sum_{j \in [1,e]} l_e \cdot \mu_{\mathcal{P},j}$.

Now, consider the following (non-compliant) strategy, which is employed only by $\mathcal{P}$. When $\mathcal{P}$ is slot leader, it produces $t > 1$ blocks, which it diffuses with maximum priority. If every party $\mathcal{P}' \neq \mathcal{P}$ follows $\Pi$, the probability $p_l$ remains the same as before. Therefore, if $\mathcal{P}, \mathcal{P}'$ are both leaders, the probability that the following slot's leader adopts the block of $\mathcal{P}'$ is $\frac{1}{t+1}$. So, the total expected rewards of $\mathcal{P}$ under the new strategy are $(1 - \frac{1}{t+1} p_l) \cdot R \cdot \sum_{j \in [1,e]} l_e \cdot \mu_{\mathcal{P},j}$. Since $\frac{1}{t+1} < \frac{1}{2}$, the new strategy profile is directly reachable from the $\sigma_{\Pi}$, when every party follows $\Pi$. Additionally, as $t \to \infty$, the expected rewards tend to the (maximum) value $R \cdot \sum_{j \in [1,e]} l_e \cdot \mu_{\mathcal{P},j}$.

Regarding *Profit*, for $t = 2$ the increase in expected rewards is: $((1 - \frac{1}{3} p_l) - (1 - \frac{1}{2} p_l)) \cdot R \cdot \sum_{j \in [1,e]} l_e \cdot \mu_{\mathcal{P},j} \Rightarrow \frac{1}{6} p_l \cdot R \cdot \sum_{j \in [1,e]} l_e \cdot \mu_{\mathcal{P},j}$, thus the bound for $\epsilon$ follows with the same reasoning as Theorem 10. □

## 7.6 Externalities

In this section, we enhance our analysis with parameters external to the distributed system. First, we introduce an exchange rate, to model the rewards' price in the same unit of account as the cost. We analyze how the exchange rate should behave to ensure compliance, assuming an external utility which is awarded for an infraction. Following, we identify historical patterns to approximate this behavior and, finally, introduce penalties, which disincentivize infraction when the exchange rate behavior does not suffice.

### 7.6.1 Utility under Externalities

In distributed ledger systems, rewards are denominated in the ledger's native currency (i.e., satoshi in Bitcoin, wei in Ethereum, etc). However, cost is typically denominated in fiat (USD, GBP, etc). Therefore, we introduce an *exchange rate*, between the ledger's

native currency and USD, to denominate the rewards and cost in the same unit of account and precisely estimate a party's utility.

The exchange rate $X_{\mathcal{E}}$ is a random variable, parameterized by a strategy profile $\sigma$. For a trace $\mathfrak{I}_\sigma$ under $\sigma$, the exchange rate takes a non-negative real value. The exchange rate is applied once, at the end of the execution. Intuitively, this implies that a party eventually sells their rewards at the end of the execution. Therefore, its utility depends on the accumulated rewards, during the execution, and the exchange rate at the end.

The infraction predicate expresses a deviant behavior that parties may exhibit. So far, we considered distributed protocols in a standalone fashion, analyzing whether they incentivize parties to avoid infractions. In reality, a ledger exists alongside other systems, and a party's utility may depend on parameters external to the distributed ledger. For instance, double spending against Bitcoin is a common hazard, which does not increase an attacker's *Bitcoin rewards*, but awards them external rewards, e.g., goods that are purchased with the double-spent coins.

The external – to the ledger – reward is modeled as a random variable $B_{\mathcal{P},\mathcal{E}_\sigma}$, which takes non-negative integer values. Similarly to the rewards' random variable, it is parameterized by a party $\mathcal{P}$ and a strategy profile $\sigma$. The infraction utility is applied once when computing a party's utility and has the property that, for every trace $\mathfrak{I}$ during which a party $\mathcal{P}$ performs no infraction, it holds that $B_{\mathcal{P},\mathfrak{I}} = 0$. Therefore, a party receives these external rewards only by performing an infraction.

Taking into account the exchange rate and the external infraction-based utility, we now define a new utility. As with Definition 34, the new utility $U$ takes two forms, *Reward* and *Profit*. For the former, $U$ normalizes the protocol rewards by applying the exchange rate and adds the external, infraction rewards. For *Profit*, it also subtracts the cost (which is already denominated in the base currency). Definition 37 defines the utility under externalities. For ease of reading, we use the following notation:

- $\rho_{\mathcal{P},\sigma} = E[R_{\mathcal{P},\mathcal{E}_\sigma}]$;
- $x_\sigma = E[X_{\mathcal{E}_\sigma}]$;
- $b_{\mathcal{P},\sigma} = E[B_{\mathcal{P},\mathcal{E}_\sigma}]$;
- $c_{\mathcal{P},\sigma} = E[C_{\mathcal{P},\mathcal{E}_\sigma}]$.

**Definition 37.** *Let: i) $\sigma$ be a strategy profile; ii) $\mathcal{E}_\sigma$ be an execution under $\sigma$; iii) $x_\sigma$ be the (expected) exchange rate of $\mathcal{E}_\sigma$; iv) $b_{\mathcal{P},\sigma}$ be the (expected) external rewards of $\mathcal{P}$ under $\sigma$. We define two types of utility $U_\mathcal{P}$ of a party $\mathcal{P}$ for $\sigma$ under externalities:*

1. *Reward:* $U_\mathcal{P}(\sigma) = \rho_{\mathcal{P},\sigma} \cdot x_\sigma + b_{\mathcal{P},\sigma}$
2. *Profit:* $U_\mathcal{P}(\sigma) = \rho_{\mathcal{P},\sigma} \cdot x_\sigma + b_{\mathcal{P},\sigma} - c_{\mathcal{P},\sigma}$

### 7.6.2 Compliance under Externalities

We now revisit our results, taking externalities into account. The core idea is to find the relation between the assets' price and external, infraction-based rewards, such that the former counters the latter, hence parties are incentivized to remain compliant.

In systems that are compliant in the standalone setting, i.e., without considering externalities, it suffices to show that the exchange rate is reduced to an extent which counterbalances the external rewards. We consider two strategy profiles: i) $\sigma_\Pi$ is the profile under which all parties follow the protocol $\Pi$; ii) $\sigma_{S_\mathcal{P}}$ is the profile under which all parties except $\mathcal{P}$ follow $\Pi$, whereas $\mathcal{P}$ deviates by following a non-compliant strategy $S_\mathcal{P}$; for this deviation, $\mathcal{P}$ receives external rewards $b_{\mathcal{P},\sigma_{S_\mathcal{P}}}$. Assume a system that is $(\epsilon, \mathcal{X})$-compliant; under externalities, the system remains compliant, for the same $\epsilon$, as long as it holds: $\forall \mathcal{P}\ \forall S_\mathcal{P} : (\rho_{\mathcal{P},\sigma_{S_\mathcal{P}}} \cdot x_{\sigma_{S_\mathcal{P}}} + b_{\mathcal{P},\sigma_{S_\mathcal{P}}}) - (\rho_{\mathcal{P},\sigma_\Pi} \cdot x_{\sigma_\Pi}) < \epsilon$. In some cases, e.g., under fair rewards or in the synchronous setting of single-leader PoS (cf. Subsection 7.5.2.1), the expected rewards are equal under both profiles. In those cases, it holds: $b_{\mathcal{P},\sigma_{S_\mathcal{P}}} \leq \rho_{\mathcal{P},\sigma_\Pi} \cdot (x_{\sigma_\Pi} - x_{\sigma_{S_\mathcal{P}}}) + \epsilon$. Therefore, if the exchange rate is sufficiently reduced, when $\mathcal{P}$ performs an infraction, $\mathcal{P}$ is incentivized to remain compliant. More formally, Theorem 12 analyzes Ouroboros under a synchronous network and externalities; similar statements can be made for the positive results of Sections 7.4 and 7.5.1.

**Theorem 12.** *Assume i) a synchronous network (cf. Section 7.1.1), ii) the conflicting predicate $\mathcal{X}_{conf}$, and iii) that $\forall \mathcal{P} \in \mathbb{P} : \mu_\mathcal{P} < \frac{1}{2}$. Also let: i) $\mathbb{S}_{-\mathcal{X}_{conf}}$: the set of all non $\mathcal{X}_{conf}$-compliant strategies; ii) $x_{\sigma_\Pi}$: the (expected) exchange rate under $\mathcal{E}_{\sigma_\Pi}$; iii) $x_{\sigma_{S_\mathcal{P}}}$: the (expected) exchange rate when only $\mathcal{P}$ employs some non $\mathcal{X}_{conf}$-compliant strategy $S_\mathcal{P}$; iv) $b_{\mathcal{P},\sigma_{S_\mathcal{P}}}$: the external utility that $S_\mathcal{P}$ yields for $\mathcal{P}$. Ouroboros with block-proportional rewards (cf. Definition 36, for fixed block reward $R$) under the aforementioned externalities is not $(\epsilon, \mathcal{X}_{conf})$-compliant (cf. Definition 32) w.r.t. utility Reward (cf. Definition 34) and, if $R > C$, it is also not $(\epsilon, \mathcal{X}_{conf})$-compliant w.r.t. utility Profit, in both cases for some $\epsilon < \max\{\max_{\mathcal{P}\in\mathbb{P}}\{\max_{S_\mathcal{P}\in\mathbb{S}_{-\mathcal{X}_{conf}}} \{\rho_{\mathcal{P},\sigma_\Pi} \cdot (x_{\sigma_{S_\mathcal{P}}} - x_{\sigma_\Pi}) + b_{\mathcal{P},\sigma_{S_\mathcal{P}}}\}\}, 0\}$.*

*Proof.* Following the same reasoning as Theorem 9, if a party $\mathcal{P}$ deviates by only producing conflicting messages, but does not abstain, its expected rewards are the same as following the protocol; specifically, due to network synchronicity, after every round when $\mathcal{P}$ is leader, every other party adopts one of the blocks produced by $\mathcal{P}$ (although possibly not everybody adopts the same block), and, since all these blocks are part of the (equally-long) longest chain (at that point), eventually one of these blocks will be output in the chain of the observer. Consequently, it holds that $\rho_{\mathcal{P},\sigma_{S_\mathcal{P}}} = \rho_{\mathcal{P},\sigma_\Pi}$.

Second, the maximum additional utility that a party $\mathcal{P}$ may receive by deviating from the honest protocol via producing conflicting blocks is: $\max_{S_\mathcal{P} \in \mathbb{S}_{-\mathcal{X}_{conf}}} \{\rho_{\mathcal{P},\sigma_\Pi} \cdot (x_{\sigma_{S_\mathcal{P}}} - x_{\sigma_\Pi}) + b_{\mathcal{P},\sigma_{S_\mathcal{P}}}\}$. Therefore, if for at least one party this value is non-negligible, $\epsilon$ is not small enough and so the protocol is not compliant. □

In the above negative results, non-compliance arises in (a) systems that employ fair rewards and (b) PoS systems where a party is incentivized to produce multiple conflicting messages, i.e., under a lossy network or multiple leaders per slot.

Regarding (a), Section 7.4 shows that fair rewards ensure compliance under utility *Reward*, but non-compliance regarding profit. Specifically, assuming a minimal participation cost $C^\perp_\mathcal{P}$, we showed that, if $\mathcal{P}$ abstains, they incur zero cost without any reward reduction. To explore compliance of fair rewards under externalities, we consider two strategy profiles $\sigma_\Pi, \sigma_{S_\mathcal{P}}$, as before. Notably, $S_\mathcal{P}$ is the abstaining strategy which, as shown in Section 7.4, maximizes utility in the standalone setting. For the two profiles, the profit for $\mathcal{P}$ becomes $\rho_{\mathcal{P},\sigma_\Pi} \cdot x_{\sigma_\Pi} - c_{\mathcal{P},\sigma_\Pi}$ and $\rho_{\mathcal{P},\sigma_{S_\mathcal{P}}} \cdot x_{\sigma_{S_\mathcal{P}}} + b_{\mathcal{P},\sigma_{S_\mathcal{P}}}$ respectively. Again, in both cases the party's rewards are equal. Therefore, since it holds that $C^\perp_\mathcal{P} \leq c_{\mathcal{P},\sigma_\Pi}$, $\mathcal{P}$ is incentivized to be $(\epsilon, \mathcal{X}_{conf})$-compliant (for some $\epsilon$) if:

$$\begin{aligned} \rho_{\mathcal{P},\sigma_{S_\mathcal{P}}} \cdot x_{\sigma_{S_\mathcal{P}}} + b_{\mathcal{P},\sigma_{S_\mathcal{P}}} &\leq \rho_{\mathcal{P},\sigma} \cdot x_{\sigma_\Pi} - c_{\mathcal{P},\sigma_\Pi} + \epsilon \Rightarrow \\ c_{\mathcal{P},\sigma_\Pi} + b_{\mathcal{P},\sigma_{S_\mathcal{P}}} &\leq \rho_{\mathcal{P},\sigma_\Pi} \cdot (x_{\sigma_\Pi} - x_{\sigma_{S_\mathcal{P}}}) + \epsilon \Rightarrow \\ C^\perp_\mathcal{P} + b_{\mathcal{P},\sigma_{S_\mathcal{P}}} &\leq \rho_{\mathcal{P},\sigma_\Pi} \cdot (x_{\sigma_\Pi} - x_{\sigma_{S_\mathcal{P}}}) + \epsilon \end{aligned} \tag{7.8}$$

If the abstaining strategy yields no external rewards, as is typically the case, $b_{\mathcal{P},\sigma_{S_\mathcal{P}}} = 0$, so the exchange rate needs to only counterbalance the minimal participation cost.

Regarding (b), we consider single-leader PoS under a lossy network, since the analysis is similar for multi-leader PoS. We again consider two strategy profiles $\sigma_\Pi, \sigma_{S_\mathcal{P}}$ as above. Now, under $\sigma_{S_\mathcal{P}}$, $\mathcal{P}$ produces $k$ blocks during each slot for which it is leader, to increase the probability that at least one of them is output in the observer's final

chain. Also, for simplicity, we set $b_{\mathcal{P},\sigma_{S_{\mathcal{P}}}} = 0$. These PoS systems become $(\epsilon', \mathcal{X}_{conf})$-compliant (for some $\epsilon'$) if:

$$\begin{aligned} \rho_{\mathcal{P},\sigma_{S_{\mathcal{P}}}} \cdot x_{\sigma_{S_{\mathcal{P}}}} &\leq \rho_{\mathcal{P},\sigma_{\Pi}} \cdot x_{\sigma_{\Pi}} + \epsilon \Rightarrow \\ (1 - d^k \cdot (1-d)^2) \cdot R_{max} \cdot x_{\sigma_{S_{\mathcal{P}}}} &\leq (1 - d \cdot (1-d)^2) \cdot R_{max} \cdot x_{\sigma_{\Pi}} + \epsilon \Rightarrow \\ x_{\sigma_{S_{\mathcal{P}}}} &\leq \frac{1 - d \cdot (1-d)^2}{1 - d^k \cdot (1-d)^2} \cdot x_{\sigma_{\Pi}} + \epsilon' \end{aligned} \tag{7.9}$$

where $R_{max} = R \cdot \sum_{i \in [1,e]} l_e \cdot \mu_{\mathcal{P},i}$ and $d, R, l_e$ are as in Subsection 7.5.2.2.

### 7.6.3 Attacks and Market Response

To estimate the exchange rate's behavior vis-à-vis external infraction rewards, we turn to historical data from the cryptocurrency market. Although no infractions of the type considered in this work have been observed in deployed PoS systems, we extrapolate data from similar attacks against PoW cryptocurrencies (Table 7.1).

In the considered attacks, the perpetrator $\mathcal{A}$ performed double spending. Specifically, $\mathcal{A}$ created a fork and two conflicting transactions, each of which is published on the two chains of the fork, the main and the adversarial chain. The main chain's transaction is redeemed for external rewards, e.g., a payment in USD, while the adversarial chain's transaction simply transfers the assets between two accounts of $\mathcal{A}$. Therefore, $\mathcal{A}$ both receives external rewards and retains its cryptocurrency rewards.

The adversarial chain contains a number of blocks created by $\mathcal{A}$. After this chain becomes longest and is adopted by the network, $\mathcal{A}$ sells its block rewards for USD. To evaluate the exchange rate at this point, we set the period between the launch of the attack and the (presumable) selling of the block rewards to $5$ days. This value depends on various parameters. For instance, in Bitcoin, the rewards for a block $\mathcal{B}$ can be redeemed after a "coinbase maturity" period of $100$ confirmations, i.e., after at least $100$ blocks have been mined on top of $\mathcal{B}$ (equiv. $17$ hours).[13] Furthermore, transactions are typically not finalized immediately; for instance, most parties finalize a Bitcoin transaction after $6$ confirmations and an Ethereum transaction after $240$ confirmations (equiv. approximately $1$ hour). Usually this restraint is tightened [Voe20] after an attack is revealed.

To estimate the difference in rewards that an infraction effects, we use cryptocurrency prices from CoinMarketCap[14]. First, we obtain the price $P_C$ of each cryptocur-

[13] A Bitcoin block is created on expectation every $10$ minutes.

[14] `https://coinmarketcap.com/`

rency $C$ $5$ days after the attack. Second, we compute the percentage difference $p_{BTC}$ of Bitcoin's price, between the end and the beginning of the $5$ day period. The value $P_C \cdot p_{BTC}$ expresses the *expected* price of the cryptocurrency, assuming no attack had occurred.[15] Next, we find the number of blocks $b$ created during the attack and the reward $R$ per block. Therefore, the reward difference is computed as $P_C \cdot p_{BTC} \cdot b \cdot R$.

As shown in [GKR20, DKT+20], this attack is optimal. Therefore, using the computations in [Nak08a] and the reorganized blocks during each attack, we approximate the necessary percentage of adversarial power to successfully mount the attack with at least $0.5$ probability.

| System | Date | External Utility | Rewards | Reward Difference | Attacker Hash Rate % |
|---|---|---|---|---|---|
| Ethereum Classic | 5/1/19 [Nes19] | $\$1.1$M | $\$12.410$ | $\$-2,646$ | $0.48026$ |
| | 1/8/20 [Stu20a] | $\$5.6$M | $\$84,059$ | $\$-11,806$ | $0.4913$ |
| | 6/8/20 [Stu20b] | $\$1.68$M | $\$91,715$ | $\$-5,761$ | $0.4913$ |
| | 30/8/20 [Voe20] | unknown | $\$120,131$ | $\$-12,992$ | $0.5$ |
| Horizen | 8/6/18 [Zen18] | $\$550,000$ | $\$5,756$ | $\$-752$ | $0.461373$ |
| Vertcoin | 2/12/18 [Nes18] | $\$100,000$ | $\$3,978$ | $\$-879$ | $0.487124$ |
| Bitcoin Gold | 16/5/18 [Osb18] | $\$17.5$M | $\$11,447$ | $\$-1,404$ | $0.441631$ |
| | 23/1/20 [Lov20] | $\$72,000$ | $\$4,247$ | $\$814$ | $0.43991$ |
| Feathercoin | 1/6/13 [Max13a] | $\$63,800$ | $\$1,203$ | $\$-95.73$ | $0.48283$ |
| Litecoin Cash | 4/7/19 [Lov19] | $\$5,511$ | $\$80.66$ | $\$-0.15$ | $0.47167$ |

Table 7.1: Double spending attacks and the market's response to them. External utility is estimated as the reward from double-spent transactions. To compute the reward difference, we multiply the rewards from reorganized blocks with the exchange rate difference, i.e., the asset's price $5$ days after the attack minus the expected price, if an attack had not occurred (following Bitcoin's price in the same period).

[15] Historically, the prices of Bitcoin and alternative cryptocurrencies are strongly correlated [L.18].

### 7.6.4 Penalties

Table 7.1 shows that, historically, the attacks are profitable, so the market's response is insufficient to keep a party compliant. Interestingly, in many occasions the external utility was so high that, even if the exchange rate became $0$, it would exceed the amount of lost rewards. Therefore, an additional form of utility reduction is necessary. In many PoS systems, like Casper [BG17, BRLP19], Gasper [BHK+20], and Tezos [Tez20], this decrease is implemented as penalties for misbehavior.

In detail, each party $\mathcal{P}$ is required to lock an amount of assets in a deposit $g_{\mathcal{P}}$. If $\mathcal{P}$ violates a well-defined condition, its deposit is forfeited, along with its accumulated rewards. Typically, misbehavior is identified by a non-interactive proof of the misbehavior, e.g., a signed malicious message.

Considering profiles $\sigma_{\Pi}, \sigma_{S_{\mathcal{P}}}$ as before, $\mathcal{P}$'s rewards take shape as follows. Under $\sigma_{\Pi}$, $\mathcal{P}$ receives $\rho_{\mathcal{P},\sigma_{\Pi}}$, which are exchanged under rate $x_{\sigma_{\Pi}}$; also, it retains its deposit $g_{\mathcal{P}}$, which is exchanged under the same rate. Under $\sigma_{S_{\mathcal{P}}}$, $\mathcal{P}$ forfeits both its rewards and deposit, but receives external utility $b_{\mathcal{P},\sigma_{S_{\mathcal{P}}}}$. Therefore, under penalties a party is incentivized to be compliant if the forfeited deposit and rewards are larger than the external utility (cf. Theorem 13).

**Theorem 13.** *Assume i) a synchronous network (cf. Section 7.1.1), ii) the conflicting predicate $\mathcal{X}_{conf}$, and iii) that $\forall \mathcal{P} \in \mathbb{P} : \mu_{\mathcal{P}} < \frac{1}{2}$. Also let: i) $\mathbb{S}_{-\mathcal{X}_{conf}}$: the set of all non-compliant strategies; ii) $x_{\sigma_{\Pi}}$: the (expected) exchange rate under $\sigma_{\Pi}$; iii) $x_{\sigma_{S_{\mathcal{P}}}}$: the (expected) exchange rate when only $\mathcal{P}$ employs some non-compliant conflicting strategy $S_{\mathcal{P}}$; iv) $b_{\mathcal{P},\sigma_{S_{\mathcal{P}}}}$: the external utility that $S_{\mathcal{P}}$ yields for $\mathcal{P}$. Finally, assume the block-proportional rewards (cf. Definition 36 for fixed block reward $R$) for which it also holds:*

$$\forall \mathfrak{I}\, \forall \mathcal{P} \in \mathbb{P} : R_{\mathcal{P},\mathfrak{I}} = \begin{cases} \varrho(\mathcal{C}_{\Omega,\mathfrak{I}}, \mathcal{P}) \cdot \mathcal{R}_{\Omega,\mathfrak{I}} + g_{\mathcal{P}}, & \mathcal{P} \text{ produces no conflicting blocks in } \mathfrak{I} \\ 0, & \text{otherwise} \end{cases}$$

*for a protocol-specific deposit value $g_{\mathcal{P}}$ with $\rho_{\mathcal{P},\sigma} = E[R_{\mathcal{P},\mathcal{E}_{\sigma}}]$, i.e., the expected rewards of $\mathcal{P}$ under profile $\sigma$.*

*Ouroboros with the above rewards and under the aforementioned externalities is not $(\epsilon, \mathcal{X}_{conf})$-compliant (cf. Definition 32) w.r.t. utility Reward (cf. Definition 34) and, if $R > C$, it is also not $(\epsilon, \mathcal{X}_{conf})$-compliant w.r.t. utility Profit, in both cases for a bound $\epsilon < \max_{\mathcal{P} \in \mathbb{P}} \{ \max_{S_{\mathcal{P}} \in \mathbb{S}_{-\mathcal{X}_{conf}}} \{ b_{\mathcal{P},\sigma_{S_{\mathcal{P}}}} \} \} - \rho_{\mathcal{P},\sigma_{\Pi}} \cdot x_{\sigma_{\Pi}} - \mathsf{negl}(\kappa)$.*

*Proof.* When a party $\mathcal{P}$ employs a non-compliant strategy $S_{\mathcal{P}}$, it receives an external utility $b_{\mathcal{P},\sigma_{S_{\mathcal{P}}}}$. Also, in that case, it produces conflicting blocks (due to the non-compliance property of $S_{\mathcal{P}}$). Therefore, by definition of the above block-proportional

rewards, $R_{\mathcal{P},\sigma_{S_\mathcal{P}}} = 0$; in other words, when $\mathcal{P}$ employs $S_\mathcal{P}$ and produces conflicting blocks, $\mathcal{P}$ forfeits the protocol's rewards (which include the original rewards plus the deposit $g_\mathcal{P}$). Therefore, when $\mathcal{P}$ employs $S_\mathcal{P}$ and all other parties employ $\Pi$, $\mathcal{P}$'s utility is $U_\mathcal{P}(\sigma_{S_\mathcal{P}}) = b_{\mathcal{P},\sigma_{S_\mathcal{P}}}$ (cf. Definition 37).

We also remind that (from the proof of Theorem 9), $\mathcal{P}$ can bias the leader schedule with some negligible probability $negl(\kappa)$, if it controls a minority of power.

Therefore, for any party $\mathcal{P}$ and strategy $S_\mathcal{P}$, $\sigma_{S_\mathcal{P}}$ is directly $\epsilon$-reachable from $\sigma_\Pi$ if:

$$\rho_{\mathcal{P},\sigma_\Pi} \cdot x_{\sigma_\Pi} + \mathsf{negl}(\kappa) + \epsilon < b_{\mathcal{P},\sigma_{S_\mathcal{P}}} \Leftrightarrow$$
$$\epsilon < b_{\mathcal{P},\sigma_{S_\mathcal{P}}} - \rho_{\mathcal{P},\sigma_\Pi} \cdot x_{\sigma_\Pi} - \mathsf{negl}(\kappa)$$

Across all parties and all non-compliant strategies, the maximum such $\epsilon$ is:

$$\epsilon < \max_{\mathcal{P}\in\mathbb{P}}\{\max_{S_\mathcal{P}\in\mathbb{S}_- x_{conf}}\{b_{\mathcal{P},\sigma_{S_\mathcal{P}}}\}\} - \rho_{\mathcal{P},\sigma_\Pi} \cdot x_{\sigma_\Pi} - \mathsf{negl}(\kappa)$$

We note that, as shown in Theorem 9, Ouroboros with block proportional rewards under a synchronous network is an equilibrium, i.e., the honest protocol yields the maximum rewards for each party compared to all other strategies. In the present setting, the honest protocol again yields the maximum utility, compared to all other *compliant* strategies. To prove this it suffices to observe that, if a party does not produce conflicting blocks, its rewards are a linear function of the rewards of the setting of Theorem 9; therefore, between compliant strategies, the honest protocol yields the maximum rewards (as shown in Theorem 9).

Therefore, the given bound of $\epsilon$ is bounded from below and, for $\epsilon$ less than this bound, there exists a party $\mathcal{P}$ that is incentivized to employ a (non-compliant) strategy $S_\mathcal{P}$ and produce conflicting blocks, rendering $\Pi$ not $(\epsilon, \mathcal{X}_{conf})$-compliant. □

An issue of both externality-based Theorems (12 and 13) is that $\epsilon$'s bound depends on $b_{\mathcal{P},\sigma_{S_\mathcal{P}}}$. This bound may be high (i.e., non-negligible), but it does not depend on protocol parameters, but instead depends on external (to the protocol) variables (e.g., the payoff of a double-spending attack). Therefore, it is outside of the control of the protocol's designer, who can only ensure that a protocol is secure by only *assuming* an upper bound on this external payoff.

Intuitively though, $\epsilon$'s bound shows that attacks are prevented by two parameters. First, the larger the deposit, the more attacks it protects against. For example, a deposit of $\$10$M would protect against all attacks of Table 7.1. However, large deposits also shut off small parties, who do not own adequate assets. Therefore, a tradeoff exists

in preventing double spending attacks and enabling widespread participation. Second, the longer the duration of an attack, the more blocks an adversary needs to produce, hence the larger the rewards that it forfeits. Typically, the attack duration depends on the number of confirmations that the offended party, e.g., an exchange, enforces. Therefore, enforcing different confirmation limits, based on a transaction's value, would both enable fast settlement for small transfers and protect large transactions.

Taking this observation into account, we briefly review users' behavior, when running Ouroboros (cf. Section 7.5.2.1) under deposits and penalties. In an Ouroboros execution, it is possible to identify the percentage of parties that actively participate during each epoch, by observing the block density and the number of empty slots (i.e., when no block is diffused). Therefore, it is possible to estimate the level of infraction, i.e., double-signing, that a party needs to perform to mount a double-spending attack, and then enforce a transaction finalization rule to disincentivize such attacks.

Let $\mathcal{P}$ be a user of an Ouroboros ledger. $\mathcal{P}$ enforces a rule of $k$ confirmations, i.e., finalizes a transaction after it is "buried" under $k$ blocks. Let $\tau$ be a transaction published in a block on slot $r$, with value $v_\tau$. After $l$ slots, $\mathcal{P}$ sees that $\tau$ has been buried under $b$ blocks, with $b = x \cdot l$ for some $x > 0.5$. Therefore, $(1-x) \cdot 100\%$ of slots are – seemingly – empty. $\mathcal{P}$ will (on expectation) confirm $\tau$ after $\frac{1}{x} \cdot k$ slots, i.e., when $k$ blocks are produced; of these, $\frac{1-x}{x} \cdot k$ are empty.

Let $\mathcal{A}$ be a party that wants to double-spend $\tau$. $\mathcal{A}$ should produce a private chain with at least $k$ blocks. Of these, at most $\frac{1-x}{x} \cdot k$ correspond to the respective empty slots, while $k - \frac{1-x}{x} \cdot k = \frac{2 \cdot x - 1}{x} \cdot k$ conflict with existing blocks, i.e., are evidence of infraction.

Let $d$ be a deposit amount, which corresponds to a single slot. Thus, for a period of $t$ slots, the total deposited assets $D = t \cdot d$ are distributed evenly across all slots. $\mathcal{A}$ can be penalized only for infraction blocks, i.e., for slots which showcase conflicting blocks. In a range of $\frac{1}{x} \cdot k$ slots, infraction slots are $\frac{2 \cdot x - 1}{x} \cdot k$.

Therefore, $\mathcal{A}$ forfeits at most $\frac{2 \cdot x - 1}{x} \cdot k \cdot d$ in deposit and $\frac{2 \cdot x - 1}{x} \cdot k \cdot R$ in rewards that correspond to infraction blocks. Therefore, if $v_\tau > \frac{2 \cdot x - 1}{x} \cdot k \cdot (d + R)$, $\mathcal{A}$ can profitably double-spend $\tau$. Consequently, depending on the amount $d$ of deposit per slot, the block reward $R$, and the rate $(1-x)$ of empty slots, for a transaction $\tau$ with value $v_\tau$, $\mathcal{P}$ should set the confirmation window's size to:

$$k_\tau > \frac{v_\tau}{\frac{2 \cdot x - 1}{x} \cdot (d + R)} \tag{7.10}$$

Finally, the system should allow each participant to withdraw their deposit after

some time. However, it should also enforce *some* time limit, to ensure that adequate deposit exists to (possibly) enforce a penalty. Intuitively, a party $\mathcal{P}$ should be able to withdraw a deposit amount that corresponds to a slot $r$, only if no transaction exists, such that $r$ is part of the window of size $k$ (with $k$ computed as above). In other words, $\mathcal{P}$'s deposit should be enough to cover all slots, which $\mathcal{P}$ has led and which are part of the confirmations' window of at least one non-finalized transaction.

**Remark.** *Penalties are applicable, and indeed common, in PoS systems, like those mentioned above, but the same does not hold for PoW systems. In PoW protocols like Bitcoin, the block producers are decoupled from the users of the system, thus it is often impossible to identify which party produced each block. Naturally, the same is true for fully anonymous protocols, both PoW [MGGR13, BCG+14] and PoS [GOT19, KKKZ19].*

# Chapter 8

# Macroeconomic Principles

Chapter 7 investigated distributed ledger systems from a microeconomic perspective, focusing on the incentives and strategic choices of individual parties. Nonetheless, if such systems are to support real-world economies, a macroeconomic treatment is also imperative. In this chapter, we provide some preliminary results on this line of research.

Section 8.1 explores the limitations in enforcing macroeconomic policies in distributed ledgers. Importantly, we show that, in decentralized anonymous systems, no macroeconomic policy which redistributes wealth from the larger to the smaller parties can be applied. Instead, the best one can hope for is a linear increase in each party's wealth, proportionally to their capital. To quantify how different systems fare in this regard, we introduce the notion of "cryptocurrency egalitarianism", a quantitative metric that helps compare systems w.r.t. how much they favor wealthy investors.

Next, Section 8.2 explores how taxation could be enforced. Given the previous impossibility result, we assume the existence of a centralized taxation authority. Our goal is to enable the authority to correctly identify the users' assets, in order to enforce its taxation policy, in a privacy-preserving and efficient manner. In that direction, we propose two schemes based on programmable money, i.e., currency which is transferable as long as certain preconditions are met.

## 8.1 Cryptocurrency Egalitarianism

In almost all blockchain cryptocurrency systems, block generators are incentivized to participate via block rewards, i.e., for each block they successfully produce and which is subsequently adopted by all other participants. In many cryptocurrencies, the rewards serve a dual purpose: incentivise the the miners/minters but also create and distribute

the underlying cryptocurrency to the system's maintainers. These rewards follow various schedules that are designed based on the macroeconomic desiderata envisioned by the architects of the cryptocurrency. For example, the rate of coin production is halved every $210,000$ blocks in Bitcoin. Ethereum and Litecoin follow similar schedules. On the contrary, Monero has a smooth emission schedule in which the rewards are gradually reduced at every new block generated. The question of what this schedule should be can have significant impact on the variance of stake ownership after an execution of a sufficient number of protocol rounds [FKO+19]. Taking this into account, in this chapter we consider the block generators as investors and focus on the comparison of the expected returns of investors with different purchasing power.

A central economic property that arises from this line of thought is *cryptocurrency egalitarianism* (also "crypto-egalitarianism"). This property states that rewards should be proportional to the invested capital. Therefore, wealthy investors should not be disproportionately rewarded, but everybody should have equal opportunity to both participate and earn rewards. Until now, the term crypto-egalitarianism has been left undefined, although several cryptocurrencies claim to be more egalitarian than others [VS13, McM13]. However, lacking a quantifiable metric, the discussion around egalitarianism remains ill-posed. The core contribution of this chapter is to put forth the first concrete definition of egalitarianism, in a way which is generic, practically measurable, and applicable to any cryptocurrency. Additionally, we show that wealth redistribution, from the rich to the poor, is impossible in decentralized, anonymous (or pseudonymous) systems; thus, rewarding everybody proportionately to their capital is the best achievable setting.

**Related work.** The macro and microeconomics of blockchain design have been studied from several perspectives, but remain an active area of research with a number of open questions. *Egalitarianism* in particular has been studied in PoW systems from the perspective of *memory-hard functions* [ABP17, BK16]. These works operate under the premise that memory hardness provides egalitarianism, in the sense that the cost of one computational step is roughly the same irrespective of the underlying computational platform (typically ASIC vs. generic). In this chapter we generalize this question, by asking whether computational power grows proportionally to capital invested, i.e., whether larger wealth results disproportionately more rewards. Additionally, a notable work by Fanti *et al.* [FKO+19] introduces the complementary notion of *equitability*. That work studies the evolution of a system after a series of rounds, putting forth the property that

stake ownership remains in proportion *before* and *after* rewards have been awarded. By studying the behavior of the returns' *variance* under the randomness of executions, they show that the distribution of capital follows a Pólya process. This chapter can be seen as complementary to their results, by quantifying the *expectation* of rewards and then studying the variance under the randomness of initial capital allocation. Therefore, a cryptocurrency can be perfectly egalitarian and poorly equitable and vice versa; notably, it is possible to obtain a cryptocurrency both egalitarian and equitable, by adopting correctly parameterized PoS under a geometric reward function.

### 8.1.1 PoW vs. PoS

Before studying the egalitarianism of different cryptocurrency consensus mechanisms, we consider the leader election process, to establish an understanding of the differences in egalitarianism between the two models of PoW and PoS.

**Proof-of-Work.** The number of hash evaluations is one of the several critical parameters to consider when purchasing mining hardware. Other important parameters include the price of a mining unit, as well as its electricity consumption. Mining hardware is divided in various tiers based on performance, namely CPU miners, GPU miners, FPGA miners, and specialized ASIC miners [Tay13]. Although the pricing of such devices may be similar, the hashing rate and, in turn, the Return on Investment, is highly dependent on the hardware's tier. For example, as of December 2018, the mining hardware "Whatsminer M10" produced by the company "MicroBT" cost $\$1,022.00$ per unit and produces $\$0.104266$ per hour of operation in net gains, i.e., average mined Bitcoins per hour denominated in US dollars minus the electricity costs. On the other hand, the mining hardware "8 Nano Pro", produced by the company "ASICMiner", cost $\$6,000.00$ per unit, but produces $\$0.315327$ per hour of operation in net gains, i.e., almost three times the hourly net gains of its cheaper competitor. Thus, if one can afford to purchase the more expensive hardware, each of their subsequent dollar invested in electricity returns more mined coins.

It has long been folklore knowledge in the blockchain community that mining becomes more egalitarian by using a memory-hard PoW function. This intuition is correct, the core reason being the difficulty to construct specialized hardware for memory-hard functions. For example, no ASICs currently exist for Monero mining. Therefore, the only way to scale mining operations is by purchasing more general purpose hardware.

However, since the mining hardware in this case varies little, both in terms of cost and performance, scaling returns become proportional to investments.

In this chapter, we only analyze the scaling of the economics of mining with respect to hardware. We also do not take into account basic costs such as shipping and the availability of a basic machine to co-ordinate mining (such as a personal computer not performing mining itself). A multitude of additional factors play important roles for mining operations, such as space rental costs, machine cooling and maintenance costs, or bulk electricity purchase. As is common in economies of scale, these relative costs are reduced for large-scale operations, although they are similar for all PoW cryptocurrencies and thus do not affect relative comparisons between them. We also remark that we analyze mining costs for small capital investments. If larger capital, e.g., above a few million US dollars, is available, corporations can develop their own specialized hardware and gain a competitive advantage by treating it as a trade secret [Tay13]. These details make the comparison in favour of PoS *more pronounced*, as PoS operations do not incur such types of costs and do not lend themselves to specialized mining hardware research.

**Proof-of-Stake.** PoS is often criticized for its lack of egalitarianism. The rationale is that, in PoS, the more money one stakes, the more money one generates. Thus, "the rich get richer", which is precisely the opposite of egalitarianism. Additionally, in PoS systems, the money owners could constitute a *closed, rich club*, refusing to share the assets with any outsiders. In contrast, this argument claims, PoW is naturally egalitarian; everyone is paid based not on the money they own, but on the computational power they put to work. In this case, since computational power is a *natural* resource and cannot be exclusively owned, a closed rich club cannot be formed. Although this argument seems agreeable at first, the results of this chapter contradict it. In fact, correctly parameterized stake-based systems are much more egalitarian than work-based ones.[1]

It is instructive to dispel the above argument intuitively, before we support our position with data. First, the argument that money can be exclusively owned, but computational power cannot, is rather misguided. Indeed, this may be true in the case of a peculiar oligopoly, where a small faction of parties mutually agrees to never sell to outsiders, despite external demand. However, in an open market, both money and computational power can be freely purchased and, in fact, any non-negligible amount of computational power must be necessarily purchased that way. In this work, we assume an open market

[1] Variations of PoS, such as delegated PoS, may not be perfectly egalitarian, since the delegates, i.e., the leaders of the stake pools, typically earn extra profits for managing the stake pools [BKKS20].

for both mining hardware and financial capital, which allows open participation. Therefore, given that both money and computational power are purchasable, we consider the funds one invests, either in technology or in financial capital, in order to maximize the returns from a cryptocurrency's block generation mechanisms. The amount of cryptocurrency generated by a given investment can be concretely measured and compared, thus the question can now be analyzed quantitatively and answered concretely.

### 8.1.2 A Formal Model of Crypto-Egalitarianism

The core contribution of this chapter is a formal definition of an economic measure of *egalitarianism* in cryptocurrencies.

Before we present our definition, let us first state the *desiderata* of such a definition. First, we want to enable concrete measurements on cryptocurrencies, in a manner that is quantitative and not vague. Thus far, egalitarianism claims have been rather informal, failing to include exact data [VS13, McM13]. As such, different cryptocurrencies claim egalitarianism over the others, without demonstrating the claims or provide conclusive arguments. Second, an egalitarianism definition must measure the protocol maintenance returns of smaller vs. larger investors. We thus desire a metric which extracts a smaller value to indicate a *lack of egalitarianism* (e.g., when large wealth generates blocks disproportionately faster than small wealth), or a larger value to indicate *perfect egalitarianism* (i.e., when every invested dollar has exactly equal power in terms of cryptocurrency generation).

The first step in establishing our crypto-egalitarianism definition is to define the *egalitarian curve* $f$. The horizontal axis of this curve plots the financial capital, which is available for investment, denominated in a fiat currency (USD).[2] The vertical axis plots the Return On Investment (ROI), which measures the amount of cryptocurrency that is created during the investment period and remains unspent at the end of it, given an optimal allocation of the initial capital. We require that ROI is computed over *freshly generated* cryptocurrency; thus, it must be newly-mined or minted, and not purchased from existing investors. Finally, the curve is plotted with a fixed investment duration in mind; naturally, curves of different cryptocurrencies can be compared only if they use the same duration.

**Definition 38** (Egalitarian curve)**.** *Given a cryptocurrency $c$ and the set of all possible in-*

[2] Given that we explore a small investment duration, it makes little difference whether these are nominal USD or real USD, as long as they are the same when applying comparisons.

*vestment strategies $\mathbb{B}$, we define the egalitarian curve $f_{c,d} : \mathbb{R}^+ \longrightarrow \mathbb{R}^+$ of $c$ for an investment period $d$ as:*

$$f_{c,d}(v) = \frac{\max_{B \in \mathbb{B}} E[B(v)] - v}{v} \tag{8.1}$$

The value $\max_{B \in \mathbb{B}} E[B(v)]$ identifies the maximum expectation of returns across all investment strategies $\mathbb{B}$, i.e., the amount of returns which the *optimal* strategy ensures for a given initial capital $v$. The blockchain execution is modeled as a random variable, since returns vary by execution; specifically, the randomness of the execution can affect returns, as a participant may be "lucky", i.e., produce more blocks than expected [FKO+19].

We remark that we *do* allow strategies to reinvest capital. For instance, returns earned from mining rewards can be reinvested in electricity costs for future mining. Furthermore, for unit consistency, we assume the strategy $B(v)$ returns the freshly generated coins denominated in the same units as the capital $v$. Second, we assume participants act independently and follow the protocol according to its specifications.

Using our definition of the egalitarian curve, we now define (Definition 39) egalitarianism as a concrete number. Considering the initial capital $v$ as a random variable, which follows a certain distribution $\mathcal{D}$, egalitarianism is the variance of the expected ROI, when the capital is chosen from the given distribution.

**Definition 39** (Egalitarianism)**.** *Given a cryptocurrency $c$ and its egalitarian curve $f$, we define the egalitarianism $e$ of $c$, for investment duration $d$ under initial capital distribution $\mathcal{D}$, as follows:*

$$e_{c,d,\mathcal{D}} = -\mathsf{Var}_{v \leftarrow \mathcal{D}}[f_{c,d}(v)] \tag{8.2}$$

The intuition behind this definition is that, to have egalitarianism, the ROI must remain the same across different capital investments. As such, any deviation from the mean is non-egalitarian. Naturally, if a system's egalitarianism is *higher* than another, we say that the former is *more egalitarian* than the latter. Of course, to be accurate, such comparisons must be made after fixing the parameters $c$ and $d$, as well as the initial capital distribution $\mathcal{D}$. Fixing $\mathcal{D}$ to be the uniform distribution between a minimum and a maximum capital, the returns are the same for all initial capitals alike.

Based on the above, we can define the *ideal egalitarian curve*. First, as an interesting thought experiment, we consider the egalitarian curve which is decreasing (and is, arguably, *the* ideal curve). In this case, small investors would receive proportionally

more newly created cryptocurrencies for every dollar they invest, i.e., the system would redistribute wealth from the rich to the poor. However, one can quickly see that, in decentralized cryptocurrencies where the identities of the participants are unknown, this is impossible. Indeed, the fact that decentralized cryptocurrencies allow anonymous generation of new identities [Dou02] allows a wealthy investor to split their capital into smaller ones. Thus, if the curve were ever to have a negative slope, the sum of the smaller splits of the rich investment would achieve a higher gain. By the definition of the curve, which mandates that it depicts the ROI of an *optimal* investment, this would be a contradiction. Corollary 3 makes this intuition more precise.

**Corollary 3** (Sybil strategies)**.** *Fix a cryptocurrency $c$ and an investment period interval $d$. Given capital $v$, for every natural number $i \in \mathbb{N}^\star$, it holds that $f_{c,d}(v) \leq f_{c,d}(i \cdot v)$.*

*Proof.* We prove the statement by contradiction. Assume that, for some capital $v$, there exists a natural number $i \in \mathbb{N}^\star$ such that $f_{c,d}(v) > f_{c,d}(i \cdot v)$. Also assume that, for capital $v$, the optimal strategy is $B'$, so $\max_{B \in \mathbb{B}} \mathbb{E}[B(v)] = \mathbb{E}[B'(v)]$. For capital $i \cdot v$, there exists a strategy $B''$, such that the capital is split into $i$ equally-sized parts, with the strategy $B'$ applied on each part. Given that the execution of each such sub-strategy is independent, the expected returns for $B''$ are:

$$\mathbb{E}[B''(i \cdot v)] = i \cdot \mathbb{E}[B'(v)] = i \cdot \max_{B \in \mathbb{B}} \mathbb{E}[B(v)] \tag{8.3}$$

Additionally, $B''$ is at best the optimal strategy, so:

$$\max_{B \in \mathbb{B}} \mathbb{E}[B(i \cdot v)] \geq \mathbb{E}[B''(i \cdot v)] \overset{(8.3)}{\Longrightarrow} \max_{B \in \mathbb{B}} \mathbb{E}[B(i \cdot v)] \geq i \cdot \max_{B \in \mathbb{B}} \mathbb{E}[B(v)] \tag{8.4}$$

However, it should also hold that:

$$\begin{gathered} f_{c,d}(v) > f_{c,d}(i \cdot v) \Rightarrow \\ \frac{\max_{B \in \mathbb{B}} \mathbb{E}[B(v)] - v}{v} > \frac{\max_{B \in \mathbb{B}} \mathbb{E}[B(i \cdot v)] - i \cdot v}{i \cdot v} \overset{(8.4)}{\Longrightarrow} \\ \frac{\max_{B \in \mathbb{B}} \mathbb{E}[B(v)] - v}{v} > \frac{i \cdot \max_{B \in \mathbb{B}} \mathbb{E}[B(v)] - i \cdot v}{i \cdot v} \Rightarrow \\ \frac{\max_{B \in \mathbb{B}} \mathbb{E}[B(v)] - v}{v} > \frac{\max_{B \in \mathbb{B}} \mathbb{E}[B(v)] - v}{v} \end{gathered}$$

which is impossible. □

Corollary 3 shows that, in purely decentralized systems, a decreasing egalitarian curve is impossible. Therefore, the next-best ideal curve is a constant one, where the

ROI is stable regardless of capital invested. Under this condition, the amount of freshly generated cryptocurrency is exactly proportional to the money invested. Consequently, a cryptocurrency with an ideal egalitarian curve is perfectly egalitarian (Definition 40).

**Definition 40** (Perfect egalitarianism)**.** *A cryptocurrency $c$ is perfectly egalitarian, for investment duration $d$ and initial capital distribution $\mathcal{D}$, if $e_{c,d,\mathcal{D}} = 0$.*

### 8.1.3 Discussion

In this chapter, in providing a concrete definition of crypto-egalitarianism, we enable an evidence-based discussion to substitute folklore arguments. The first application of this metric was provided in [KKNZ19], which compared some of the largest cryptocurrency systems to date. Using our model, the egalitarianism of four indicative PoW-based cryptocurrencies (Bitcoin, Litecoin [Lee11], Ethereum [B$^+$14, Woo14], and Monero [VS13]) was measured. The assessed claims of these projects were found in agreement with our data, thus presenting for the first time economic comparisons which quantify them precisely. On the pure PoS side, it was shown that egalitarian behavior is similar across all coins, independently of externalities such as hardware characteristics. Therefore, it suffices to perform a case study of an indicative PoS protocol (in this case, Ouroboros [KRDO17]). It was then shown that pure PoS coins can be perfectly egalitarian, contrary to their PoW counterparts. These results were very optimistic in terms of usability of our metric, as they provide concrete figures which measure the egalitarianism of several popular cryptocurrencies. The most unexpected result arised from the comparison between the PoW and PoS mechanisms. Although blockchain folklore argued in favour of PoW systems in terms of egalitarianism, these results show that, in fact, it is PoS systems which are more egalitarian under our proposed model.

Another interesting property that arises from this work is the impossibility of wealth redistribution in cryptocurrency systems. As shown in Corollary 3, in a purely decentralized setting, where no real-world identity checks exist (and, arguably, cannot exist), a wealthy participant can always pose as multiple poor users. Therefore, any attempt to redistribute wealth from the rich to the poor, based on entirely technological tools and without taking into account real-world social structures, seems doomed to fail. In conclusion, all decentralized cryptocurrencies are “rich get richer” schemes; the pertinent question, which this chapter aimed at resolving, is *how fast* this takes place.

## 8.2 Tax Applications of Programmable Money

A tax gap [Gro18] is a difference between the reported and the real tax revenue, for a given jurisdiction and period of time. Research estimated that the tax gap in the USA was $16.4\%$ of revenue owed [Ser16] between 2008-2010, the total loss throughout the EU due to the tax gap to €$151.5$ billion in 2015 [MGS18], while $\frac{1}{3}$ of taxpayers in the UK under-report their earnings [Adv20] (albeit half of UK's lost taxes are product of a small, wealthy fraction of misbehaving taxpayers). Therefore, reducing the tax gaps can significantly enhance the efforts of tax-collecting authorities.

Central bank digital currencies (CBDC) have also come to prominence in recent years. In the past decade, distributed ledger-based financial systems, which were kick-started with the creation of Bitcoin [Nak08a], were accompanied by the increasing digitalization of payments [BS11]. CBDCs are the culmination of these trends, enabling fast, cheap, and safe transactions in fiat assets. Crucially though, although still mostly on a research stage,[3] CBDCs have caused great concerns on citizens regarding transaction privacy [Ban21].

This chapter offers two mechanisms that facilitate tax auditing and the identification of tax gaps in distributed ledger-based currency systems. The first is a wrapper around a generic distributed ledger, which enables taxpayers to declare their assets directly to the authorities, while undeclared assets are frozen. The second is a proof mechanism that enables the sender of some assets to prove, in a privacy-preserving manner, whether the transferred assets have been taxed. Both mechanisms are examples of programmable money (also referred to as smart money [AHA17]), where currency is programmed to be transferable under a suitable set of circumstances or its transfer has specific implications.

**Related work.** Literature offers various works on auditing of distributed ledger-based assets. A holistic approach is taken in zkLedger [NVV18], which combines a permissioned ledger with zero-knowledge proofs to create a tamper-resistant, verifiable ledger of transactions. PRCash [WKCC18] also employs a permissioned ledger and offers a regulation mechanism that restricts the total amount of assets a user can receive anonymously for a period of time. Also Garman *et al.* [GGM16] propose an anonymous ledger, which can enforce specific transaction policies. Following, Section 8.2.2 aims at offering a simpler design, which can be more easily integrated in existing pseudonymous

[3] `https://cbdctracker.org` [July 2021]

distributed ledgers, compared to the aforementioned works. Another interesting research thread considers proofs of solvency. The first such scheme for Bitcoin exchanges, proposed by Maxwell [Wil14], leaks the total amount of both assets and liabilities of the exchange; more importantly, it enables an attack that allows the exchange to hide assets, as detailed by in Zeroledge [DSE], which also proposed a privacy-preserving system that allows exchanges to prove properties about their holdings. Provisions [DBB+15] is a zero-knowledge proof of solvency mechanism for Bitcoin exchanges, based on Sigma protocols i.e., without the need to reveal the addresses or the amount of assets that an exchange controls. Similarly, Agrawal *et al.* [AGM18] describe a proof of solvency which achieves better performance compared to Provisions, although assuming a trusted setup. The mechanism of Section 8.2.3 extends Provisions and is also applicable to [AGM18].

### 8.2.1 Desiderata

In distributed ledger-based currency systems, a user $\mathcal{U}$ manages their assets via addresses. Each address $\alpha$ is associated with a key pair $(\mathsf{vk}, \mathsf{sk})$, such that the private key sk is used to claim ownership of the assets, e.g., by signing special messages; typically $\alpha = \mathsf{H}(\mathsf{vk})$ for some hash function H. Each address $\alpha$ is associated with a (public) balance $\mathsf{bal}(\alpha)$ so, given a list $[\alpha_1, \ldots, \alpha_n]$ of all addresses that $\mathcal{U}$ controls, $\mathcal{U}$'s total assets are $\Theta = \sum_{i=1}^{n} \mathsf{bal}(\alpha_i)$. Our goal will be to retain as much privacy as possible, so $\Theta$ should be the only information that is leaked to $\mathcal{T}$, without de-anonymization of individual transaction data.

To showcase the limitations of current systems, consider the following example. Assume that Alice tax evades, i.e., creates a secret, undeclared address $\alpha$ and controls some assets $\theta$ in it. Given the pseudonymous nature of the ledger, $\alpha$ cannot be correlated with Alice, until she uses it. Following, Alice issues a transaction $\tau$ which sends $\theta$ assets from $\alpha$ to Bob. If Bob suspects that Alice evaded taxation for these $\theta$ assets, they might want to report her to the authorities for inspection. However, the complaint should be accompanied by a proof that $\alpha$ is controlled by Alice, i.e., a proof that Alice knows the private key associated with $\alpha$. This is necessary as $\mathcal{T}$ needs to distinguish between two scenarios: i) Alice controls $\alpha$ and tax evades; ii) Bob is lying about Alice owning $\alpha$. In the first scenario, Bob *does* know that $\alpha$ is controlled by Alice, but $\tau$ is not sufficient to prove it. Instead, Bob needs a proof which can only be supplied by Alice, e.g., a signature from Alice which acknowledges $\tau$ or $\alpha$. However, if Alice tax evades,

naturally she would not create such incriminating proof.

It is important that we retain as many good features of existing ledger systems as possible. The most notable such feature is transaction privacy, thus our work considers pseudonymous, Bitcoin-like levels of privacy, and minimizes the information leaked to the authorities during a tax auditing. Another important aspect is the mechanism's performance. A fundamental ingredient of payment systems is the seamless transaction experience, so it is important to allow users to transact at all times, while also avoiding significant strain during taxation periods. Finally, our mechanisms aim to minimize the amount of (additional) published data, since storage in distributed ledgers is particularly costly.

In summary, the desiderata of our mechanisms are as follows:

- *Tax gap identification and counterincentive*: Tax evasion, i.e., failure of a user $\mathcal{U}$ to declare the amount of assets they own, should be either detectable by a tax authority $\mathcal{T}$, with access to the ledger, or render the assets unusable.
- *High level of privacy*: $\mathcal{T}$ should — at most — learn the total amount of assets owned by each taxpayer at the end of a fiscal year; this information should be leaked only to $\mathcal{T}$ and no additional information should be leaked to any other party, apart from the information already published on the ledger.
- *Unobstructed operation*: The introduction of a taxation mechanism should not result in any period during which the — tax compliant — users are prohibited from transacting.
- *Low performance overhead*: The taxation mechanism should not introduce a major performance overhead, in terms of computation and storage requirements from the users and the taxation authority.
- *Balanced load*: The computation and storage overhead of taxation should be spread over a period of time, rather than introduce performance spikes.

### 8.2.2 Tax Auditable Distributed Ledger

In this section we describe a ledger with a built-in tax auditing mechanism. Our design is generic, such that existing ledgers can incorporate it with minimal changes in the underlying consensus protocol. An *auditable ledger* enforces a user $\mathcal{U}$ to declare the amount of assets they own to a taxation authority $\mathcal{T}$, with failure to do so rendering the assets

unusable. We achieve this while leaking to $\mathcal{T}$ only the total amount of assets that $\mathcal{U}$ owns at a specific point in time, e.g., the end of a fiscal year. We note that we consider only pseudonymous ledgers, so potentially de-anonymizable data may be published on the ledger, e.g., addresses which may be linked to the user who controls them.

We assume that $\mathcal{T}$ holds a list of all taxpayers and is identified by a key $(\mathsf{sk}_{\mathcal{T}}, \mathsf{vk}_{\mathcal{T}})$. Also there exist taxation periods, which last for a pre-specified amount of time $d$. For example, a taxation period may last $1$ calendar year, at the end of which taxpayers need to declare their assets to the authorities.

The core idea is that assets unaccounted for, at the end of the taxation period, are frozen, until their owners declare them to the authority. Specifically, at the end of a taxation period, all assets are frozen. To unfreeze an asset, a taxpayer $\mathcal{U}$ declares it to $\mathcal{T}$ as follows.

First, $\mathcal{U}$ creates a new key pair $(\mathsf{sk}_{\mathcal{U}}, \mathsf{vk}_{\mathcal{U}})$ and the corresponding address $\alpha_{\mathcal{U}}$ and sends $\alpha_{\mathcal{U}}$ to $\mathcal{T}$ as part of a KYC process. Next, $\mathcal{T}$ certifies $\alpha_{\mathcal{U}}$ by issuing the signature $\sigma = \mathsf{Sign}(\alpha_{\mathcal{U}}, \mathsf{sk}_{\mathcal{T}})$, which it gives to $\mathcal{U}$. The tuple $\alpha^t_{\mathcal{U}} = \langle \alpha_{\mathcal{U}}, \sigma \rangle$ is the *certified address*, which is used by the user to transact with frozen assets. $\mathcal{T}$ maintains a mapping of taxpayers and certified addresses, i.e., for every taxpayer $\mathcal{U}$ it holds a list $A_{\mathcal{U}}$ of all certified taxation addresses that $\mathcal{U}$ requested.

A transaction $\tau = \langle \alpha_s, \alpha_d, \Theta \rangle$, which moves $\Theta$ frozen assets from an address $\alpha_s$, is valid only if $\alpha_d = \langle \alpha, \sigma \rangle \wedge \mathsf{Verify}(\alpha, \sigma, \mathsf{vk}_{\mathcal{T}}) = 1$. Consequently, miners accept transactions that unfreeze assets only as long as said assets are transferred to a certified address. Therefore, $\mathcal{T}$ can compute the amount of $\mathcal{U}$'s assets as $\Theta_{\mathcal{U}} := \sum_{i=1}^{n} \mathsf{bal}(\alpha_{\mathcal{U}}[i])$, $n$ being the total number of $\mathcal{U}$'s certified addresses.

We note that the system can accommodate multiple taxation authorities from different countries. In that case, $\mathcal{T}$ is a federation of authorities, each identified by a single key. Each authority's key is published on the ledger and a taxpayer can certify their addresses and declare their assets to the respective authorities.

Naturally, this mechanism introduces some challenges. Although standard pay-to-public-key-hash addresses are $25$ bytes, certified addresses may be significantly larger, due to the certification signature of $\mathcal{T}$. For instance, ECDSA signatures in the DER format result in $72$ additional bytes, thus making certified addresses $99$ bytes long. Nevertheless, certified addresses are expected to be used only once, to declare the assets, thus the overall storage cost should not be significant. Another important consideration regards to the private state of the taxation authority; given the statute of limitations, $\mathcal{T}$ might need to maintain its taxation private key and the mapping of certified addresses

for a significant period, possibly resulting in significant maintenance costs.

We showcase our design via an auditable variation of Bitcoin ledger, denoted as $\mathcal{L}^t$. $\mathcal{L}^t$ is initially parameterized by the public key of the authority $(\mathsf{sk}_{\mathcal{T}}, \mathsf{vk}_{\mathcal{T}})$, which is part of the ledger's genesis block. During the execution, $\mathcal{T}$ can update its key by simply signing a new key $\mathsf{vk}'_{\mathcal{T}}$ with $\mathsf{sk}_{\mathcal{T}}$ and publishing it on the ledger. A taxation period lasts $52560$ blocks, i.e., roughly $1$ calendar year, so block $52560$ and its multiples are "tax-auditing" blocks. When a tax-auditing block is issued, all assets on $\mathcal{L}^t$ which are controlled by non-certified addresses are frozen. To transact with assets from a frozen address, a user sends them to a certified address, as described above.

Freezing complicates the system in a number of ways. First, the liveness of a transaction [GKL15] may be affected. For instance, a transaction which spends from a non-certified address will be rejected, if it is created before but published after a tax-auditing block. We sidestep this issue by enabling users to use certified addresses before the freezing period, hence the liveness guarantees of the ledger apply unconditionally on certified addresses. Second, it is possible that multiple competing tax-auditing blocks are created, e.g., multiple blocks which extend the tax-auditing block. Therefore, $\mathcal{T}$ needs to pick one and certify it. Afterwards, this certified block cannot be reverted and acts as a "checkpoint".

We note that $\mathcal{L}^t$ covers the desiderata proposed in Section 8.2.1. Regarding privacy, although $\mathcal{T}$ can de-anonymize the set of $\mathcal{L}^t$ users at a specific point in time, i.e., when the assets freeze, the users can employ standard Bitcoin addresses and transactions outwith this period. Additionally, as with standard Bitcoin addresses, third parties cannot obtain information regarding the identity of a certified address's owner (as long as the signature itself does not leak it). In terms of performance, a user can transact with their assets effortlessly, as long as they use certified addresses to receive or unfreeze assets around the taxation period. Importantly, users can certify their addresses ahead of the freezing time, thus the additional load can be spread over a period of a few days or weeks.

### 8.2.3 A Tax-Auditing Extension for Provisions

We now build a tax auditing mechanism for existing ledgers, which is based on Provisions [DBB+15]. The goal of this mechanism is to enable all payment recipients to verify whether the assets used by a sender $\mathcal{E}$ in a transaction have been properly declared to the authority $\mathcal{T}$. This is achieved in two stages, first with an asset declaration stage

that involves $\mathcal{T}$ and second with a payer address auditing protocol, which is created in tandem with the transaction that pays a recipient, and after $\mathcal{E}$ commits to owning the assets. If $\mathcal{E}$ fails to provide such proof, the implication is that $\mathcal{E}$ performs tax evasion. To build this proof mechanism we rely on Provisions [DBB+15], particularly its *proof of assets*. Our scheme comprises of two simple protocols, which $\mathcal{E}$ runs with the taxation authority and their counter-party respectively. As we show, our protocols retain the privacy guarantees of Provisions.

Provisions is a privacy-preserving auditing mechanism for Bitcoin exchanges. Using Provisions a party can verify that a (cooperating) Bitcoin exchange is solvent, i.e., possesses enough assets to cover the liabilities towards its users. In order to achieve this, Provisions defines three protocols: i) proof of assets, ii) proof of liabilities, and iii) proof of solvency. The first protocol commits the exchange — in a zero-knowledge fashion — to the total amount of assets it possesses. The second commits it to the liabilities towards its clients, such that each client can verify that the exchange has included his/her deposits in the collective proof. Finally, the proof of solvency proves that the exchange's assets are equal or surpass its liabilities. Our work is only concerned in the assets owned by the exchange, thus we focus on the proof of assets. All proofs are considered under a group $G$ of prime order $q$ with fixed public generators $g, h$. The proof of assets considers the following:

- $\mathsf{PK} = \{y_1, ..., y_n\}$: the total (anonymity) set of public keys;
- $s_i$: a bit such that, if the exchange controls $y_i$, i.e., if it possesses the private key of $y_i$, then $s_i = 1$, otherwise $s_i = 0$;
- $\mathsf{bal}(y_i)$: the amount of assets that the address corresponding to $y_i$ controls;
- $\Theta = \sum_{i=1}^{n} s_i \cdot \mathsf{bal}(y_i)$: the amount of assets that the exchange controls;
- $b_i = g^{\mathsf{bal}(y_i)}$: a biding (but not hiding) commitment to the balance of $y_i$.

The exchange publishes the Pedersen commitments [Ped92] for each $s_i \cdot \mathsf{bal}(y_i), s_i$:

$$p_i = b_i^{s_i} \cdot h^{v_i} = g^{\mathsf{bal}(y_i) \cdot s_i} \cdot h^{v_i} \tag{8.5}$$

$$l_i = y_i^{s_i} h^{t_i} = g^{\hat{x}_i} h^{t_i} \tag{8.6}$$

where $v_i, t_i \in \mathbb{Z}_q$ are chosen at random, $x_i$ is the private key for $y_i$, and $\hat{x}_i = x_i \cdot s_i$. Then, the exchange proves knowledge of values $s_i, v_i, t_i, \hat{x}_i$ for every $i \in [1, n]$ via a $\Sigma$-protocol, such that conditions (8.5), (8.6) are satisfied.

**Asset Declaration.** In our case, $\mathcal{E}$ declares the total amount of assets it controls, i.e., the value $\Theta$, to $\mathcal{T}$ who verifies that $\mathcal{E}$'s commitments correspond to $\Theta$. We obtain the condition $Z_\Theta = \prod_{i=1}^{n} p_i = g^\Theta \cdot h^v$, where $v = \sum_{i=1}^{n} v_i$, is a (publicly computable) Pedersen commitment to $\mathcal{E}$'s assets. Given that $\mathcal{T}$ knows $\Theta$, $\mathcal{E}$ needs only to prove knowledge of a value $v$, such that this condition is satisfied. This is done via the Schnorr protocol [Sch90] of Figure 8.1, which guarantees privacy as described in Lemma 5.

Asset Declaration Protocol $\mathcal{P}_{asset}$

Public data: $g, h, Z_\Theta = \prod_{i=1}^{n} p_i$
Verifier's input from prover: $\Theta$
Prover's input: $v = \sum_{i=1}^{n} v_i$

1. The prover ($\mathcal{E}$) chooses $r \xleftarrow{\$} \mathbb{Z}_q$ and sends $\lambda = h^r$ to the verifier ($\mathcal{T}$).
2. The verifier replies with a challenge $c \xleftarrow{\$} \mathbb{Z}_q$.
3. The prover responds with $\theta = r + c \cdot v$.
4. The verifier accepts if $h^\theta \stackrel{?}{=} \lambda \cdot (Z_\Theta \cdot g^{-\Theta})^c$.

Figure 8.1: Tax-auditing between $\mathcal{E}$ (prover) and $\mathcal{T}$ (verifier).

**Lemma 5.** *For public values $g, h$ and $Z_\Theta$, the protocol $\mathcal{P}_{asset}$ is an honest-verifier zero-knowledge argument of knowledge of quantity $v$ satisfying $Z_\Theta = \prod_{i=1}^{n} p_i = g^\Theta \cdot h^v$ for $i \in [1, n]$.*

**Payer Address Auditing.** The second part of our taxation proof enables the tax auditing of a specific address used by a payer $\mathcal{E}$ whenever a payment is made to an arbitrary user $\mathcal{U}$. $\mathcal{E}$ will prove two conditions to $\mathcal{U}$: i) for some $i \in [1, n]$, the public key $y_i$ (which is published as part of the Provisions scheme) corresponds to the address from which $\mathcal{U}$ receives their assets; ii) the corresponding bit $s_i$ for $y_i$ in the commitment condition (8.6) is $s_i = 1$. The first condition can be easily proven by providing $\mathcal{U}$ with an index $i$, such that $\mathcal{U}$ confirms that the address in question is equal to the hash of $y_i$. To prove the second condition, we observe that, for $s_i = 1$, $p_i = g^{\mathsf{bal}(y_i)} h^{v_i}$ and $l_i = y_i h^{t_i}$. Therefore, $\mathcal{E}$ needs only to prove knowledge of $t_i$ and $v_i$, such that this statement is satisfied, which can be achieved via the Schnorr protocol of Figure 8.2, its privacy properties formalized in Lemma 6.

Address Verification Protocol $\mathcal{P}_{address}$

Public data: $h$, $(y_i, l_i)$, bal$(y_i)$ for $i \in [1, n]$
Verifier's input from prover: $i$
Prover's input: $t_i$

1. The prover ($\mathcal{E}$) chooses $r_1, r_2 \xleftarrow{\$} \mathbb{Z}_q$ and sends $\lambda_1 = h^{r_1}, \lambda_2 = h^{r_2}$ to the verifier.
2. The verifier replies with a challenge $c \xleftarrow{\$} \mathbb{Z}_q$.
3. The prover responds with $\theta_1 = r_1 + c \cdot t_i$, $\theta_2 = r_2 + c \cdot v_i$.
4. The verifier accepts if $h^{\theta_1} \stackrel{?}{=} \lambda_1 \cdot (l_i \cdot y_i^{-1})^c$ and $h^{\theta_2} \stackrel{?}{=} \lambda_2 \cdot (p_i \cdot g^{-\text{bal}(y_i)})^c$.

Figure 8.2: Address verification between $\mathcal{E}$ (prover) and a user $\mathcal{U}$ (verifier).

**Lemma 6.** *For public values $g, h$ and $y_i, l_i, p_i$, bal$(y_i)$, the protocol $\mathcal{P}_{address}$ is an honest-verifier zero-knowledge argument of knowledge of quantities $t_i, v_i$ satisfying $l_i = y_i h^{t_i}$ and $p_i = g^{\text{bal}(y_i)} h^{v_i}$ respectively.*

Finally, both protocols can be turned into non-interactive zero-knowledge (NIZK) proofs of knowledge, in the random oracle model, using the Fiat-Shamir transformation [FS87].

# Chapter 9

# Conclusion

The goal of this thesis was to expand the horizon of how to build secure and efficient applications on top of distributed ledgers. This goal was approached via a series of research works, each building on and expanding the existing literature, from and towards multiple directions. Our research evolved around three axes: i) cryptography; ii) distributed systems; iii) game theory and economics.

On the cryptographic side, we both analyzed pre-existing protocols and proposed constructions of our own. Specifically, in focusing on the security and safety of distributed ledger-based systems, we analyzed hardware wallets (Chapter 3), describing a formal model to capture their necessary properties and evaluate real-world implementations. Following this, we proposed a holistic model of wallets for Proof-of-Stake-based systems (Chapter 4). This line of work brought about a formal definition of a PoS wallet's core, a novel security notion (address malleability), and a rigorous description of incorporating the wallet's core in a PoS distributed ledger system. Notably, our model highlighted a number of disadvantages, perhaps most intesting of which is the tendency for centralization around a few participants. After identifying this drawback, we devised a collective stake pool protocol (Chapter 5), which enables a group of people to form a coalition in a trustless manner and be as competitive as centrally-controlled participating entities. Crucially, across the thesis, our security analyses employed simulation-based proofs of security, striving to offer the highest level of guarantees possible.

On the systems' side, we focused on improving the efficiency and scalability of distributed ledger-based systems. Motivated by the ever-growing (and eventually unsustainable) amount of data published in existing distributed ledgers, we devised a framework for improving transaction state efficiency (Chapter 6). Our framework is inspired by similar techniques in traditional database systems and consists of multiple layers, each

enabling incremental improvements towards minimizing a ledger's state.

On the economics' side, we analyzed distributed ledger systems first from a micro-economic perspective. Chapters 5 and 6 offer high-level descriptions of incentivizing users to behave honestly, when participating in collective stake pools and creating efficient transactions respectively. More importantly, Chapter 7 engages in a thorough analysis of the Nash dynamics of blockchain-based financial systems and introduces the notion of compliance. The results of our analysis highlight core differences between PoW and PoS-based systems, formalized the necessity of penalizing misbehaving parties in the latter, and showcased that the market does not often respond rationally, thus cannot (on its own) offer protection against economic attacks.

Finally, Chapter 8 explored macroeconomic properties of distributed ledgers. Section 8.1 introduced crypto-egalitarianism, a property that expresses the reward rate of participants, with respect to their capital investment in the system. We offered a formal definition of crypto-egalitarianism, which boils it down to a single number, enabling a precise comparison, in contrast to existing ad hoc arguments. Crucially, our research proved that wealth redistribution in favor of the poor is impossible in completely decentralized systems; consequently, some level of central control is needed to enforce any such policy in a democratically-mandated manner. Building on this necessity, Section 8.2 explored how to reduce tax gaps, thus helping a government to efficiently enforce its intended tax policies via a distributed ledger-based monetary and payment system.

## Future Work

Throughout this thesis, multiple research questions arose, paving the way for interesting future directions. The model of Chapter 3 highlighted the need for the operator of a hardware wallet to faithfully follow the prescribed protocol; future work could evaluate the error probability, identify the causes of mistakes, and propose techniques to reduce such risk. Building on the results of Chapter 4, future work could explore efficient (i.e., short) non-malleable address schemes, as well as explore the implications of incorporating higher levels of anonymity and privacy. The collective stake pool of Chapter 5 currently requires closing and re-creating a pool, in order to update its parameters and add new members; a more efficient design could expand this scheme to enable such changes in a dynamic manner. The analysis of Chapter 6 is focused on UTxO-based ledgers and size (as a cost function); future research could propose an adapted state efficiency framework, which incorporates account-based ledgers and ex-

plores its behavior under varying cost models. In Chapter 7, we consider only two infraction predicates and rewards that stem only from the system itself; future work could explore alternative infraction predicates, e.g., to take into account selfish mining, how reward transfers between parties may affect the compliance analysis, as well as possible correlations between the exchange rate and protocol rewards, to produce an analysis for settings closer to the real-world. The crypto-egalitarianism property of Section 8.1 is defined over the, rather simplistic, economic model of Bitcoin (and its disciples); further research is needed, both to evaluate various existing systems and identify whether economies of scale in various parameters exacerbate the identified gap between PoW and PoS. Finally, Section 8.2 offered a glimpse of how ledgers can solve real-world problems that tax authorities face; further research could enable collaboration between tax authorities of different countries, incorporate tax gap identification mechanisms in anonymous ledgers, and offer incentives to motivate adoption, instead of depending on law enforcement.

## Epilogue

During the $4$-odd years of working on this thesis, a chasm grew in me. On the one hand, I devoted large amounts of time and energy in understanding, building, and fixing decentralized ledgers. On the other hand, my initial interest in Bitcoin and cryptocurrencies gradually turned to disillusionment and eventual distaste. Akin to an atheist monk, I kept working day in day out on beautiful designs, which were used by ideologies that I detested. I believe that this thesis manifests this internal conflict, with technical contributions existing side by side with snarky comments.

Two questions emerged from this conflict. Are decentralized ledgers merely tools, possibly usable in societally beneficial applications, instead of speculative financial products of late capitalism? Should intellectual work always serve a societal purpose, or is science for science's sake good enough? Neither question is new. The former is expressed with the classic parable, that a knife can be used both to slice bread and as a murder instrument. The latter came to the spotlight during the $19$th and $20$th centuries in science, due to nuclear power, and in art, with the emergence of aestheticism.

I don't have an answer to either question. I don't know if a definitive answer *can* exist. Nonetheless, I do know that acknowledging these questions may lead to the same consequences as acknowledging Camus's absurd: revolt, freedom, and passion.[1]

[1] *The Myth of Sisyphus*, Albert Camus

# Bibliography


[AAC+17] Hamza Abusalah, Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, and Leonid Reyzin. Beyond hellman's time-memory trade-offs with applications to proofs of space. In Takagi and Peyrin [TP17], pages 357–379. doi:10.1007/978-3-319-70697-9_13.

[AAMMZ12] Julian Assange, Jacob Appelbaum, Andy Müller-Maguhn, and Jérémie Zimmermann. *Cypherpunks: Freedom and the Future of the Internet*. OR books New York, 2012.

[ABLZ18] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of bitcoin transactions. In Meiklejohn and Sako [MS18], pages 541–560. doi:10.1007/978-3-662-58387-6_29.

[ABP17] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Depth-robust graphs and their cumulative memory complexity. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 3–32, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany. doi:10.1007/978-3-319-56617-7_1.

[ADN06] Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold RSA with adaptive and proactive security. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 593–611, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany. doi:10.1007/11761679_35.

[Adv20] Arun Advani. Who does and doesn't pay taxes? *Fiscal Studies*, 2020.

[AGKK19] Myrto Arapinis, Andriana Gkaniatsou, Dimitris Karakostas, and Aggelos Kiayias. A formal treatment of hardware wallets. In Goldberg and Moore [GM19], pages 426–445. doi:10.1007/978-3-030-32101-7_26.

[AGM18] Shashank Agrawal, Chaya Ganesh, and Payman Mohassel. Non-interactive zero-knowledge proofs for composite statements. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 643–673, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany. doi:10.1007/978-3-319-96878-0_22.

[AHA17] Michel Avital, Jonas Hedman, and Lars Albinsson. Smart money: Blockchain-based customizable payments system. *Dagstuhl Reports*, 7(3):104–106, 2017.

[Alo17] JD Alois. Ethereum parity hack may impact eth 500.000 or 146 million. https://www.crowdfundinsider.com/2017/11/124200-ethereum-parity-hack-may-impact-eth-500000-146-million/, 2017. [Online; accessed 1-Sep-2018].

[And15] Gavin Andresen. Utxo uh-oh..., 2015. http://gavinandresen.ninja/utxo-uhoh.

[AW19] Nick Arnosti and S. Matthew Weinberg. Bitcoin: A natural oligopoly. In Avrim Blum, editor, *ITCS 2019: 10th Innovations in Theoretical Computer Science Conference*, volume 124, pages 5:1–5:1, San Diego, CA, USA, January 10–12, 2019. LIPIcs. doi:10.4230/LIPIcs.ITCS.2019.5.

[B+14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[Bam14] Chris Bambery. *A People's History of Scotland*. Verso Trade, 2014.

[Ban21] European Central Bank. Eurosystem report on the public consultation on a digital euro, 2021. URL: https://www.ecb.europa.eu/pub/pdf/other/Eurosystem_report_on_the_public_consultation_on_a_digital_euro~539fa8cd8d.en.pdf.

[Bau04] William J Baumol. Welfare economics and the theory of the state. In *The encyclopedia of public choice*, pages 937–940. Springer, 2004.

[BBF18] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. Cryptology ePrint Archive, Report 2018/1188, 2018. `https://eprint.iacr.org/2018/1188`.

[BBMS14] Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors. *FC 2014 Workshops*, volume 8438 of *Lecture Notes in Computer Science*, Christ Church, Barbados, March 7, 2014. Springer, Heidelberg, Germany.

[BBO07] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 535–552, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany. `doi:10.1007/978-3-540-74143-5_30`.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press. `doi:10.1109/SP.2014.36`.

[BCJR15] Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors. *FC 2015 Workshops*, volume 8976 of *Lecture Notes in Computer Science*, San Juan, Puerto Rico, January 30, 2015. Springer, Heidelberg, Germany.

[BDWW14] Tobias Bamert, Christian Decker, Roger Wattenhofer, and Samuel Welten. Bluewallet: The secure bitcoin wallet. In *International Workshop on Security and Trust Management*, pages 65–80. Springer, 2014.

[Ben70] Jeremy Bentham. An introduction to the principles of morals and legislation (1789), ed. by j. *H Burns and HLA Hart, London*, 1970.

[BG17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 913–930, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi:10.1145/3243734.3243848.

[BGM+18] Christian Badertscher, Juan A. Garay, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. But why does it work? A rational protocol design treatment of bitcoin. In Nielsen and Rijmen [NR18], pages 34–65. doi:10.1007/978-3-319-78375-8_2.

[BHK+20] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. Combining ghost and casper, 2020. arXiv:2003.03052.

[Bit15] Bitcoin. July 2015 flood attack, 2015. https://en.bitcoin.it/wiki/July_2015_flood_attack.

[Bit20a] Bitcoin. Miner fees, 2020. https://en.bitcoin.it/wiki/Miner_fees.

[Bit20b] Bitcoin. Protocol documentation, 2020. https://en.bitcoin.it/wiki/Protocol_documentation.

[Bit21a] Bitinfocharts. Bitcoin avg. transaction fee historical chart, July 2021. https://bitinfocharts.com/comparison/bitcoin-transactionfees.html.

[Bit21b] Bitinfocharts. Bitcoin rich list, July 2021. https://bitinfocharts.com/top-100-richest-bitcoin-addresses.html.

[BK16] Alex Biryukov and Dmitry Khovratovich. Egalitarian computing. In Holz and Savage [HS16], pages 315–326.

[BKKS18] Lars Brünjes, Aggelos Kiayias, Elias Koutsoupias, and Aikaterini-Panagiota Stouka. Reward sharing schemes for stake pools. *CoRR*, abs/1807.11218, 2018. URL: http://arxiv.org/abs/1807.11218, arXiv:1807.11218.

[BKKS20] Lars Brünjes, Aggelos Kiayias, Elias Koutsoupias, and Aikaterini-Panagiota Stouka. Reward sharing schemes for stake pools. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*, pages 256–275. IEEE, 2020. doi:10.1109/EuroSP48549.2020.00024.

[BM99] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany. doi:10.1007/3-540-48405-1_28.

[BMC+15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press. doi:10.1109/SP.2015.14.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Katz and Shacham [KS17], pages 324–356. doi:10.1007/978-3-319-63688-7_11.

[BO15] Rainer Böhme and Tatsuaki Okamoto, editors. *FC 2015: 19th International Conference on Financial Cryptography and Data Security*, volume 8975 of *Lecture Notes in Computer Science*, San Juan, Puerto Rico, January 26–30, 2015. Springer, Heidelberg, Germany.

[BRLP19] Vitalik Buterin, Daniël Reijsbergen, Stefanos Leonardos, and Georgios Piliouras. Incentives in ethereum's hybrid casper protocol. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019*, pages 236–244. IEEE, 2019. doi:10.1109/BLOC.2019.8751241.

[BS11] Codruta Boar and Róbert Szemere. Payments go (even more) digital*, 2011. URL: https://www.bis.org/statistics/payment_stats/commentary2011.htm.

[But14] Vitalik Buterin. On stake, 2014. https://blog.ethereum.org/2014/07/05/stake/.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press. doi:10.1109/SFCS.2001.959888.

[Can03] Ran Canetti. Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239, 2003. https://eprint.iacr.org/2003/239.

[Car] Cardano. Cardano. https://cardano.org/.

[CDK02] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems - concepts and designs (3. ed.)*. International computer science series. Addison-Wesley-Longman, 2002.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85, Amsterdam, The Netherlands, February 21–24, 2007. Springer, Heidelberg, Germany. doi:10.1007/978-3-540-70936-7_4.

[CEV14] Nicolas T. Courtois, Pinar Emirdag, and Filippo Valsorda. Private key recovery combination attacks: On extreme fragility of popular bitcoin key management, wallet and cold storage solutions in presence of poor RNG events. Cryptology ePrint Archive, Report 2014/848, 2014. https://eprint.iacr.org/2014/848.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with

identifiable aborts. In Ligatti et al. [LOKV20], pages 1769–1787. doi:10.1145/3372297.3423367.

[CGL+17] Alessandro Chiesa, Matthew Green, Jingcheng Liu, Peihan Miao, Ian Miers, and Pratyush Mishra. Decentralized anonymous micropayments. In Coron and Nielsen [CN17], pages 609–642. doi:10.1007/978-3-319-56614-6_21.

[CGMV18] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. ALGORAND AGREEMENT: Super fast and partition resilient byzantine agreement. Cryptology ePrint Archive, Report 2018/377, 2018. https://eprint.iacr.org/2018/377.

[CJKR12] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In Darek Kowalski and Alessandro Panconesi, editors, *31st ACM Symposium Annual on Principles of Distributed Computing*, pages 301–308, Funchal, Madeira, Portugal, July 16–18, 2012. Association for Computing Machinery. doi:10.1145/2332432.2332490.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. Cryptology ePrint Archive, Report 2002/059, 2002. https://eprint.iacr.org/2002/059.

[CKM19] Alexander Chepurnoy, Vasily Kharin, and Dmitry Meshkov. A systematic approach to cryptocurrency fees. In Zohar et al. [ZET+19], pages 19–30. doi:10.1007/978-3-662-58820-8_2.

[CKWN16] Miles Carlsten, Harry A. Kalodner, S. Matthew Weinberg, and Arvind Narayanan. On the instability of bitcoin without the block reward. In Weippl et al. [WKK+16], pages 154–167. doi:10.1145/2976749.2978408.

[CN17] Jean-Sébastien Coron and Jesper Buus Nielsen, editors. *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[Com18] EOS Community. Eos.io technical white paper v2, 2018. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.

[CP18] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In Nielsen and Rijmen [NR18], pages 451–467. doi:10.1007/978-3-319-78375-8_15.

[CPR19] Xi Chen, Christos H. Papadimitriou, and Tim Roughgarden. An axiomatic approach to block rewards. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*, pages 124–131. ACM, 2019. doi:10.1145/3318041.3355470.

[CPZ18] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968, 2018. https://eprint.iacr.org/2018/968.

[CS11] Steve Chien and Alistair Sinclair. Convergence to approximate nash equilibria in congestion games. *Games Econ. Behav.*, 71(2):315–327, 2011. doi:10.1016/j.geb.2009.05.004.

[CS14] Nicolas Christin and Reihaneh Safavi-Naini, editors. *FC 2014: 18th International Conference on Financial Cryptography and Data Security*, volume 8437 of *Lecture Notes in Computer Science*, Christ Church, Barbados, March 3–7, 2014. Springer, Heidelberg, Germany.

[Dam88] Ivan Damgård. Collision free hash functions and public key signature schemes. In David Chaum and Wyn L. Price, editors, *Advances in Cryptology – EUROCRYPT'87*, volume 304 of *Lecture Notes in Computer Science*, pages 203–216, Amsterdam, The Netherlands, April 13–15, 1988. Springer, Heidelberg, Germany. doi:10.1007/3-540-39118-5_19.

[DBB+15] Gaby G. Dagher, Benedikt Bünz, Joseph Bonneau, Jeremy Clark, and Dan Boneh. Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges. In Ray et al. [RLK15], pages 720–731. doi:10.1145/2810103.2813674.

[DC18] Edsko de Vries Duncan Coutts. Formal specification for a cardano wallet - an iohk technical report, 2018. `https://cardanodocs.com/files/formal-specification-of-the-cardano-wallet.pdf`.

[DCKT19] S. Dos Santos, C. Chukwuocha, S. Kamali, and R. K. Thulasiram. An efficient miner strategy for selecting cryptocurrency transactions. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 116–123, 2019.

[DDL18] Bernardo David, Rafael Dowsley, and Mario Larangeira. Kaleidoscope: An efficient poker protocol with payment distribution and penalty enforcement. In Meiklejohn and Sako [MS18], pages 500–519. `doi:10.1007/978-3-662-58387-6_27`.

[DDL19] Bernardo David, Rafael Dowsley, and Mario Larangeira. ROYALE: A framework for universally composable card games with financial rewards and penalties enforcement. In Goldberg and Moore [GM19], pages 282–300. `doi:10.1007/978-3-030-32101-7_18`.

[DDN03] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM review*, 45(4):727–784, 2003.

[dec19] decred.org. Decred—an autonomous digital currency, 2019. `https://decred.org`.

[DFKP15] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 585–605, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany. `doi:10.1007/978-3-662-48000-7_29`.

[DFL19] Poulami Das, Sebastian Faust, and Julian Loss. A formal treatment of deterministic wallets. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 651–668. ACM Press, November 11–15, 2019. `doi:10.1145/3319535.3354236`.

[DGKR18] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Nielsen and Rijmen [NR18], pages 66–98. doi:10.1007/978-3-319-78375-8_3.

[Dig21] Digiconomist. Bitcoin energy consumption index, July 2021. https://digiconomist.net/bitcoin-energy-consumption.

[DKT+20] Amir Dembo, Sreeram Kannan, Ertem Nusret Tas, David Tse, Pramod Viswanath, Xuechao Wang, and Ofer Zeitouni. Everything is a race and nakamoto always wins. In Ligatti et al. [LOKV20], pages 859–878. doi:10.1145/3372297.3417290.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO'92*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147, Santa Barbara, CA, USA, August 16–20, 1993. Springer, Heidelberg, Germany. doi:10.1007/3-540-48071-4_10.

[doc18] docdroid. Ledger receive address attack. https://www.docdroid.net/Jug5LX3/ledger-receive-address-attack.pdf, 2018. [Online; accessed 19-Sep-2018].

[Dou02] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[DPNH17] Sergi Delgado-Segura, Cristina Pérez-Solà, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. Analysis of the bitcoin UTXO set. Cryptology ePrint Archive, Report 2017/1095, 2017. https://eprint.iacr.org/2017/1095.

[DPNH19] Sergi Delgado-Segura, Cristina Pérez-Solà, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. Analysis of the bitcoin UTXO set. In Zohar et al. [ZET+19], pages 78–91. doi:10.1007/978-3-662-58820-8_6.

[DPS19] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake.

In Goldberg and Moore [GM19], pages 23–41. doi:10.1007/978-3-030-32101-7_2.

[Dry19] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the bitcoin UTXO set. Cryptology ePrint Archive, Report 2019/611, 2019. https://eprint.iacr.org/2019/611.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[DSE] Jack Doerner, Abhi Shelat, and David Evans. Zeroledge: Proving solvency with privacy.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[EGSVR16] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*, pages 45–59, 2016.

[EOB19] David Easley, Maureen O'Hara, and Soumya Basu. From mining to markets: The evolution of bitcoin transaction fees. *Journal of Financial Economics*, 134(1):91–109, 2019.

[ES14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In Christin and Safavi-Naini [CS14], pages 436–454. doi:10.1007/978-3-662-45472-5_28.

[Eth18a] Ethereum. Glossary: Account nonce, 2018. https://github.com/ethereum/wiki/wiki/Glossary.

[Eth18b] Ethereum. Proof of stake faqs, 2018. https://eth.wiki/en/concepts/proof-of-stake-faqs.

[FKKP19] Amos Fiat, Anna Karlin, Elias Koutsoupias, and Christos H. Papadimitriou. Energy equilibria in proof-of-work mining. In Anna Karlin, Nicole Immorlica, and Ramesh Johari, editors, *Proceedings of the 2019 ACM Conference on Economics and Computation, EC 2019, Phoenix, AZ, USA, June*

*24-28, 2019*, pages 489–502. ACM, 2019. doi:10.1145/3328526.3329630.

[FKO+19] Giulia C. Fanti, Leonid Kogan, Sewoong Oh, Kathleen Ruan, Pramod Viswanath, and Gerui Wang. Compounding of wealth in proof-of-stake cryptocurrencies. In Goldberg and Moore [GM19], pages 42–61. doi:10.1007/978-3-030-32101-7_3.

[FMJR20] Mehdi Fooladgar, Mohammad Hossein Manshaei, Murtuza Jadliwala, and Mohammad Ashiqur Rahman. On incentive compatible role-based reward distribution in algorand. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 452–463. IEEE, 2020. doi:10.1109/DSN48063.2020.00059.

[Fou20] Algorand Foundation. Faqs, 2020. URL: https://algorand.foundation/faq.

[FPT04] Alex Fabrikant, Christos H. Papadimitriou, and Kunal Talwar. The complexity of pure nash equilibria. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 604–612. ACM, 2004. doi:10.1145/1007352.1007445.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany. doi:10.1007/3-540-47721-7_12.

[Fus19] Russ Fustino. Algorand atomic transfers, 2019. https://medium.com/algorand/algorand-atomic-transfers-a405376aad44.

[FvW19] Ellie Frost and Aaron van Wirdum. Bitcoin's growing utxo problem and how utreexo can help solve it, 2019. https://bitcoinmagazine.com/articles/bitcoins-growing-utxo-problem-and-how-utreexo-can-help-solve-it.

[GAK17] Andriana Gkaniatsou, Myrto Arapinis, and Aggelos Kiayias. Low-level attacks in bitcoin wallets. In Phong Q. Nguyen and Jianying Zhou, editors, *ISC 2017: 20th International Conference on Information Security*, volume 10599 of *Lecture Notes in Computer Science*, pages 233–253, Ho Chi Minh City, Vietnam, November 22–24, 2017. Springer, Heidelberg, Germany.

[GBWM+04] Vipul Gupta, Simon Blake-Wilson, Bodo Moeller, Chris Hawk, and N Bolyard. Ecc cipher suites for tls, 2004.

[Ger17] David Gerard. *Attack of the 50 foot blockchain: Bitcoin, blockchain, Ethereum & smart contracts*. David Gerard, 2017.

[GG20] Rosario Gennaro and Steven Goldfeder. One round threshold ECDSA with identifiable abort. Cryptology ePrint Archive, Report 2020/540, 2020. `https://eprint.iacr.org/2020/540`.

[GGM16] Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. In Grossklags and Preneel [GP16], pages 81–98.

[GHM+17a] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017. `doi:10.1145/3132747.3132757`.

[GHM+17b] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454, 2017. `https://eprint.iacr.org/2017/454`.

[GK20a] Clemente Galdi and Vladimir Kolesnikov, editors. *SCN 20: 12th International Conference on Security in Communication Networks*, volume 12238 of *Lecture Notes in Computer Science*, Amalfi, Italy, September 14–16, 2020. Springer, Heidelberg, Germany.

[GK20b] Juan A. Garay and Aggelos Kiayias. SoK: A consensus taxonomy in the blockchain era. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, volume 12006 of *Lecture Notes in Computer Science*, pages

284–318, San Francisco, CA, USA, February 24–28, 2020. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-40186-3_13.

[GKKZ11] Juan A. Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. Adaptively secure broadcast, revisited. In Cyril Gavoille and Pierre Fraigniaud, editors, *30th ACM Symposium Annual on Principles of Distributed Computing*, pages 179–186, San Jose, CA, USA, June 6–8, 2011. Association for Computing Machinery. doi:10.1145/1993806.1993832.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-46803-6_10.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Katz and Shacham [KS17], pages 291–323. doi:10.1007/978-3-319-63688-7_10.

[GKR20] Peter Gazi, Aggelos Kiayias, and Alexander Russell. Tight consistency bounds for bitcoin. In Ligatti et al. [LOKV20], pages 819–838. doi:10.1145/3372297.3423365.

[GKW+16] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In Weippl et al. [WKK+16], pages 3–16. doi:10.1145/2976749.2978341.

[GM19] Ian Goldberg and Tyler Moore, editors. *FC 2019: 23rd International Conference on Financial Cryptography and Data Security*, volume 11598 of *Lecture Notes in Computer Science*, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019. Springer, Heidelberg, Germany.

[GMR84] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A "paradoxical" solution to the signature problem (abstract) (impromptu talk). In G. R. Blakley and David Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, page 467, Santa Barbara, CA, USA, August 19–23, 1984. Springer, Heidelberg, Germany.

[GMS17] Miraje Gentilal, Paulo Martins, and Leonel Sousa. Trustzone-backed bitcoin wallet. In *Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems*, pages 25–28. ACM, 2017.

[Gol07] Oded Goldreich. *Foundations of cryptography: volume 1, basic tools*. Cambridge university press, 2007.

[Gol09] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[Gol16] David Golumbia. *The politics of Bitcoin: software as right-wing extremism*. U of Minnesota Press, 2016.

[Goo14] LM Goodman. Tezos—a self-amending crypto-ledger white paper, 2014.

[GOT19] Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. Proof-of-stake protocols for privacy-aware blockchains. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 690–719, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-17653-2_23.

[GP16] Jens Grossklags and Bart Preneel, editors. *FC 2016: 20th International Conference on Financial Cryptography and Data Security*, volume 9603 of *Lecture Notes in Computer Science*, Christ Church, Barbados, February 22–26, 2016. Springer, Heidelberg, Germany.

[Gre12] Andy Greenberg. *This Machine Kills Secrets: How WikiLeakers, Hacktivists, and Cypherpunks Are Freeing the World's Information*. Random House, 2012.

[Gro18] FISCALIS Tax Gap Project Group. the concept of tax gaps. report ii: Corporate income tax gap estimation methodologies, 2018. URL: https://op.europa.eu/en/publication-detail/-/publication/a5da4716-e7c1-11e8-b690-01aa75ed71a1.

[GS15] Gus Gutoski and Douglas Stebila. Hierarchical deterministic bitcoin wallets that tolerate key leakage. In Böhme and Okamoto [BO15], pages 497–504. doi:10.1007/978-3-662-47854-7_31.

[GS20] John M Griffin and Amin Shams. Is bitcoin really untethered? *The Journal of Finance*, 75(4):1913–1964, 2020.

[HDM+14] Danny Yuxing Huang, Hitesh Dharmdasani, Sarah Meiklejohn, Vacha Dave, Chris Grier, Damon McCoy, Stefan Savage, Nicholas Weaver, Alex C. Snoeren, and Kirill Levchenko. Botcoin: Monetizing stolen cycles. In *ISOC Network and Distributed System Security Symposium – NDSS 2014*, San Diego, CA, USA, February 23–26, 2014. The Internet Society.

[HKZG15] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 129–144, Washington, DC, USA, August 12–14, 2015. USENIX Association.

[HLS+09] Hsu-Chun Hsiao, Yue-Hsun Lin, Ahren Studer, Cassandra Studer, King-Hang Wang, Hiroaki Kikuchi, Adrian Perrig, Hung-Min Sun, and Bo-Yin Yang. A study of user-friendly hash comparison schemes. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 105–114. IEEE, 2009.

[HS16] Thorsten Holz and Stefan Savage, editors. *USENIX Security 2016: 25th USENIX Security Symposium*, Austin, TX, USA, August 10–12, 2016. USENIX Association.

[HZ10] Martin Hirt and Vassilis Zikas. Adaptively secure broadcast. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 466–485, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany. doi:10.1007/978-3-642-13190-5_24.

[IH19] Griffin Ichiba Hotchkiss. The 1.x files: The state of stateless ethereum, 2019. https://blog.ethereum.org/2019/12/30/eth1x-files-state-of-stateless-ethereum.

[Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996. doi:10.1145/234313.234367.

[JJ99] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure information networks*, pages 258–272. Springer, 1999.

[JLG+14] Benjamin Johnson, Aron Laszka, Jens Grossklags, Marie Vasek, and Tyler Moore. Game-theoretic analysis of DDoS attacks against bitcoin mining pools. In Böhme et al. [BBMS14], pages 72–86. doi:10.1007/978-3-662-44774-1_6.

[JLG+19] Shuhao Jiang, Jiajun Li, Shijun Gong, Junchao Yan, Guihai Yan, Yi Sun, and Xiaowei Li. Bzip: A compact data memory system for utxo-based blockchains. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8. IEEE, 2019.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[KDF13] Joshua A. Kroll, Ian C. Davey, and Edward W. Felten. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *The Twelfth Workshop on the Economics of Information Security (WEIS 2013)*, 2013.

[Kee18] Keepkey. Keepkey. https://keepkey.com/, 2018. [Online; accessed 1-Sep-2018].

[Kha21] Yogita Khatri. Seventy-five eth2 validators got slashed this week due to a bug witnessed by staked, 2021. https://www.theblockcrypto.com/post/93730/eth2-validators-slashed-staked-bug.

[KJG+16] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In Holz and Savage [HS16], pages 279–296.

[KK20] Dimitris Karakostas and Aggelos Kiayias. Securing proof-of-work ledgers via checkpointing. Cryptology ePrint Archive, Report 2020/173, 2020. https://ia.cr/2020/173.

[KK21] Dimitris Karakostas and Aggelos Kiayias. Filling the tax gap via programmable money, 2021. arXiv:2107.12069.

[KKK21a] Dimitris Karakostas, Nikos Karayannidis, and Aggelos Kiayias. Efficient state management in distributed ledgers. *IACR Cryptol. ePrint Arch.*, 2021:183, 2021. URL: `https://eprint.iacr.org/2021/183`.

[KKK21b] T. Kerber, A. Kiayias, and M. Kohlweiss. Kachina - foundations of private smart contracts. In *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 47–62, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/CSF51468.2021.00002`, `doi:10.1109/CSF51468.2021.00002`.

[KKKT16a] Aggelos Kiayias, Elias Koutsoupias, Maria Kyropoulou, and Yiannis Tselekounis. Blockchain mining games. In *Proceedings of the 2016 ACM Conference on Economics and Computation*, pages 365–382. ACM, 2016.

[KKKT16b] Aggelos Kiayias, Elias Koutsoupias, Maria Kyropoulou, and Yiannis Tselekounis. Blockchain mining games. In Vincent Conitzer, Dirk Bergemann, and Yiling Chen, editors, *Proceedings of the 2016 ACM Conference on Economics and Computation, EC '16, Maastricht, The Netherlands, July 24-28, 2016*, pages 365–382. ACM, 2016. `doi:10.1145/2940716.2940773`.

[KKKZ19] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *2019 IEEE Symposium on Security and Privacy*, pages 157–174, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press. `doi:10.1109/SP.2019.00063`.

[KKL20] Dimitris Karakostas, Aggelos Kiayias, and Mario Larangeira. Account management in proof of stake ledgers. In Galdi and Kolesnikov [GK20a], pages 3–23. `doi:10.1007/978-3-030-57990-6_1`.

[KKL21] Dimitris Karakostas, Aggelos Kiayias, and Mario Larangeira. Conclave: A collective stake pool protocol. *IACR Cryptol. ePrint Arch.*, 2021:742, 2021. URL: `https://eprint.iacr.org/2021/742`.

[KKNZ19] Dimitris Karakostas, Aggelos Kiayias, Christos Nasikas, and Dionysis Zindros. Cryptocurrency egalitarianism: A quantitative approach. In Vincent Danos, Maurice Herlihy, Maria Potop-Butucaru, Julien Prat, and Sara Tucci Piergiovanni, editors, *International Conference on Blockchain*

*Economics, Security and Protocols, Tokenomics 2019, May 6-7, 2019, Paris, France*, volume 71 of *OASIcs*, pages 7:1–7:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/OASIcs.Tokenomics.2019.7.

[KKZ22] Dimitris Karakostas, Aggelos Kiayias, and Thomas Zacharias. Blockchain nash dynamics and the pursuit of compliance, 2022. arXiv:2201.00858.

[KL20] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[Kle20] Naomi Klein. *On fire: the (burning) case for a green new deal*. Simon & Schuster, 2020.

[KLOS19] Elias Koutsoupias, Philip Lazos, Foluso Ogunlana, and Paolo Serafino. Blockchain mining games with pay forward. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 917–927. ACM, 2019. doi:10.1145/3308558.3313740.

[KMB15] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In Ray et al. [RLK15], pages 195–206. doi:10.1145/2810103.2813712.

[KN12] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19:1, 2012.

[KP15] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. Cryptology ePrint Archive, Report 2015/1019, 2015. https://eprint.iacr.org/2015/1019.

[KR18] Aggelos Kiayias and Alexander Russell. Ouroboros-BFT: A simple byzantine fault tolerant consensus protocol. Cryptology ePrint Archive, Report 2018/1049, 2018. https://eprint.iacr.org/2018/1049.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain

protocol. In Katz and Shacham [KS17], pages 357–388. doi:10.1007/978-3-319-63688-7_12.

[KS17] Jonathan Katz and Hovav Shacham, editors. *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[KS19] Aggelos Kiayias and Aikaterini-Panagiota Stouka. Coalition-safe equilibria with virtual payoffs, 2019. arXiv:2001.00047.

[L.18] Kenny L. You don't need a diversified crypto portfolio to spread risk: Here's why, 2018. https://towardsdatascience.com/bitcoin-dominance-5a95f0f3319e.

[LABK17] Wenting Li, Sébastien Andreina, Jens-Matthias Bohli, and Ghassan Karame. Securing proof-of-stake blockchain protocols. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 297–315. Springer, 2017.

[LBS+15] Yoad Lewenberg, Yoram Bachrach, Yonatan Sompolinsky, Aviv Zohar, and Jeffrey S. Rosenschein. Bitcoin mining pools: A cooperative game theoretic analysis. In Gerhard Weiss, Pinar Yolum, Rafael H. Bordini, and Edith Elkind, editors, *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, May 4-8, 2015*, pages 919–927. ACM, 2015. URL: http://dl.acm.org/citation.cfm?id=2773270.

[Led18] Ledger. Ledger user manual. https://support.ledgerwallet.com/hc/en-us/articles/360009676633, 2018. [Online; accessed 1-Sep-2018].

[Lee11] Charles Lee. Litecoin, 2011.

[Lev01] Steven Levy. *Crypto: How the code rebels beat the government–saving privacy in the digital age*. Penguin, 2001.

[LJG15] Aron Laszka, Benjamin Johnson, and Jens Grossklags. When bitcoin mining pools run dry - A game-theoretic analysis of the long-term impact of

attacks between mining pools. In Brenner et al. [BCJR15], pages 63–77. doi:10.1007/978-3-662-48051-9_5.

[LKL+14] Il-Kwon Lim, Young-Hyuk Kim, Jae-Gwang Lee, Jae-Pil Lee, Hyun Nam-Gung, and Jae-Kwang Lee. The analysis and countermeasures on security breach of bitcoin. In *International Conference on Computational Science and Its Applications*, pages 720–732. Springer, 2014.

[Llo33] William Forster Lloyd. *Two lectures on the checks to population*. JH Parker, 1833.

[LOKV20] Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors. *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Virtual Event, USA, November 9–13, 2020. ACM Press.

[Lov19] James Lovejoy. Litecoin cash (lcc) was 51% attacked, 2019. https://gist.github.com/metalicjames/82a49f8afa87334f929881e55ad4ffd7.

[Lov20] James Lovejoy. Bitcoin gold (btg) was 51% attacked, 2020. https://gist.github.com/metalicjames/71321570a105940529e709651d0a9765.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[LTKS15] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In Ray et al. [RLK15], pages 706–719. doi:10.1145/2810103.2813659.

[LVTS17] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. SmartPool: Practical decentralized pooled mining. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017: 26th USENIX Security Symposium*, pages 1409–1426, Vancouver, BC, Canada, August 16–18, 2017. USENIX Association.

[M+14] Gregory Maxwell et al. Deterministic wallets, 2014.

[Mac19] Fitzroy Maclean. *Scotland: a concise history*. Thames & Hudson, 2019.

[Man11] Robert Manne. The cypherpunk revolutionary. *MONTHLY*, 65(March 2011):16–35, 2011.

[Mar] Julian Martinez. Understanding proof of stake: The nothing at stake theory. https://medium.com/coinmonks/understanding-proof-of-stake-the-nothing-at-stake-theory-1f0d71bc027.

[Max13a] Maxminer. Feathercoin's 51% attack - double spending case study, 2013. https://maxminer.files.wordpress.com/2013/06/ftc_51attack.pdf.

[Max13b] Greg Maxwell. Coinjoin: Bitcoin privacy for the real world, 2013. https://bitcointalk.org/index.php?topic=279249.msg2983902#msg2983902.

[Max17] Greg Maxwell. A deep dive into bitcoin core v0.15, 2017. http://diyhpl.us/wiki/transcripts/gmaxwell-2017-08-28-deep-dive-bitcoin-core-v0.15/.

[MB15] Malte Möser and Rainer Böhme. Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In Brenner et al. [BCJR15], pages 19–33. doi:10.1007/978-3-662-48051-9_2.

[McC20] John C. McCallum. Historical memory prices 1957+, 2020. https://en.bitcoin.it/wiki/Miner_fee://jcmit.net/memoryprice.htm.

[McM13] Robert McMillan. Ex-googler gives the world a better bitcoin. *WIRED*, Aug 2013. URL: https://www.wired.com/2013/08/litecoin/.

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Heidelberg, Germany. doi:10.1007/3-540-48184-2_32.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press. doi:10.1109/SP.2013.34.

[MGS18] R Murphy and A Guter-Sandu. Resources allocated to tackling the tax gap: a comparative eu study. *Working paper for Combating Financial Fraud and Empowering Regulators (COFFERS) Horizon 2020 project*, November(A), 2018.

[MN21] Katie Martin and Billy Nauman. Bitcoin's growing energy problem: 'it's a dirty currency', 2021. `https://www.ft.com/content/1aecb2db-8f61-427c-a413-3b929291c8ac`.

[MO19] Tal Moran and Ilan Orlov. Simple proofs of space-time and rational proofs of storage. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 381–409, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany. `doi:10.1007/978-3-030-26948-7_14`.

[MPSV99] Paz Morillo, Carles Padró, Germán Sáez, and Jorge Luis Villar. Weighted threshold secret sharing schemes. *Information processing letters*, 70(5):211–216, 1999.

[MR21] Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. Cryptology ePrint Archive, Report 2021/671, 2021. `https://eprint.iacr.org/2021/671`.

[MS18] Sarah Meiklejohn and Kazue Sako, editors. *FC 2018: 22nd International Conference on Financial Cryptography and Data Security*, volume 10957 of *Lecture Notes in Computer Science*, Nieuwpoort, Curaçao, February 26 – March 2, 2018. Springer, Heidelberg, Germany.

[Nak08a] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[Nak08b] Satoshi Nakamoto. Bitcoin p2p e-cash paper, 2008. `https://www.metzdowd.com/pipermail/cryptography/2008-October/014810.html`.

[NBF+16] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.

[NEO18] NEO. How to become a neo consensus node, 2018. https://medium.com/neo-smart-economy/how-to-become-a-consensus-node-27e5317722e6.

[Nes18] Mark Nesbitt. Vertcoin (vtc) was successfully 51% attacked, 2018. https://medium.com/coinmonks/vertcoin-vtc-is-currently-being-51-attacked-53ab633c08a4.

[Nes19] Mark Nesbitt. Deep chain reorganization detected on ethereum classic (etc), 2019. https://blog.coinbase.com/ethereum-classic-etc-is-currently-being-51-attacked-33be13ce32de.

[Nic14] Houy Nicolas. The economics of bitcoin transaction fees. *SSRN Electronic Journal*, 02 2014. doi:10.2139/ssrn.2400519.

[NKMS15] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. Cryptology ePrint Archive, Report 2015/796, 2015. https://eprint.iacr.org/2015/796.

[NR18] Jesper Buus Nielsen and Vincent Rijmen, editors. *Advances in Cryptology – EUROCRYPT 2018, Part II*, volume 10821 of *Lecture Notes in Computer Science*, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[NRTV07] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. Algorithmic game theory, 2007. *Book available for free online*, 2007.

[NVV18] Neha Narula, Willy Vasquez, and Madars Virza. zkledger: Privacy-preserving auditing for distributed ledgers. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 65–80, 2018.

[Osb18] Charlie Osborne. Bitcoin gold suffers double spend attacks, $17.5 million lost, 2018. https://www.zdnet.com/article/bitcoin-gold-hit-with-double-spend-attacks-18-million-lost/.

[Par16] Luke Parker. Bitcoin stealing malware evolves again. https://bravenewcoin.com/news/bitcoin-stealing-malware-evolves-again/, 2016. [Online; accessed 1-Sep-2018].

[PDNHJ18] Cristina Pérez-Solà, Sergi Delgado-Segura, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomart. Another coin bites the dust: An analysis of dust in UTXO based cryptocurrencies. Cryptology ePrint Archive, Report 2018/513, 2018. https://eprint.iacr.org/2018/513.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany. doi:10.1007/3-540-46766-1_9.

[PKF+18] Sunoo Park, Albert Kwon, Georg Fuchsbauer, Peter Gazi, Joël Alwen, and Krzysztof Pietrzak. SpaceMint: A cryptocurrency based on proofs of space. In Meiklejohn and Sako [MS18], pages 480–499. doi:10.1007/978-3-662-58387-6_26.

[PS17a] Rafael Pass and Elaine Shi. FruitChains: A fair blockchain. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *36th ACM Symposium Annual on Principles of Distributed Computing*, pages 315–324, Washington, DC, USA, July 25–27, 2017. Association for Computing Machinery. doi:10.1145/3087801.3087809.

[PS17b] Rafael Pass and Elaine Shi. The sleepy model of consensus. In Takagi and Peyrin [TP17], pages 380–409. doi:10.1007/978-3-319-70697-9_14.

[PSL80a] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[PSL80b] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. URL: http://doi.acm.org/10.1145/322186.322188, doi:10.1145/322186.322188.

[PSP+71] Vilfredo Pareto, Ann S Schwier, Alfred N Page, et al. *Manual of political economy*. Macmillan London, 1971.

[PSs17] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Coron and Nielsen [CN17], pages 643–673. doi:10.1007/978-3-319-56614-6_22.

[PvW08] Wouter Penard and Tim van Werkhoven. On the secure hash algorithm family. *Cryptography in Context*, pages 1–18, 2008.

[Riz15] Peter R Rizun. A transaction fee market exists without a block size limit, 2015.

[RL11] Derek D. Reed and James K. Luiselli. *Temporal Discounting*, pages 1474–1474. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-79061-9_3162.

[RLK15] Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors. *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, Denver, CO, USA, October 12–16, 2015. ACM Press.

[Ros73] R. W. Rosenthal. A class of games possessing pure-strategy nash equilibria. *International Journal of Game Theory*, 2:65–67, 1973.

[Rou16] Tim Roughgarden. *Twenty lectures on algorithmic game theory*. Cambridge University Press, 2016.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany. doi:10.1007/0-387-34805-0_22.

[Ser16] Internal Revenue Service. Federal tax compliance research: Tax gap estimates for tax years 2008–2010, 2016.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SJ93] Philip D Straffin Jr. *Game theory and strategy*, volume 36. MAA, 1993.

[SL17] Fabian Schuh and Daniel Larimer. Bitshares 2.0: General overview, 2017.

[SSZ16] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In Grossklags and Preneel [GP16], pages 515–532.

[Ste18] Steem. Steem whitepaper, 2018. https://steem.com/steem-whitepaper.pdf.

[Stu20a] Aleksey Studnev. Attacker stole 807k etc in ethereum classic 51% attack, 2020. https://bitquery.io/blog/attacker-stole-807k-etc-in-ethereum-classic-51-attack.

[Stu20b] Aleksey Studnev. Ethereum classic attack, 8 august: Catch me if you can, 2020. https://bitquery.io/blog/ethereum-classic-attack-8-august-catch-me-if-you-can.

[Tay13] Michael Bedford Taylor. Bitcoin and the age of bespoke silicon. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 16. IEEE Press, 2013.

[TBB+17] Joshua Tan, Lujo Bauer, Joseph Bonneau, Lorrie Faith Cranor, Jeremy Thomas, and Blase Ur. Can unicorns help users compare crypto key fingerprints? In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 3787–3798. ACM, 2017.

[Tez20] Tezos. Proof-of-stake in tezos, 2020. URL: https://tezos.gitlab.io/whitedoc/proof_of_stake.html.

[Tod16] Peter Todd. Making utxo set growth irrelevant with low-latency delayed txo commitments, 2016. https://petertodd.org/2016/delayed-txo-commitments.

[TP17] Tsuyoshi Takagi and Thomas Peyrin, editors. *Advances in Cryptology – ASIACRYPT 2017, Part II*, volume 10625 of *Lecture Notes in Computer Science*, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[Tre18a] Trezor. Trezor. https://trezor.io/, 2018. [Online; accessed 1-Sep-2018].

[Tre18b] Trezor. Trezor developer's guide. https://wiki.trezor.io/Developers_guide, 2018. [Online; accessed 1-Sep-2018].

[Tre18c] Trezor. Trezor user manual. https://wiki.trezor.io/User_manual, 2018. [Online; accessed 1-Sep-2018].

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[UKA07] Ersin Uzun, Kristiina Karvonen, and N. Asokan. Usability analysis of secure pairing methods. In Sven Dietrich and Rachna Dhamija, editors, *FC 2007: 11th International Conference on Financial Cryptography and Data Security*, volume 4886 of *Lecture Notes in Computer Science*, pages 307–324, Scarborough, Trinidad and Tobago, February 12–16, 2007. Springer, Heidelberg, Germany.

[Vas14] Pavel Vasin. Blackcoin's proof-of-stake protocol v2. *URL: https://blackcoin. co/blackcoin-pos-protocol-v2-whitepaper. pdf*, 2014.

[VBC+16] Marie Vasek, Joseph Bonneau, Ryan Castellucci, Cameron Keith, and Tyler Moore. The bitcoin brain drain: Examining the use and abuse of bitcoin brain wallets. In Grossklags and Preneel [GP16], pages 609–618.

[Voe20] Zack Voell. Ethereum classic hit by third 51% attack in a month, 2020. `https://www.coindesk.com/ethereum-classic-blockchain-subject-to-yet-another-51-attack`.

[Vol18] Sergei Volotikin. Software attacks on hardware wallets. *Black Hat USA 2018*, 2018.

[VS13] Nicolas Van Saberhagen. Cryptonote v 2.0, 2013.

[VTM14] Marie Vasek, Micah Thornton, and Tyler Moore. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In Böhme et al. [BBMS14], pages 57–71. `doi:10.1007/978-3-662-44774-1_5`.

[WC14] John Ross Wallrabenstein and Chris Clifton. Privacy preserving Tâtonnement - A cryptographic construction of an incentive compatible market. In Christin and Safavi-Naini [CS14], pages 399–416. `doi:10.1007/978-3-662-45472-5_26`.

[Wik20] Bitcoin Wiki. How to cheaply consolidate coins to reduce miner fees, 2020. `https://en.bitcoin.it/wiki/How_to_cheaply_consolidate_coins_to_reduce_miner_fees`.

[Wil14] Zak Wilcox. Proving your bitcoin reserves, 2014.

[Wil16] Jeffrey Wilcke. The ethereum network is currently undergoing a dos attack, 2016. https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/.

[WKCC18] Karl Wüst, Kari Kostiainen, Vedran Capkun, and Srdjan Capkun. PRCash: Centrally-issued digital currency with privacy and regulation. Cryptology ePrint Archive, Report 2018/412, 2018. https://eprint.iacr.org/2018/412.

[WKK+16] Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors. *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Vienna, Austria, October 24–28, 2016. ACM Press.

[Woo14] Gavin Wood. Ethereum yellow paper, 2014.

[Wui18] Peter Wuille. Hierarchical Deterministic Wallets. https://en.bitcoin.it/wiki/BIP_0032, 2018. [Online; accessed 1-Sep-2018].

[Zah18] Joachim Zahnentferner. Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. Cryptology ePrint Archive, Report 2018/262, 2018. https://eprint.iacr.org/2018/262.

[Zen18] ZenCash. Zencash statement on double spend attack, 2018. https://blog.zencash.com/zencash-statement-on-double-spend-attack/.

[ZET+19] Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors. *FC 2018 Workshops*, volume 10958 of *Lecture Notes in Computer Science*, Nieuwpoort, Curaçao, March 2, 2019. Springer, Heidelberg, Germany.